Microsoft SQL Server 2019 to Amazon Aurora PostgreSQL Migration Playbook

SQL Server to Aurora PostgreSQL Migration Playbook

SQL Server to Aurora PostgreSQL Migration Playbook: Microsoft SQL Server 2019 to Amazon Aurora PostgreSQL Migration Playbook

Table of Contents

Overview **1**
Tables of Feature Compatibility 2
Feature Compatibility Legend 2
AWS SCT and AWS DMS Automation Level Legend 3
Migration Tools and Services **5**
AWS Schema Conversion Tool 5
Download the Software and Drivers 5
Configure AWS SCT 6
Create a New Migration Project 6
AWS SCT Action Code Index 10
Creating Tables 11
Data Types 12
Collations 13
PIVOT and UNPIVOT 14
TOP and FETCH 14
Cursors 15
Flow Control 16
Transaction Isolation 17
Stored Procedures 18
Triggers 19
MERGE 20
Query Hints 20
Full-Text Search 21
Indexes 21
Partitioning 23
Backup 24
SQL Server Mail 24
Graph 25
SQL Server Agent 25
Service Broker 26
XML 26
Constraints 27
Linked Servers 28
Synonyms 28

AWS Database Migration Service ... 28
Migration Tasks Performed by AWS DMS ... 29
How AWS DMS Works ... 30
Latest Updates ... 31
Amazon RDS on Outposts ... 32
How It Works ... 32
Amazon RDS Proxy ... 33
Amazon RDS Proxy Benefits ... 34
How Amazon RDS Proxy Works ... 34
Amazon Aurora Serverless v1 ... 35
How to Provision ... 37
Amazon Aurora Backtrack ... 39
Backtrack Window ... 41
Backtracking Limitations ... 42
Migration Quick Tips ... 43
ANSI SQL ... 44
Case Sensitivity Differences for SQL Server and PostgreSQL ... 44
SQL Server Constraints and PostgreSQL Table Constraints ... 45
SQL Server Usage ... 45
PostgreSQL Usage ... 49
Summary ... 56
Creating Tables ... 56
SQL Server Usage ... 57
PostgreSQL Usage ... 61
Summary ... 66
Common Table Expressions ... 67
SQL Server Usage ... 67
PostgreSQL Usage ... 70
Data Types ... 75
SQL Server Usage ... 75
PostgreSQL Usage ... 77
Summary ... 83
Derived Tables ... 84
SQL Server Usage ... 85
PostgreSQL Usage ... 85
GROUP BY ... 85

- SQL Server Usage ... 86
- PostgreSQL Usage ... 90
- Summary ... 92
- Table JOIN ... 93
  - SQL Server Usage ... 94
  - PostgreSQL Usage ... 99
  - Summary ... 102
- SQL Server Temporal Tables and PostgreSQL Triggers ... 103
  - SQL Server Usage ... 103
  - PostgreSQL Usage ... 105
- Views ... 105
  - SQL Server Usage ... 105
  - PostgreSQL Usage ... 109
  - Summary ... 111
- Window Functions ... 112
  - SQL Server Usage ... 112
  - PostgreSQL Usage ... 114

T-SQL ... 119

- SQL Server Service Broker Essentials and PostgreSQL AWS Lambda or DB links ... 120
  - SQL Server Usage ... 120
  - PostgreSQL Usage ... 123
- SQL Server Cast and Convert and PostgreSQL CAST and CONVERSION ... 124
  - SQL Server Usage ... 124
  - PostgreSQL Usage ... 126
  - Summary ... 127
- SQL Server Common Library Runtime and PostgreSQL PL/Perl ... 128
  - SQL Server Usage ... 128
  - PostgreSQL Usage ... 129
- SQL Server Collations and PostgreSQL Encoding ... 130
  - SQL Server Usage ... 130
  - PostgreSQL Usage ... 133
  - Summary ... 137
- Cursors ... 138
  - SQL Server Usage ... 138
  - PostgreSQL Usage ... 139
  - Summary ... 144

Date and Time Functions 146
SQL Server Usage 146
PostgreSQL Usage 148
Summary 150
String Functions 150
SQL Server Usage 150
PostgreSQL Usage 153
Summary 156
Databases and Schemas 157
SQL Server Usage 157
PostgreSQL Usage 160
SQL Server Dynamic SQL and PostgreSQL EXECUTE and PREPARE 162
SQL Server Usage 162
PostgreSQL Usage 165
Summary 167
Transactions 169
SQL Server Usage 169
PostgreSQL Usage 173
Summary 179
SQL Server Synonyms and PostgreSQL Views, Types, and Functions 180
SQL Server Usage 180
PostgreSQL Usage 182
DELETE and UPDATE FROM 184
SQL Server Usage 184
PostgreSQL Usage 187
Summary 189
Stored Procedures 189
SQL Server Usage 189
PostgreSQL Usage 193
Summary 197
Error Handling 199
SQL Server Usage 199
PostgreSQL Usage 204
Summary 206
SQL Server Flow Control and PostgreSQL Control Structures 207
SQL Server Usage 207

PostgreSQL Usage ... 210
Summary ... 211
Full-Text Search ... 213
SQL Server Usage ... 213
PostgreSQL Usage ... 217
SQL Server Graph ... 220
SQL Server Usage ... 220
PostgreSQL Usage ... 222
JSON and XML ... 222
SQL Server Usage ... 222
PostgreSQL Usage ... 225
Summary ... 230
MERGE ... 231
SQL Server Usage ... 231
PostgreSQL Usage ... 234
PIVOT and UNPIVOT ... 235
SQL Server Usage ... 235
PostgreSQL Usage ... 239
Triggers ... 242
SQL Server Usage ... 242
PostgreSQL Usage ... 246
Summary ... 250
SQL Server TOP and FETCH and PostgreSQL LIMIT and OFFSET ... 251
SQL Server Usage ... 251
PostgreSQL Usage ... 254
Summary ... 257
User-Defined Functions ... 257
SQL Server Usage ... 257
PostgreSQL Usage ... 261
User-Defined Types ... 262
SQL Server Usage ... 262
PostgreSQL Usage ... 265
Sequences and Identity ... 268
SQL Server Usage ... 268
PostgreSQL Usage ... 273
Summary ... 278

Configuration ... 280
- Upgrades ... 280
  - SQL Server Usage ... 280
  - Upgrade in-place ... 280
  - PostgreSQL Usage ... 282
  - Summary ... 285
- Session Options ... 287
  - SQL Server Usage ... 287
  - PostgreSQL Usage ... 290
  - Summary ... 291
- Database Options ... 293
  - SQL Server Usage ... 293
  - PostgreSQL Usage ... 294
- Server Options in SQL Server and Parameter Groups in Amazon Aurora ... 294
  - SQL Server Usage ... 294
  - PostgreSQL Usage ... 296

High Availability and Disaster Recovery ... 301
- Backup and Restore ... 301
  - SQL Server Usage ... 301
  - PostgreSQL Usage ... 305
  - Summary ... 310
- High Availability Essentials ... 315
  - SQL Server Usage ... 315
  - PostgreSQL Usage ... 320
  - Summary ... 327

Indexes ... 328
- SQL Server Usage ... 328
  - Clustered Indexes ... 329
  - Non-Clustered Indexes ... 330
  - Filtered Indexes and Covering Indexes ... 331
  - Indexes On Computed Columns ... 331
- PostgreSQL Usage ... 333
  - Cluster Table ... 333
  - B-tree Indexes ... 334
  - Column and Multiple Column Secondary Indexes ... 335
  - Expression Indexes and Partial Indexes ... 335

Partial Indexes 336
BRIN Indexes 337
Summary 337
Management 339
SQL Server Agent and PostgreSQL Scheduled Lambda 339
SQL Server Usage 339
PostgreSQL Usage 340
Alerting 340
SQL Server Usage 340
PostgreSQL Usage 342
Database Mail 345
SQL Server Usage 345
PostgreSQL Usage 349
ETL 354
SQL Server Usage 354
PostgreSQL Usage 356
SQL Server Export and Import with Text Files and PostgreSQL pg_dump and pg_restore 361
SQL Server Usage 361
PostgreSQL Usage 363
Summary 366
Viewing Server Logs 366
SQL Server Usage 367
PostgreSQL Usage 368
Maintenance Plans 370
SQL Server Usage 371
PostgreSQL Usage 373
Summary 375
Monitoring 376
SQL Server Usage 376
PostgreSQL Usage 380
SQL Server Resource Governor and PostgreSQL Dedicated Amazon Aurora Clusters or Aurora Read-Replicas 381
SQL Server Usage 382
PostgreSQL Usage 384
SQL Server Linked Servers and PostgreSQL DBLink and FDWrapper 389
SQL Server Usage 389

PostgreSQL Usage 392
Scripting 394
SQL Server Usage 394
PostgreSQL Usage 395
Performance Tuning 399
Run Plans 399
SQL Server Usage 399
PostgreSQL Usage 401
SQL Server Query Hints and Plan Guides and PostgreSQL DB Query Planning 404
SQL Server Usage 404
PostgreSQL Usage 408
SQL Server Managing Statistics and PostgreSQL Table Statistics 409
SQL Server Usage 409
PostgreSQL Usage 411
Summary 413
Physical Storage 415
SQL Server Columnstore Index and PostgreSQL Columnstore 415
SQL Server Usage 415
PostgreSQL Usage 416
SQL Server Indexed Views and PostgreSQL Materialized Views 416
SQL Server Usage 416
PostgreSQL Usage 417
Summary 419
SQL Server Partitioning and PostgreSQL Partitions or Table Inheritance 421
SQL Server Usage 421
PostgreSQL Usage 423
Summary 437
Security 439
Column Encryption 439
SQL Server Usage 439
PostgreSQL Usage 441
Data Control Language 443
SQL Server Usage 443
PostgreSQL Usage 444
Transparent Data Encryption 448
SQL Server Usage 448

PostgreSQL Usage ... 449
Users and Roles ... 453
SQL Server Usage ... 454
PostgreSQL Usage ... 455
Summary ... 456
Deprecated Features List ... 458
Migration Quick Tips ... 459
Management ... 459
SQL ... 459

Overview

The first section of this document provides an overview of AWS Schema Conversion Tool (AWS SCT) and the AWS Database Migration Service (AWS DMS) tools for automating the migration of schema, objects and data. The remainder of the document contains individual sections for the source database features and their Aurora counterparts. Each section provides a short overview of the feature, examples, and potential workaround solutions for incompatibilities.

You can use this playbook either as a reference to investigate the individual action codes generated by AWS SCT, or to explore a variety of topics where you expect to have some incompatibility issues. When using AWS SCT, you may see a report that lists Action codes , which indicates some manual conversion is required, or that a manual verification is recommended. For your convenience, this Playbook includes an AWS SCT Action Code Index section providing direct links to the relevant topics that discuss the manual conversion tasks needed to address these action codes. Alternatively, you can explore the Tables of Feature Compatibility section that provides high-level graphical indicators and descriptions of the feature compatibility between the source database and Aurora. It also includes a graphical compatibility indicator and links to the actual sections in the playbook.

The Migration Quick Tips section provides a list of tips for administrators or developers who have little experience with Aurora (PostgreSQL or MySQL). It briefly highlights key differences between the source database and Aurora that they are likely to encounter.

Note that not all of the source database features are fully compatible with Aurora or have simple workarounds. From a migration perspective, this document doesn't yet cover all source database features and capabilities.

This database migration playbook covers the following topics:

- Migration Tools and Services
- ANSI SQL
- T-SQL
- High Availability and Disaster Recovery
- Configuration
- Indexes
- Management

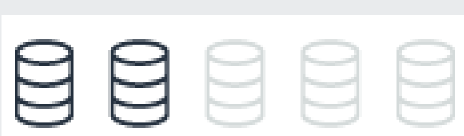
- Performance Tuning
- Physical Storage
- Security
- SQL Server 2018 Deprecated Features List
- Migration Quick Tips

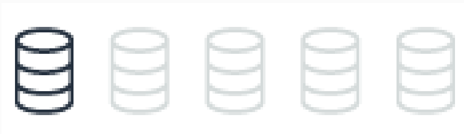
Disclaimer

The various code snippets, commands, guides, best practices, and scripts included in this document should be used for reference only and are provided as-is without warranty. Test all of the code, commands, best practices, and scripts outlined in this document in a non-production environment first. Amazon and its affiliates are not responsible for any direct or indirect damage that may occur from the information contained in this document.

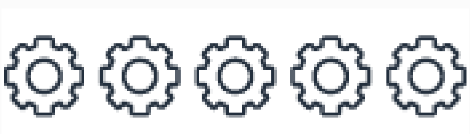
Tables of Feature Compatibility

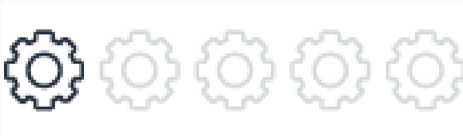
Feature Compatibility Legend

Automation level icon	Description
	**Very high compatibility**. None or minimal low-risk and low-effort rewrites needed.
	**High compatibility**. Some low-risk rewrites needed, easy workarounds exist for incompati ble features.
	**Medium compatibility**. More involved low-medium risk rewrites needed, some redesign may be needed for incompatible features.
	**Low compatibility**. Medium to high risk rewrites needed, some incompatible features

Automation level icon	Description
	require redesign and reasonable-effort workarounds exist.
	**Very low compatibility**. High risk and/or high-effort rewrites needed, some features require redesign and workarounds are challenging.
	Not compatible. No practical workarounds yet, may require an application level architectural solution to work around incompatibilities.

AWS SCT and AWS DMS Automation Level Legend

Automation level icon	Description
	**Full automation**. AWS SCT performs fully automatic conversion, no manual conversion needed.
	**High automation**. Minor, simple manual conversions may be needed.
	**Medium automation**. Low-medium complexity manual conversions may be needed.
	**Low automation**. Medium-high complexity manual conversions may be needed.

Automation level icon	Description
	**Very low automation**. High risk or complex manual conversions may be needed.
	No automation. Not currently supported by AWS SCT, manual conversion is required for this feature.

Migration Tools and Services

Topics

- AWS Schema Conversion Tool
- AWS SCT Action Code Index
- AWS Database Migration Service
- Amazon RDS on Outposts
- Amazon RDS Proxy
- Amazon Aurora Serverless v1
- Amazon Aurora Backtrack
- Migration Quick Tips

AWS Schema Conversion Tool

The AWS Schema Conversion Tool (AWS SCT) is a Java utility that connects to source and target databases, scans the source database schema objects (tables, views, indexes, procedures, and so on), and converts them to target database objects.

This section provides a step-by-step process for using AWS SCT to migrate an SQL Server database to an Aurora PostgreSQL database cluster. Since AWS SCT can automatically migrate most of the database objects, it greatly reduces manual effort.

We recommend to start every migration with the process outlined in this section and then use the rest of the Playbook to further explore manual solutions for objects that couldn't be migrated automatically. For more information, see Schema Conversion Tool user guide.

> **Note**
>
> This walkthrough uses the AWS DMS Sample Database. You can download it from GitHub.

Download the Software and Drivers

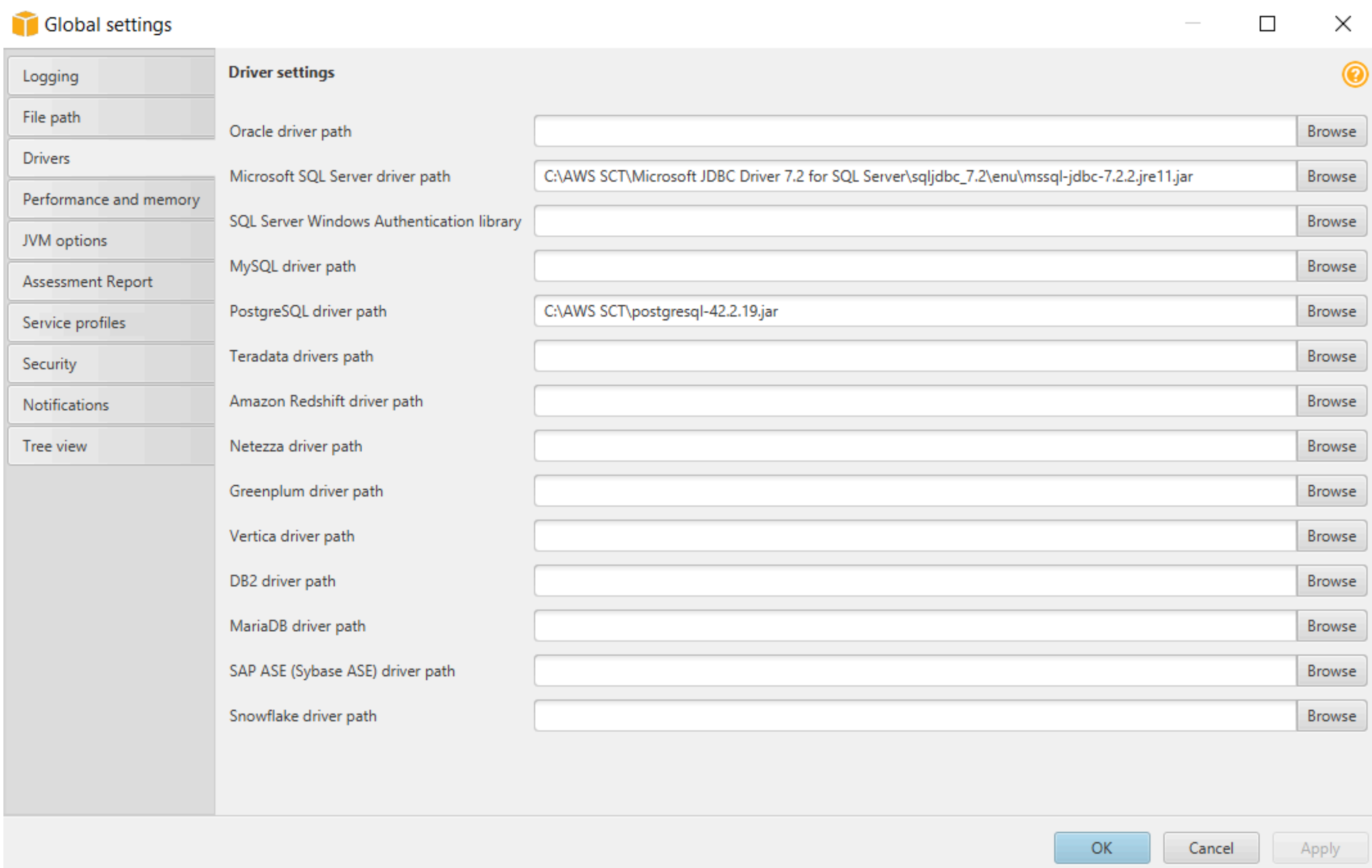
Download and install AWS SCT from the Schema Conversion Tool user guide.

Download the Microsoft SQL Server and PostgreSQL drivers.

Find other supported drivers in the Schema Conversion Tool user guide.

Configure AWS SCT

1. Start AWS Schema Conversion Tool (AWS SCT).
2. Choose **Settings** and then choose **Global settings**.
3. On the left navigation bar, choose **Drivers**.
4. Enter the paths for the SQL Server and PostgreSQL drivers downloaded in the first step.



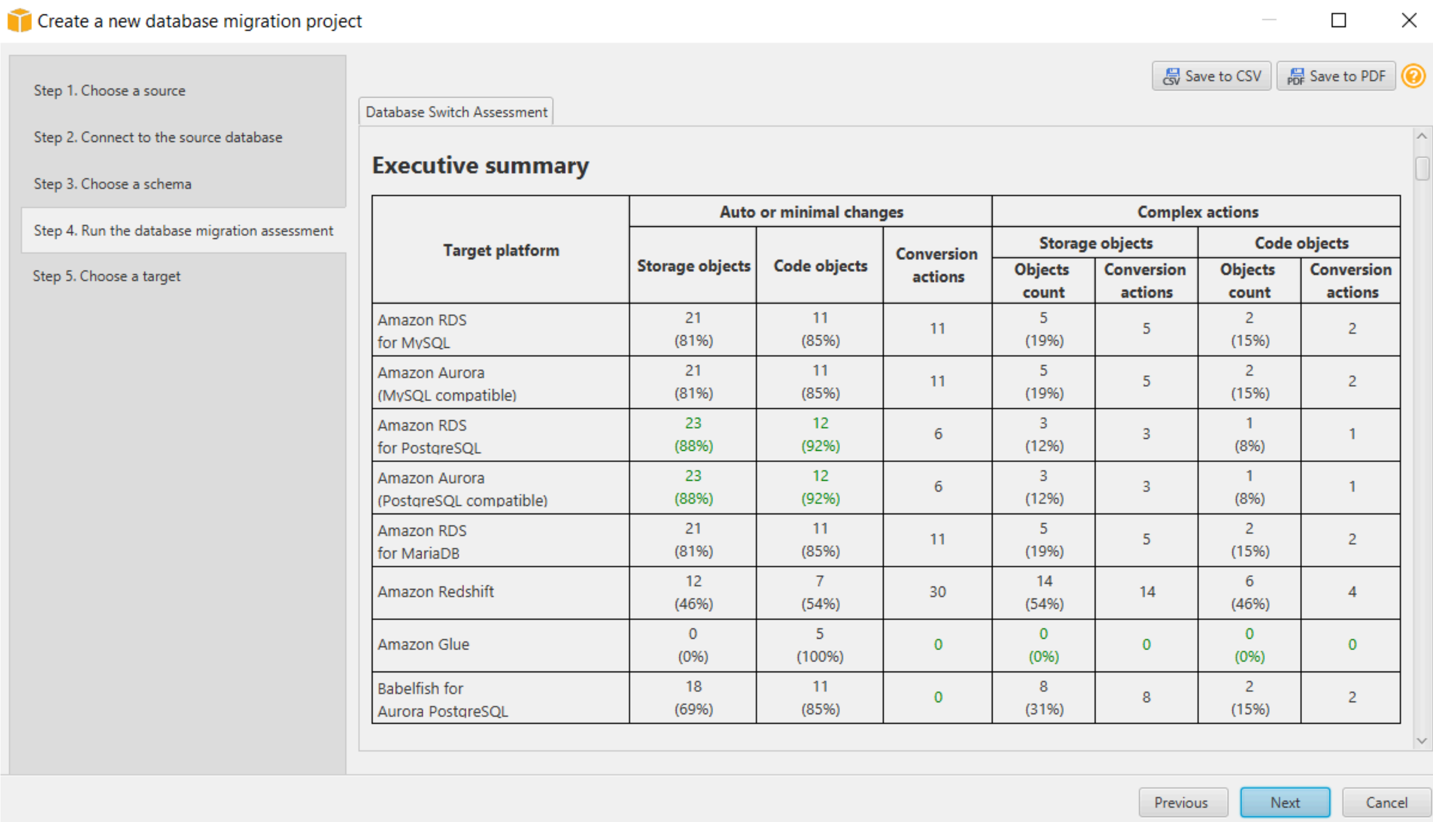
5. Choose **Apply** and then **OK**.

Create a New Migration Project

1. Choose **File**, and then choose **New project wizard**. Alternatively, use the keyboard shortcut **Ctrl +W**.

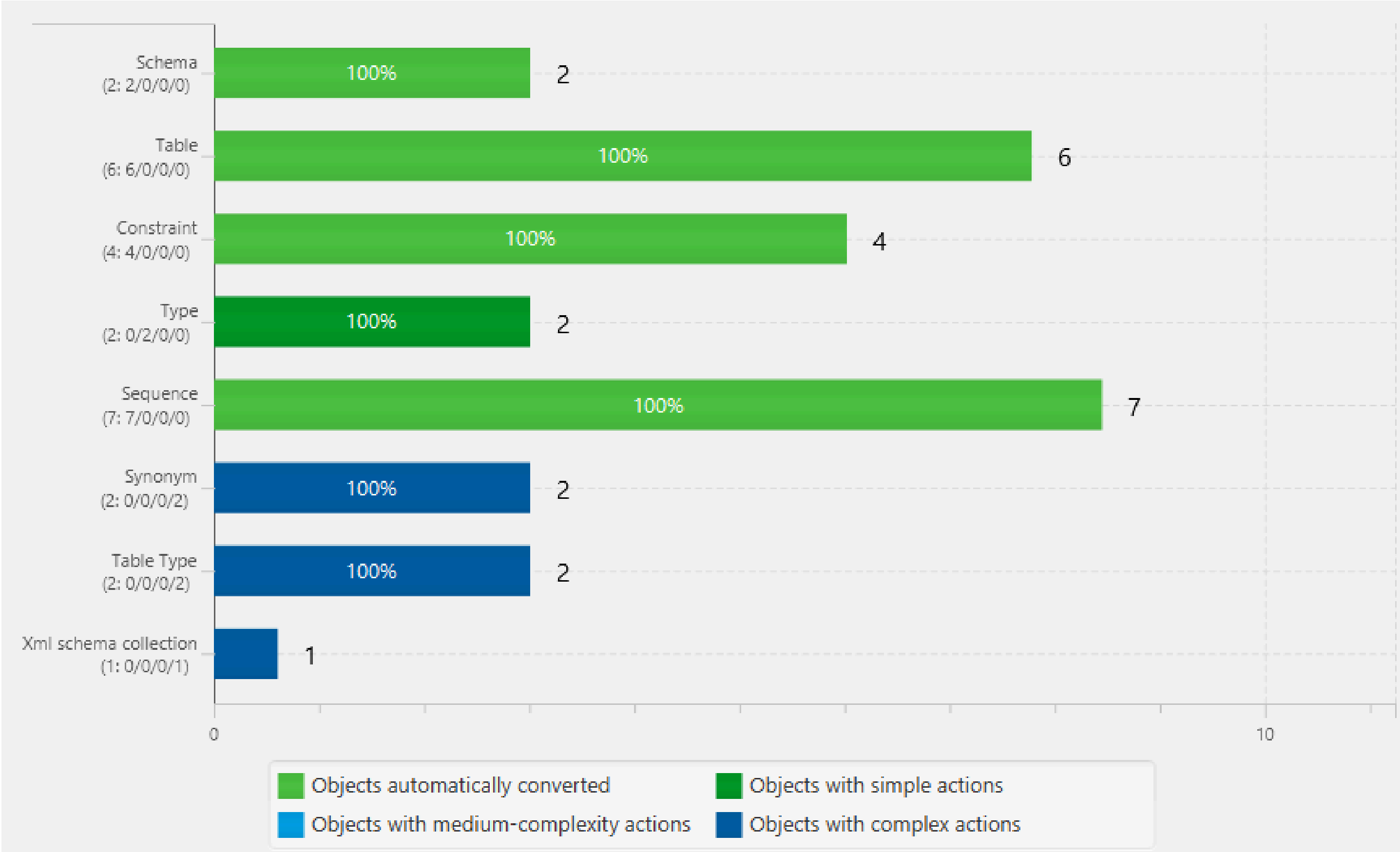
2. Enter a project name and select a location for the project files. For **Source engine**, choose **Microsoft SQL Server**, and then choose **Next**.
3. Enter connection details for the source SQL Server database and choose **Test connection** to verify. Choose **Next**.
4. Select the schema or database to migrate and choose **Next**.

The progress bar displays the objects that AWS SCT analyzes. When AWS SCT completes the analysis, the application displays the database migration assessment report. Read the Executive summary and other sections. Note that the information on the screen is only partial. To read the full report, including details of the individual issues, choose **Save to PDF** at the top right and open the PDF document.

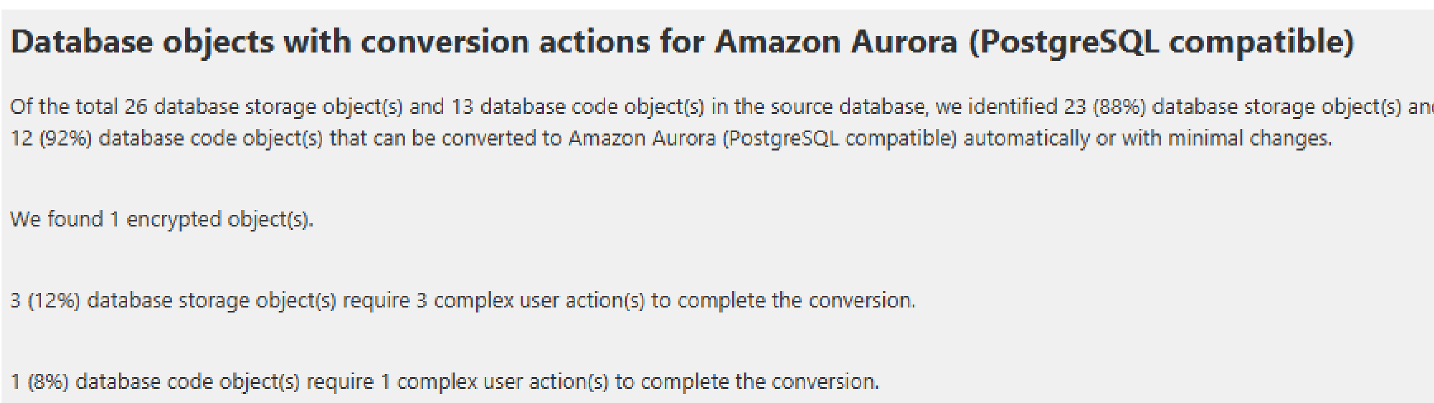


Target platform	Auto or minimal changes			Complex actions			
	Storage objects	Code objects	Conversion actions	Storage objects		Code objects	
				Objects count	Conversion actions	Objects count	Conversion actions
Amazon RDS for MySQL	21 (81%)	11 (85%)	11	5 (19%)	5	2 (15%)	2
Amazon Aurora (MySQL compatible)	21 (81%)	11 (85%)	11	5 (19%)	5	2 (15%)	2
Amazon RDS for PostgreSQL	23 (88%)	12 (92%)	6	3 (12%)	3	1 (8%)	1
Amazon Aurora (PostgreSQL compatible)	23 (88%)	12 (92%)	6	3 (12%)	3	1 (8%)	1
Amazon RDS for MariaDB	21 (81%)	11 (85%)	11	5 (19%)	5	2 (15%)	2
Amazon Redshift	12 (46%)	7 (54%)	30	14 (54%)	14	6 (46%)	4
Amazon Glue	0 (0%)	5 (100%)	0	0 (0%)	0	0 (0%)	0
Babelfish for Aurora PostgreSQL	18 (69%)	11 (85%)	0	8 (31%)	8	2 (15%)	2

Scroll down to the **Database objects with conversion actions for Amazon Aurora (PostgreSQL compatible)** section.



Scroll further down to the **Detailed recommendations for Amazon Aurora (PostgreSQL compatible) migrations** section.



Database objects with conversion actions for Amazon Aurora (PostgreSQL compatible)

Of the total 26 database storage object(s) and 13 database code object(s) in the source database, we identified 23 (88%) database storage object(s) and 12 (92%) database code object(s) that can be converted to Amazon Aurora (PostgreSQL compatible) automatically or with minimal changes.

We found 1 encrypted object(s).

3 (12%) database storage object(s) require 3 complex user action(s) to complete the conversion.

1 (8%) database code object(s) require 1 complex user action(s) to complete the conversion.

Return to AWS SCT and choose **Next**. Enter the connection details for the target Aurora PostgreSQL database and choose **Finish**.

When the connection is complete, AWS SCT displays the main window. In this interface, you can explore the individual issues and recommendations discovered by AWS SCT.

Choose the schema, open the context (right-click) menu, and then choose **Create report** to create a report tailored for the target database type. You can view this report in AWS SCT.

The progress bar updates while the report is generated.

AWS SCT displays the executive summary page of the database migration assessment report.

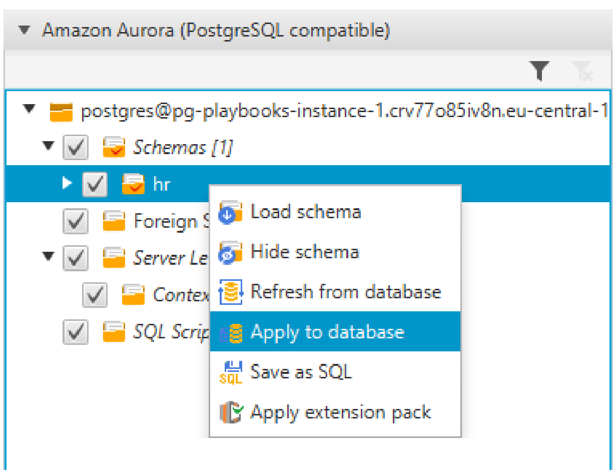
Choose **Action items**. In this window, you can investigate each issue in detail and view the suggested course of action. For each issue, drill down to view all instances of that issue.

Choose the database name, open the context (right-click) menu, and choose **Convert schema**. Make sure that you uncheck the `sys` and `information_schema` system schemas. Aurora PostgreSQL already has an `information_schema` schema.

This step doesn't make any changes to the target database.

On the right pane, AWS SCT displays the new virtual schema as if it exists in the target database. Drilling down into individual objects displays the actual syntax generated by AWS SCT to migrate the objects.

Choose the database on the right pane, open the context (right-click) menu, and choose either **Apply to database** to automatically run the conversion script against the target database, or choose **Save as SQL** to save to an SQL file.

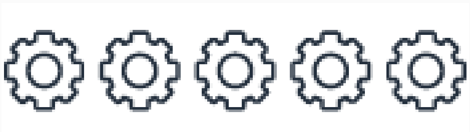
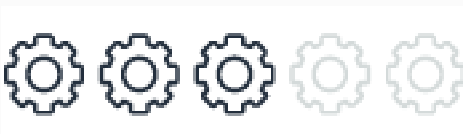


We recommend saving to an SQL file because you can verify and QA the converted code. Also, you can make the adjustments needed for objects that couldn't be automatically converted.

For more information, see the Schema Conversion Tool user guide.

AWS SCT Action Code Index

The following table shows the icons we use to describe the automation levels of AWS Schema Conversion Tool (AWS SCT) and AWS Database Migration Service (AWS DMS).

Automation level icon	Description
	**Full automation**. AWS SCT performs fully automatic conversion, no manual conversion needed.
	**High automation**. Minor, simple manual conversions may be needed.
	**Medium automation**. Low-medium complexity manual conversions may be needed.
	**Low automation**. Medium-high complexity manual conversions may be needed.
	**Very low automation**. High risk or complex manual conversions may be needed.
	No automation. Not currently supported by AWS SCT, manual conversion is required for this feature.

The following sections list the AWS Schema Conversion Tool Action codes for topics that are covered in this playbook.

Note

The links in the table point to the Microsoft SQL Server topic pages, which are immediately followed by the PostgreSQL pages for the same topics.

Creating Tables

AWS SCT automatically converts the most commonly used constructs of the `CREATE TABLE` statement as both SQL Server and Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) support the entry level American National Standards Institute (ANSI) compliance. These items include table names, containing security schema or database, column names, basic column data types, column and table constraints, column default values, primary, `UNIQUE`, and foreign keys. Some changes may be required for computed columns and global temporary tables.

For more information, see Creating Tables.

Action code	Action message
7659	If you use recursion, make sure that table variables in your source database and temporary tables in your target database have the same scope.
7665	PostgreSQL doesn’t support `FILESTREAM` clauses. AWS SCT skips `FILESTREAM` clauses in the converted code.
7678	AWS SCT replaced computed columns with regular columns in the converted code.
7679	AWS SCT replaced computed columns with triggers in the converted code.

Action code	Action message
7680	PostgreSQL doesn't support global temporary tables.
7812	Make sure that you remove the temporary table before the end of the function.
7835	PostgreSQL doesn't support `CREATE TABLE` statements with the `AS FileTable` option.

Data Types

Data type syntax and rules are very similar between SQL Server and Aurora PostgreSQL and most are converted automatically by AWS SCT. Note that date and time handling paradigms are different for SQL Server and Aurora PostgreSQL and require manual verification or conversion. Also note that due to differences in data type behavior between SQL Server and Aurora PostgreSQL, manual verification and strict testing are highly recommended.

For more information, see Data Types.

Action code	Action message
7657	PostgreSQL doesn't support the `hierarchyid` data type.
7658	PostgreSQL doesn't support the `sql_variant` data type.
7662	PostgreSQL doesn't support the `geography` data type.
7664	PostgreSQL doesn't support the `geometry` data type.

Action code	Action message
7690	PostgreSQL doesn't support table types.
7706	AWS SCT can't convert the declaration of a variable of the unsupported %s data type.
7707	AWS SCT can't convert the usage of a variable of the unsupported %s data type.
7708	AWS SCT can't convert the usage of the unsupported %s data type.
7773	AWS SCT can't convert arithmetic operations with dates.
7775	Converted code might lose accuracy compared to the source code.

Collations

The collation paradigms of SQL Server and Aurora PostgreSQL are significantly different. AWS SCT can't migrate collations automatically to PostgreSQL.

For more information, see [SQL Server Collations and PostgreSQL Encoding](#).

Action code	Action message
7646	AWS SCT can't convert collations.

PIVOT and UNPIVOT

Aurora PostgreSQL version 10 doesn't support `PIVOT` and `UNPIVOT` clauses. AWS SCT can't automatically convert `PIVOT` and `UNPIVOT` clauses.

For more information, see PIVOT and UNPIVOT.

Action code	Action message
7905	PostgreSQL doesn't support `PIVOT` clauses for `SELECT` statements.
7906	PostgreSQL doesn't support `UNPIVOT` clauses for `SELECT` statements.

TOP and FETCH

Aurora PostgreSQL supports the non-ANSI compliant but popular with other engines `LIMIT… OFFSET` operator for paging results sets. AWS SCT can't automatically convert some options such as `WITH TIES`. These options require manual conversion.

For more information, see SQL Server TOP and FETCH and PostgreSQL LIMIT and OFFSET.

Action code	Action message
7605	PostgreSQL doesn't support the `WITH TIES` argument in `TOP` clauses.
7796	PostgreSQL doesn't support `TOP` clauses in `UPDATE` statements.

Action code	Action message
7798	PostgreSQL doesn't support TOP clauses in DELETE statements.
7799	PostgreSQL doesn't support TOP clauses in INSERT operators.

Cursors

PostgreSQL has PL/pgSQL cursors that enable you to iterate business logic on rows read from the database. They can encapsulate the query and read the query results a few rows at a time. All access to cursors in PL/pgSQL is performed through cursor variables, which are always of the `refcursor` data type. There are specific options which aren't supported for automatic conversion by AWS SCT.

For more information, see Cursors.

Action code	Action message
7637	PostgreSQL doesn't support global cursors.
7639	PostgreSQL doesn't support dynamic cursors.
7700	AWS SCT can't convert the KEYSET option because PostgreSQL doesn't support changing the membership and order of rows for cursors.
7701	AWS SCT doesn't convert the FAST_FORWARD option because this is a default option for cursors in PostgreSQL.

Action code	Action message
7702	AWS SCT doesn’t convert the `READ_ONLY` option because this is a default option for cursors in PostgreSQL.
7704	PostgreSQL doesn’t support the `OPTIMISTIC` option for cursors.
7705	PostgreSQL doesn’t support the `TYPE_WARNING` option for cursors.
7803	PostgreSQL doesn’t support the `FOR UPDATE` option.

Flow Control

Although the flow control syntax of SQL Server differs from Aurora PostgreSQL, AWS SCT can convert most constructs automatically including loops, command blocks, and delays. Aurora PostgreSQL doesn’t support the `GOTO` and `WAITFOR TIME` commands, which require manual conversion.

For more information, see [SQL Server Flow Control and PostgreSQL Control Structures](#).

Action code	Action message
7628	PostgreSQL doesn’t support `GOTO` statements.
7691	PostgreSQL doesn’t support the `WAITFOR TIME` feature.
7801	Make sure that your table isn’t locked by an open cursor.

Action code	Action message
7802	Make sure that you delete the table that you created within the procedure before the end of the procedure.
7810	PostgreSQL doesn't support `SET NOCOUNT OFF` statements.
7821	AWS SCT can't convert the `WAITFOR` operator with a variable.
7826	AWS SCT can't convert the default value of the `DateTime` variable.
7827	AWS SCT can't convert default values.

Transaction Isolation

Aurora PostgreSQL supports the four transaction isolation levels specified in the SQL:92 standard: `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`, all of which are automatically converted by AWS SCT. Also, AWS SCT converts `BEGIN / COMMIT` and `ROLLBACK` commands that use slightly different syntax. Manual conversion is required for named, marked, and delayed durability transactions that aren't supported by Aurora PostgreSQL.

For more information, see Transactions.

Action code	Action message
7807	AWS SCT can't convert the transaction management command. PostgreSQL doesn't support explicit transaction management commands such as `BEGIN TRAN`, `SAVE TRAN` in functions.

Stored Procedures

Aurora PostgreSQL stored procedures provide very similar functionality to SQL Server stored procedures. You can automatically convert them with AWS SCT. Manual conversion is required for procedures that use `RETURN` values and some less common `EXECUTE` options such as the `RECOMPILE` and `RESULTS SETS`.

For more information, see Stored Procedures.

Action code	Action message
7640	PostgreSQL doesn't support `EXECUTE` statements with the `WITH RECOMPILE` option.
7641	PostgreSQL doesn't support `EXECUTE` statements with the `RESULT SETS UNDEFINED` option.
7642	PostgreSQL doesn't support `EXECUTE` statements with the `RESULT SETS NONE` option.
7643	PostgreSQL doesn't support `EXECUTE` statements with the `RESULT SETS DEFINITION` option.
7672	PostgreSQL doesn't support `EXECUTE` statements that run a character string.
7695	PostgreSQL doesn't support support the call of a procedure as a variable.
7800	PostgreSQL doesn't support result sets in the SQL Server style.

Action code	Action message
7830	AWS SCT can't convert arithmetic operations with the `CASE` operand.
7838	AWS SCT can't convert `EXECUTE` statements with `LOGIN` or `USER` options.
7839	Converted code might not work correctly because of parameter names.

Triggers

Aurora PostgreSQL supports `BEFORE` and `AFTER` triggers for `INSERT`, `UPDATE`, and `DELETE`. However, Aurora PostgreSQL triggers differ substantially from SQL Server's triggers. You can migrate the most common use cases with minimal code changes.

For more information, see Triggers.

Action code	Action message
7809	PostgreSQL doesn't support `INSTEAD OF` triggers on tables.
7832	AWS SCT can't convert `INSTEAD OF` triggers on views.
7909	AWS SCT can't convert `UPDATE(column)` or `COLUMNS_UPDATED` statements.

MERGE

Aurora PostgreSQL version 10 doesn't support `MERGE` statements. AWS SCT can't automatically convert these statements. Manual conversion is straightforward in most cases.

For more information, see MERGE.

Action code	Action message
7915	Converted code might produce different results compared to the source code. Make sure that the constraint includes the %s column.
7916	AWS SCT can't emulate the `MERGE` statement using the `INSERT ON CONFLICT` statement.

Query Hints

You can use AWS SCT to convert basic query hints such as index hints, except for data manipulation language (DML) statements. Note that specific optimizations used for SQL Server may be completely inapplicable to a new query optimizer. AWS recommends to start migration testing with all hints removed. Then, selectively apply hints as a last resort if other means such as schema, index, and query optimizations have failed. Plan guides aren't supported by Aurora PostgreSQL.

For more information, see SQL Server Query Hints and Plan Guides and PostgreSQL DB Query Planning.

Action code	Action message
7823	PostgreSQL doesn't support table hints in DML statements.

Full-Text Search

Migrating full-text indexes from SQL Server to Aurora PostgreSQL requires a full rewrite of the code that deals with both creating, managing, and querying full-text indexes. AWS SCT can't automatically convert these statements.

For more information, see Full-Text Search.

Action code	Action message
7688	PostgreSQL doesn't support FREETEXT predicates.

Indexes

Basic non-clustered indexes, which are the most commonly used type of indexes are automatically migrated by AWS SCT. In addition, filtered indexes, indexes with included columns, and some SQL Server specific index options can't be migrated automatically and require manual conversion.

For more information, see Indexes.

Action code	Action message
7675	PostgreSQL doesn't support ASC and DESC sorting options for constraints.
7681	PostgreSQL doesn't support clustered indexes.
7682	PostgreSQL doesn't support the INCLUDE option in indexes.
7781	PostgreSQL doesn't support the PAD_INDEX option in indexes.
7782	PostgreSQL doesn't support the SORT_IN_TEMPDB option in indexes.
7783	PostgreSQL doesn't support the IGNORE_DUP_KEY option in indexes.
7784	PostgreSQL doesn't support the STATISTICS_NORECOMPUTE option in indexes.
7785	PostgreSQL doesn't support the STATISTICS_INCREMENTAL option in indexes.
7786	PostgreSQL doesn't support the DROP_EXISTING option in indexes.
7787	PostgreSQL doesn't support the ONLINE option in indexes.
7788	PostgreSQL doesn't support the ALLOW_ROW_LOCKS option in indexes.
7789	PostgreSQL doesn't support the ALLOW_PAGE_LOCKS option in indexes.
7790	PostgreSQL doesn't support the MAXDOP option in indexes.

Action code	Action message
7791	PostgreSQL doesn't support the `DATA_COMPRESSION` option in indexes.

Partitioning

Aurora PostgreSQL uses table inheritance, some of the physical aspects of partitioning in SQL Server don't apply to Aurora PostgreSQL. For example, the concept of file groups and assigning partitions to file groups. Aurora PostgreSQL supports a much richer framework for table partitioning than SQL Server, with many additional options such as hash partitioning, and sub partitioning.

For more information, see [SQL Server Partitioning and PostgreSQL Partitions or Table Inheritance](#).

Action code	Action message
7910	PostgreSQL doesn't support `NULL` columns for partitioning.
7911	PostgreSQL doesn't support foreign keys referencing partitioned tables.
7912	PostgreSQL doesn't support foreign key references from partitioned tables to other tables.
7913	PostgreSQL doesn't support `LEFT` partitioning.
7914	Converted code might produce different results compared to the source code.

Starting from version 11, PostgreSQL supports NULL columns for partitioning. In this case, you can ignore the action item with the 7910 code and use NULL columns for partitioning in your target tables.

Backup

Migrating from a self-managed backup policy to a Platform as a Service (PaaS) environment such as Aurora PostgreSQL is a complete paradigm shift. You don't need to worry about transaction logs, file groups, disks running out of space, and purging old backups. Amazon Relational Database Service (Amazon RDS) provides guaranteed continuous backup with point in time restore up to 35 days. Therefore, AWS SCT doesn't automatically convert backups.

For more information, see Backup and Restore.

Action code	Action message
7903	PostgreSQL doesn't support functionality similar to SQL Server Backup.

SQL Server Mail

Aurora PostgreSQL doesn't provide native support for sending emails from the database.

For more information, see Database Mail.

Action code	Action message
7900	PostgreSQL doesn't support functionality similar to SQL Server Database Mail.

Graph

AWS SCT doesn’t convert graph database capabilities.

For more information and potential workarounds, see SQL Server Graph and PostgreSQL Apache AGE Extension.

Action code	Action message
7931	AWS SCT can’t convert SQL Graph tables.
7932	AWS SCT can’t convert DML constructs of SQL Graph databases.

SQL Server Agent

Aurora PostgreSQL doesn’t provide functionality similar to SQL Server Agent as an external, cross-instance scheduler. However, Aurora PostgreSQL provides a native, in-database scheduler. It is limited to the cluster scope and can’t be used to manage multiple clusters. Therefore, AWS SCT can’t automatically convert Agent jobs and alerts.

For more information, see SQL Server Agent and PostgreSQL Scheduled Lambda.

Action code	Action message
7902	PostgreSQL doesn’t support functionality similar to SQL Server Agent.

Service Broker

Aurora PostgreSQL doesn't provide a compatible solution to the SQL Server Service Broker. However, you can use DB Links and AWS Lambda to achieve similar functionality.

For more information, see SQL Server Service Broker Essentials.

Action code	Action message
7901	PostgreSQL doesn't support functionality similar to SQL Server Service Broker.

XML

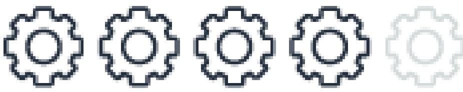
The XML options and features in Aurora PostgreSQL are similar or almost identical to SQL Server `XPATH` and `XQUERY` functions. PostgreSQL doesn't support `FOR` XML clause, the walkaround for that is using `string_agg` instead. In some cases, it might be more efficient to use JSON instead of XML.

For more information, see JSON and XML.

Action code	Action message
7816	PostgreSQL doesn't support methods for the XML data type.
7817	PostgreSQL doesn't support the `FOR XML PATH` option in SQL queries.

Action code	Action message
7920	PostgreSQL doesn't support `EXPLICIT` mode with `FOR XML`.
7924	PostgreSQL doesn't support XPath queries that return multiple elements.

Constraints



Constraints feature is almost fully automated and compatible between SQL Server and Aurora PostgreSQL. The differences are: missing `SET DEFAULT` and check constraint with sub-query.

For more information, see [SQL Server Constraints and PostgreSQL Table Constraints](#).

Action code	Action message
7606	PostgreSQL doesn't support foreign keys that reference partitioned tables.
7675	PostgreSQL doesn't support `ASC` and `DESC` sorting options for constraints.
7825	AWS SCT removed the default value of the `DateTime` column.
7915	Converted code might produce different results compared to the source code. Make sure that the constraint includes the %s column.

Linked Servers

Aurora PostgreSQL supports remote data access from the database. Connectivity between schemas is trivial, but connectivity to other instances require an extension installation.

For more information, see SQL Server Linked Servers and PostgreSQL DBLink and FDWrapper.

Action code	Action message
7645	PostgreSQL doesn’t support running pass-thro ugh commands on linked servers.

Synonyms

Aurora PostgreSQL supports synonyms. If synonyms refer to tables, views, or functions, you can replace them with views or functions to wrap those. It becomes more challenging when synonyms refer to other objects.

For more information, see SQL Server Synonyms and PostgreSQL Views, Types, and Functions.

Action code	Action message
7792	PostgreSQL doesn’t support synonyms.

AWS Database Migration Service

The AWS Database Migration Service (AWS DMS) helps you migrate databases to AWS quickly and securely. The source database remains fully operational during the migration, minimizing downtime to applications that rely on the database. The AWS Database Migration Service can migrate your data to and from most widely-used commercial and open-source databases.

The service supports homogenous migrations such as Oracle to Oracle as well as heterogeneous migrations between different database platforms such as Oracle to Amazon Aurora or Microsoft SQL Server to MySQL. You can also use AWS DMS to stream data to Amazon Redshift, Amazon DynamoDB, and Amazon S3 from any of the supported sources, which are Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, SAP ASE, SQL Server, IBM DB2 LUW, and MongoDB, enabling consolidation and easy analysis of data in a petabyte-scale data warehouse. The AWS Database Migration Service can also be used for continuous data replication with high availability.

For AWS DMS pricing, see Database Migration Service pricing.

For all supported sources for AWS DMS, see Sources for data migration.

For all supported targets for AWS DMS, see Targets for data migration.

Migration Tasks Performed by AWS DMS

In a traditional solution, you need to perform capacity analysis, procure hardware and software, install and administer systems, and test and debug the installation. AWS DMS automatically manages the deployment, management, and monitoring of all hardware and software needed for your migration. You can start your migration within minutes of starting the AWS DMS configuration process.

With AWS DMS, you can scale up (or scale down) your migration resources as needed to match your actual workload. For example, if you determine that you need additional storage, you can easily increase your allocated storage and restart your migration, usually within minutes. On the other hand, if you discover that you aren't using all of the resource capacity you configured, you can easily downsize to meet your actual workload.

AWS DMS uses a pay-as-you-go model. You only pay for AWS DMS resources while you use them as opposed to traditional licensing models with up-front purchase costs and ongoing maintenance charges.

AWS DMS automatically manages all of the infrastructure that supports your migration server including hardware and software, software patching, and error reporting.

AWS DMS provides automatic failover. If your primary replication server fails for any reason, a backup replication server can take over with little or no interruption of service.

AWS DMS can help you switch to a modern, perhaps more cost-effective database engine than the one you are running now. For example, AWS DMS can help you take advantage of the managed database services provided by Amazon RDS or Amazon Aurora. Or, it can help you move to the managed data warehouse service provided by Amazon Redshift, NoSQL platforms like Amazon DynamoDB, or low-cost storage platforms like Amazon S3. Conversely, if you want to migrate away from old infrastructure but continue to use the same database engine, AWS DMS also supports that process.

AWS DMS supports nearly all of today's most popular DBMS engines as data sources, including Oracle, Microsoft SQL Server, MySQL, MariaDB, PostgreSQL, Db2 LUW, SAP, MongoDB, and Amazon Aurora.

AWS DMS provides a broad coverage of available target engines including Oracle, Microsoft SQL Server, PostgreSQL, MySQL, Amazon Redshift, SAP ASE, Amazon S3, and Amazon DynamoDB.

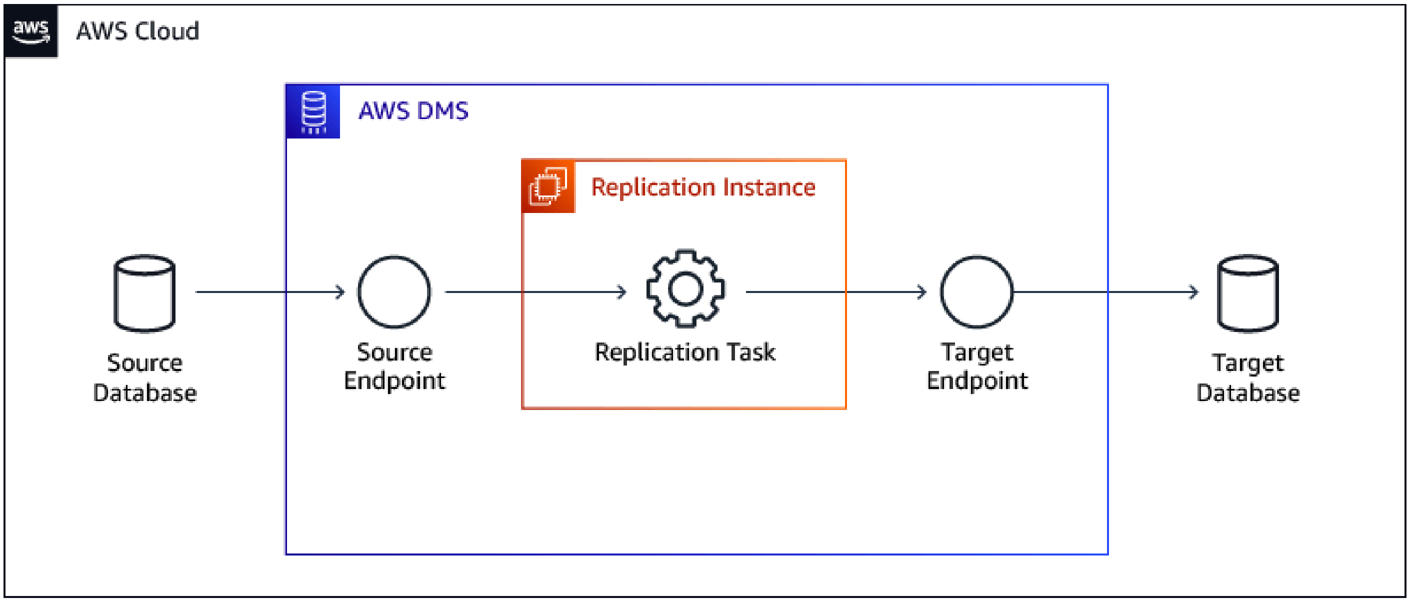
You can migrate from any of the supported data sources to any of the supported data targets. AWS DMS supports fully heterogeneous data migrations between the supported engines.

AWS DMS ensures that your data migration is secure. Data at rest is encrypted with AWS Key Management Service (AWS KMS) encryption. During migration, you can use Secure Socket Layers (SSL) to encrypt your in-flight data as it travels from source to target.

How AWS DMS Works

At its most basic level, AWS DMS is a server in the AWS Cloud that runs replication software. You create a source and target connection to tell AWS DMS where to extract from and load to. Then, you schedule a task that runs on this server to move your data. AWS DMS creates the tables and associated primary keys if they don't exist on the target. You can pre-create the target tables manually if you prefer. Or you can use AWS SCT to create some or all of the target tables, indexes, views, triggers, and so on.

The following diagram illustrates the AWS DMS process.



Latest Updates

AWS DMS is continuously evolving and supporting more and more options, find some of the latest updates following:

- Support for full-load with change data capture (CDC) and CDC-only tasks running against Oracle source tables created using the `CREATE TABLE AS` statement.
- New MySQL version AWS DMS now supports MySQL version 8.0 as a source except when the transaction payload is compressed.
- Support for AWS Secrets Manager integration. You can store the database connection details (user credentials) for supported endpoints securely in AWS Secrets Manager. You can then submit the corresponding secret instead of plain-text credentials to AWS DMS when you create or modify an endpoint. AWS DMS then connects to the endpoint databases using the secret. For more information, see Using secrets to access Database Migration Service endpoints.
- Support for Oracle extended data types for source and target.
- Support for TLS 1.2 for MySQL endpoints.
- Support for TLS 1.2 for SQL Server endpoints.

For a complete guide with a step-by-step walkthrough including all the latest notes for migrating SQL Server to Aurora MySQL with AWS DMS, see Migrating a SQL Server Database to Amazon Aurora MySQL.

For more information about AWS DMS, see What is Database Migration Service? and Best practices for Database Migration Service.

Amazon RDS on Outposts

Note

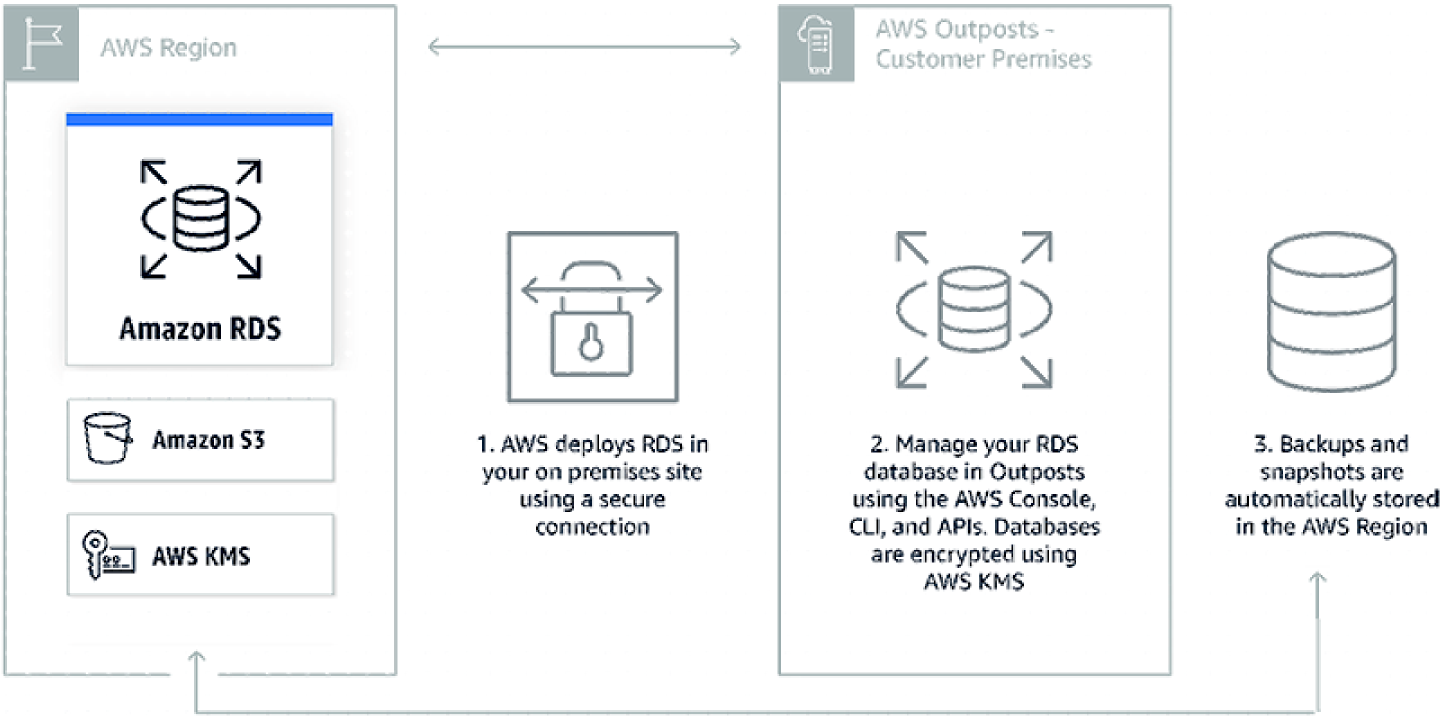
This topic is related to Amazon Relational Database Service (Amazon RDS) and isn't supported with Amazon Aurora.

Amazon RDS on Outposts is a fully managed service that offers the same AWS infrastructure, AWS services, APIs, and tools to virtually any data center, co-location space, or on-premises facility for a truly consistent hybrid experience. Amazon RDS on Outposts is ideal for workloads that require low latency access to on-premises systems, local data processing, data residency, and migration of applications with local system inter-dependencies.

When you deploy Amazon RDS on Outposts, you can run Amazon RDS on premises for low latency workloads that need to be run in close proximity to your on-premises data and applications. Amazon RDS on Outposts also enables automatic backup to an AWS Region. You can manage Amazon RDS databases both in the cloud and on premises using the same AWS Management Console, APIs, and CLI. Amazon RDS on Outposts supports Microsoft SQL Server, MySQL, and PostgreSQL database engines, with support for additional database engines coming soon.

How It Works

Amazon RDS on Outposts lets you run Amazon RDS in your on-premises or co-location site. You can deploy and scale an Amazon RDS database instance in Outposts just as you do in the cloud, using the AWS console, APIs, or CLI. Amazon RDS databases in Outposts are encrypted at rest using AWS KMS keys. Amazon RDS automatically stores all automatic backups and manual snapshots in the AWS Region.



This option is helpful when you need to run Amazon RDS on premises for low latency workloads that need to be run in close proximity to your on-premises data and applications.

For more information, see AWS Outposts Family, Amazon RDS on Outposts, and Create Amazon RDS DB Instances on Outposts.

Amazon RDS Proxy

Amazon RDS Proxy is a fully managed, highly available database proxy for Amazon Relational Database Service (RDS) that makes applications more scalable, more resilient to database failures, and more secure.

Many applications, including those built on modern server-less architectures, can have many open connections to the database server, and may open and close database connections at a high rate, exhausting database memory and compute resources. Amazon RDS Proxy allows applications to pool and share connections established with the database, improving database efficiency and application scalability. With Amazon RDS Proxy, fail-over times for Aurora and Amazon RDS databases are reduced by up to 66%. You can manage database credentials, authentication, and access through integration with AWS Secrets Manager and AWS Identity and Access Management (IAM).

You can turn on Amazon RDS Proxy for most applications with no code changes. You don't need to provision or manage any additional infrastructure. Pricing is simple and predictable: you pay for each vCPU of the database instance for which the proxy is enabled. Amazon RDS Proxy is now generally available for Aurora MySQL, Aurora PostgreSQL, Amazon RDS for MySQL, and Amazon RDS for PostgreSQL.

Amazon RDS Proxy Benefits

- **Improved application performance**. Amazon RDS proxy manages a connection pooling which helps with reducing the stress on database compute and memory resources that typically occurs when new connections are established and it is useful to efficiently support a large number and frequency of application connections.
- **Increase application availability**. By automatically connecting to a new database instance while preserving application connections Amazon RDS Proxy can reduce fail-over time by 66%.
- **Manage application security**. Amazon RDS Proxy also enables you to centrally manage database credentials using AWS Secrets Manager.
- **Fully managed**. Amazon RDS Proxy gives you the benefits of a database proxy without requiring additional burden of patching and managing your own proxy server.
- **Fully compatible with your database**. Amazon RDS Proxy is fully compatible with the protocols of supported database engines, so you can deploy Amazon RDS Proxy for your application without making changes to your application code.
- **Available and durable**. Amazon RDS Proxy is highly available and deployed over multiple Availability Zones (AZs) to protect you from infrastructure failure.

How Amazon RDS Proxy Works

For more information, see Amazon RDS Proxy for Scalable Serverless Applications and Amazon RDS Proxy.

Amazon Aurora Serverless v1

Amazon Aurora Serverless version 1 (v1) is an on-demand autoscaling configuration for Amazon Aurora. An Aurora Serverless DB cluster is a DB cluster that scales compute capacity up and down based on your application's needs. This contrasts with Aurora provisioned DB clusters, for which you manually manage capacity. Aurora Serverless v1 provides a relatively simple, cost-effective option for infrequent, intermittent, or unpredictable workloads. It is cost-effective because it automatically starts up, scales compute capacity to match your application's usage, and shuts down when it's not in use.

To learn more about pricing, see Serverless Pricing under MySQL-Compatible Edition or PostgreSQL-Compatible Edition on the Amazon Aurora pricing page.

Aurora Serverless v1 clusters have the same kind of high-capacity, distributed, and highly available storage volume that is used by provisioned DB clusters. The cluster volume for an Aurora Serverless v1 cluster is always encrypted. You can choose the encryption key, but you can't disable encryption. That means that you can perform the same operations on an Aurora Serverless v1 that you can on encrypted snapshots. For more information, see Aurora Serverless v1 and snapshots.

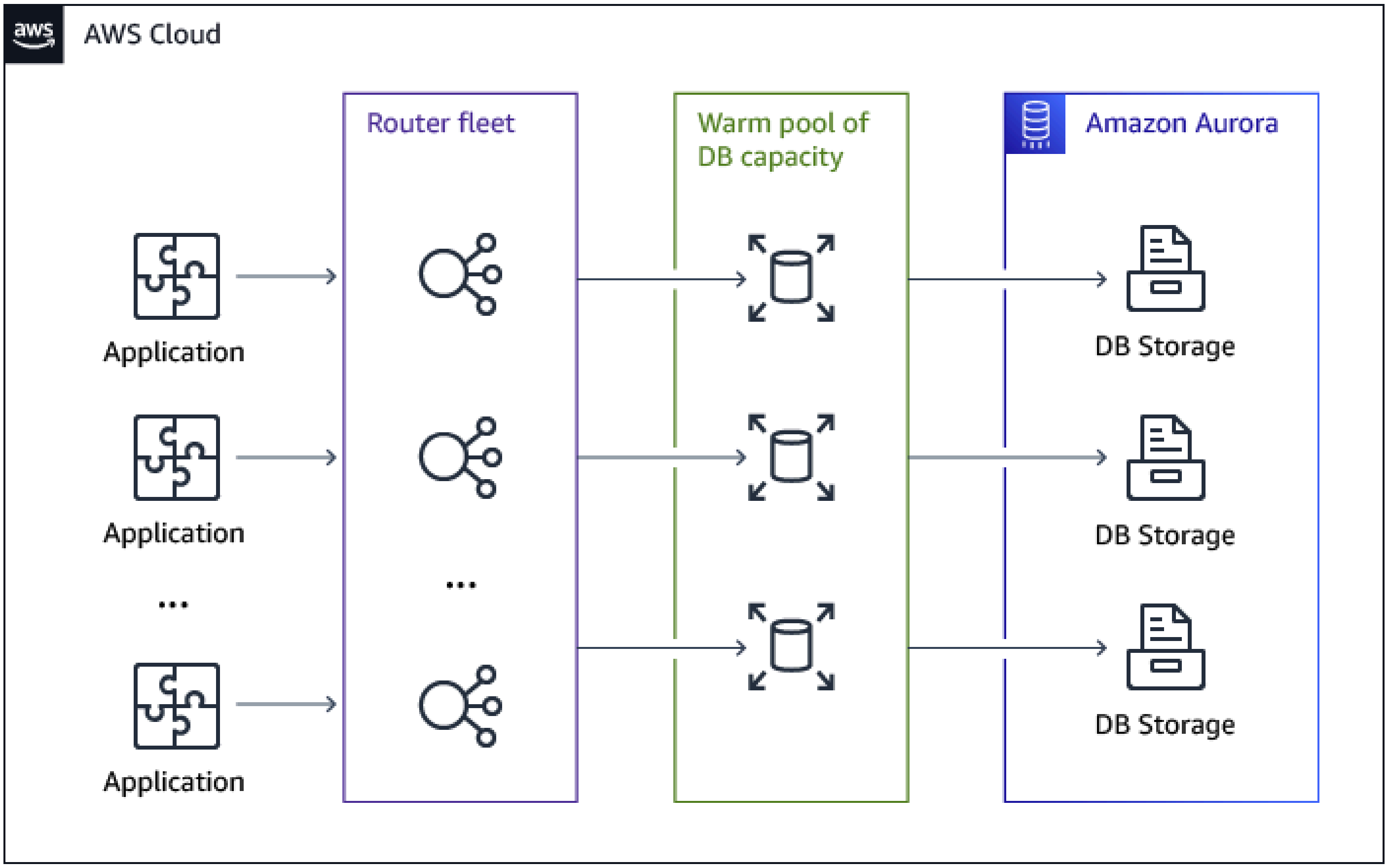
Aurora Serverless v1 provides the following advantages:

- **Simpler than provisioned**. Aurora Serverless v1 removes much of the complexity of managing DB instances and capacity.
- **Scalable**. Aurora Serverless v1 seamlessly scales compute and memory capacity as needed, with no disruption to client connections.
- **Cost-effective**. When you use Aurora Serverless v1, you pay only for the database resources that you consume, on a per-second basis.
- **Highly available storage**. Aurora Serverless v1 uses the same fault-tolerant, distributed storage system with six-way replication as Aurora to protect against data loss.

Aurora Serverless v1 is designed for the following use cases:

- **Infrequently used applications**. You have an application that is only used for a few minutes several times for each day or week, such as a low-volume blog site. With Aurora Serverless v1, you pay for only the database resources that you consume on a per-second basis.
- **New applications**. You're deploying a new application and you're unsure about the instance size you need. By using Aurora Serverless v1, you can create a database endpoint and have the database automatically scale to the capacity requirements of your application.
- **Variable workloads**. You're running a lightly used application, with peaks of 30 minutes to several hours a few times each day, or several times for each year. Examples are applications for human resources, budgeting, and operational reporting applications. With Aurora Serverless v1, you no longer need to provision for peak or average capacity.
- **Unpredictable workloads**. You're running daily workloads that have sudden and unpredictable increases in activity. An example is a traffic site that sees a surge of activity when it starts raining. With Aurora Serverless v1, your database automatically scales capacity to meet the needs of the application's peak load and scales back down when the surge of activity is over.
- **Development and test databases**. Your developers use databases during work hours but don't need them on nights or weekends. With Aurora Serverless v1, your database automatically shuts down when it's not in use.
- **Multi-tenant applications**. With Aurora Serverless v1, you don't have to individually manage database capacity for each application in your fleet. Aurora Serverless v1 manages individual database capacity for you.

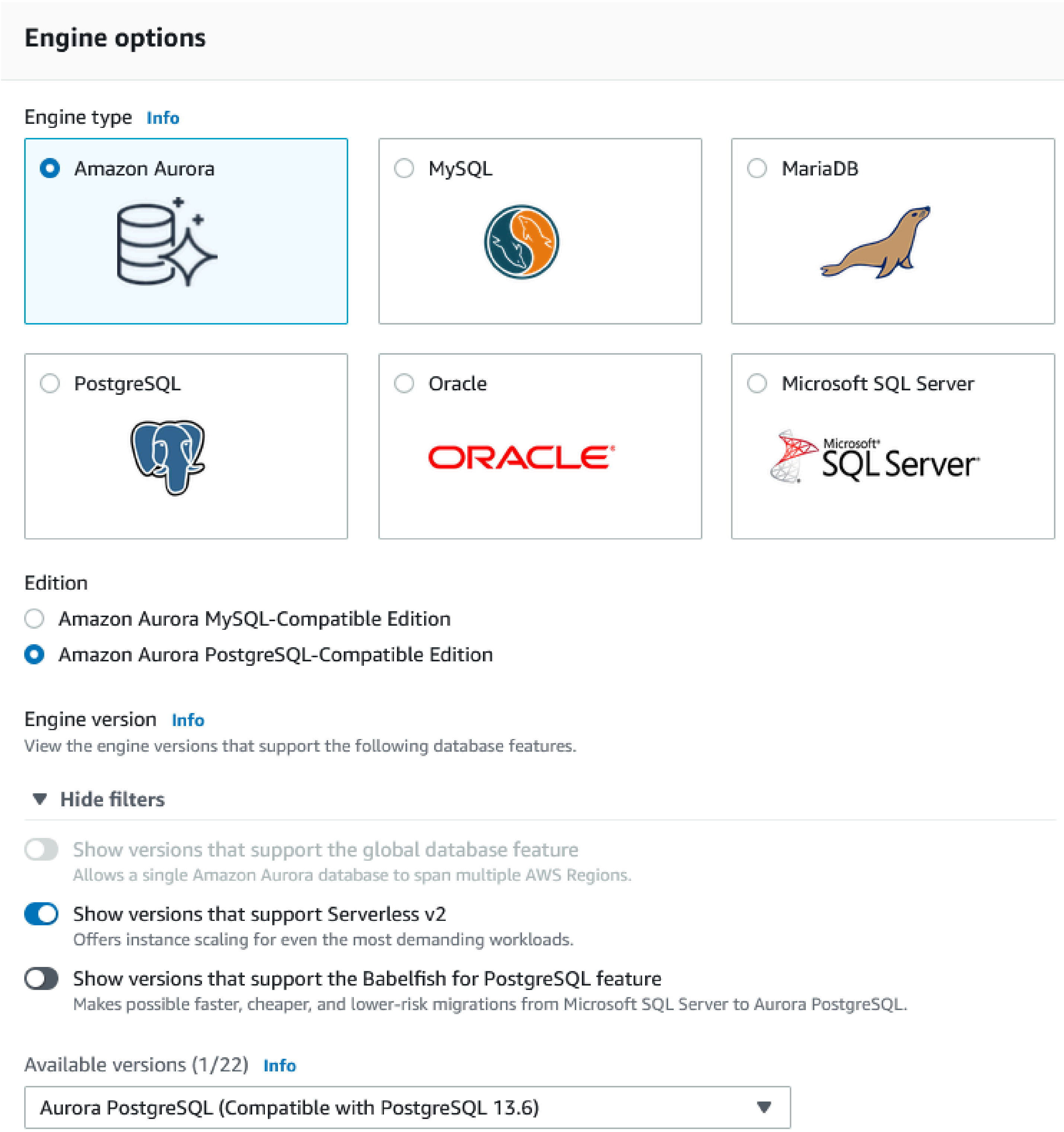
This process takes almost no time and since the storage is shared between nodes Aurora can scale up or down in seconds for most workloads. The service currently has autoscaling thresholds of 1.5 minutes to scale up and 5 minutes to scale down. That means metrics must exceed the limits for 1.5 minutes to trigger a scale up or fall below the limits for 5 minutes to trigger a scale down. The cool-down period between scaling activities is 5 minutes to scale up and 15 minutes to scale down. Before scaling can happen the service has to find a "scaling point" which may take longer than anticipated if you have long-running transactions. Scaling operations are transparent to the connected clients and applications since existing connections and session state are transferred to the new nodes. The only difference with pausing and resuming is a higher latency for the first connection, typically around 25 seconds. You can find more details in the documentation.



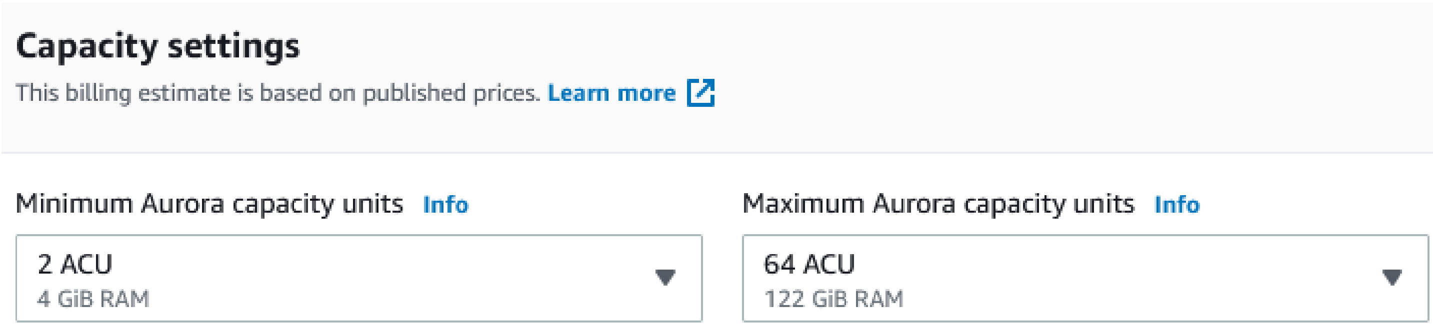
How to Provision

Log in to your Management Console, choose **Amazon RDS** , and then choose **Create database**.

On **Engine options**, for **Engine versions**, choose **Show versions that support Serverless v2**.



Choose the capacity settings for your use case.



For more information, see Amazon Aurora Serverless, Aurora Serverless MySQL Generally Available, and Amazon Aurora PostgreSQL Serverless Now Generally Available.

Amazon Aurora Backtrack

We've all been there, you need to make a quick, seemingly simple fix to an important production database. You compose the query, give it a once-over, and let it run. Seconds later you realize that you forgot the `WHERE` clause, dropped the wrong table, or made another serious mistake, and interrupt the query, but the damage has been done. You take a deep breath, whistle through your teeth, wish that reality came with an Undo option.

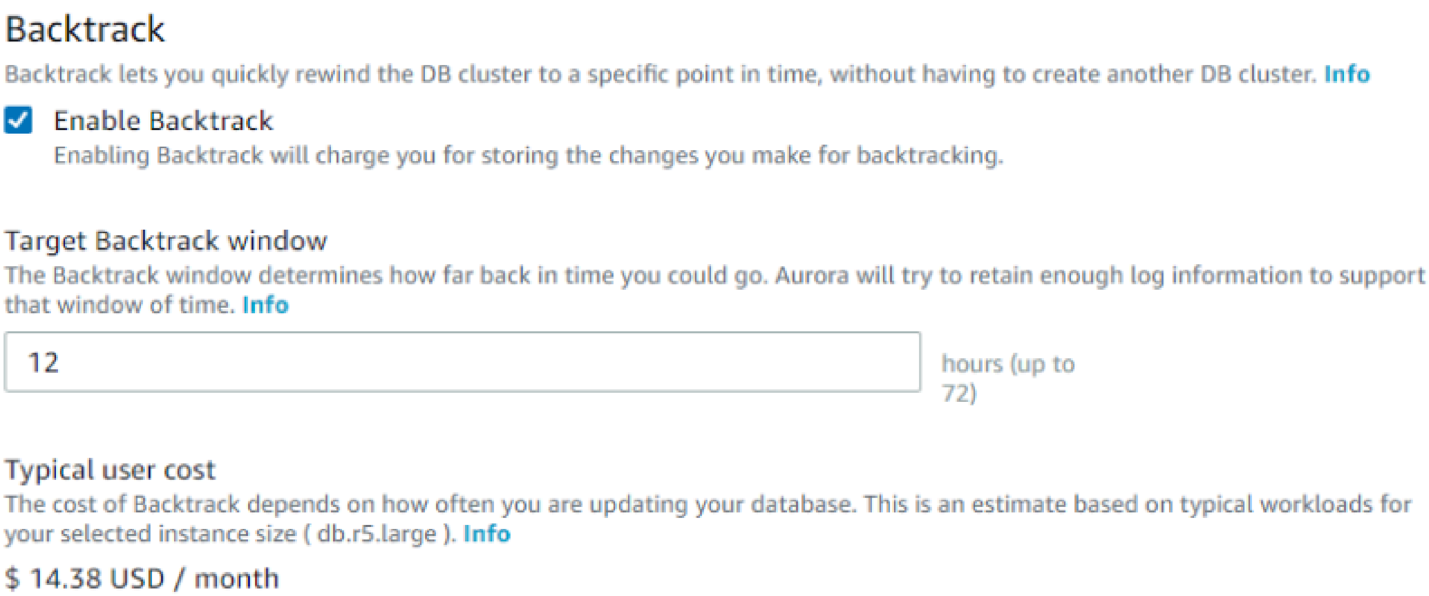
Backtracking rewinds the DB cluster to the time you specify. Backtracking isn't a replacement for backing up your DB cluster so that you can restore it to a point in time. However, backtracking provides the following advantages over traditional backup and restore:

- You can easily undo mistakes. If you mistakenly perform a destructive action, such as a `DELETE` without a `WHERE` clause, you can backtrack the DB cluster to a time before the destructive action with minimal interruption of service.
- You can backtrack a DB cluster quickly. Restoring a DB cluster to a point in time launches a new DB cluster and restores it from backup data or a DB cluster snapshot, which can take hours. Backtracking a DB cluster doesn't require a new DB cluster and rewinds the DB cluster in minutes.
- You can explore earlier data changes. You can repeatedly backtrack a DB cluster back and forth in time to help determine when a particular data change occurred. For example, you can backtrack a DB cluster three hours and then backtrack forward in time one hour. In this case, the backtrack time is two hours before the original time.

Amazon Aurora uses a distributed, log-structured storage system (read Design Considerations for High Throughput Cloud-Native Relational Databases to learn a lot more); each change to your database generates a new log record, identified by a Log Sequence Number (LSN). Enabling the backtrack feature provisions a FIFO buffer in the cluster for storage of LSNs. This allows for quick access and recovery times measured in seconds.

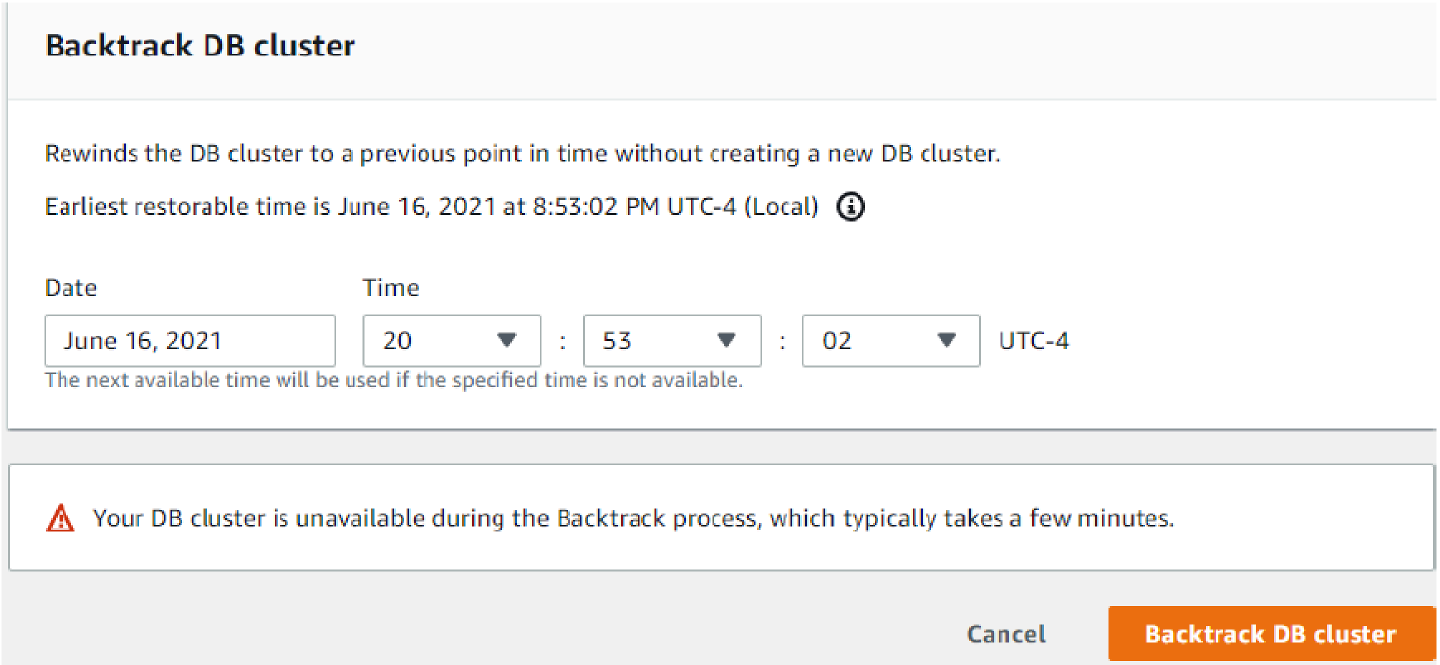
When you create a new Aurora MySQL DB cluster, backtracking is configured when you choose **Enable Backtrack** and specify a Target Backtrack window value that is greater than zero in the Backtrack section.

To create a DB cluster, follow the instructions in Creating an Amazon Aurora DB cluster. The following image shows the Backtrack section.



After a production error, you can simply pause your application, open up the Aurora Console, select the cluster, and choose **Backtrack DB cluster**.

Then you select Backtrack and choose the point in time just before your epic fail, and choose **Backtrack DB cluster**.



Then you wait for the rewind to take place, unpause your application and proceed as if nothing had happened. When you initiate a backtrack, Aurora will pause the database, close any open connections, drop uncommitted writes, and wait for the backtrack to complete. Then it will resume normal operation and be able to accept requests. The instance state will be backtracking while the rewind is underway.

Backtrack Window

With backtracking, there is a target backtrack window and an actual backtrack window:

- The target backtrack window is the amount of time you want to be able to backtrack your DB cluster. When you enable backtracking, you specify a target backtrack window. For example, you might specify a target backtrack window of 24 hours if you want to be able to backtrack the DB cluster one day.
- The actual backtrack window is the actual amount of time you can backtrack your DB cluster, which can be smaller than the target backtrack window. The actual backtrack window is based on your workload and the storage available for storing information about database changes, called change records.

As you make updates to your Aurora DB cluster with backtracking enabled, you generate change records. Aurora retains change records for the target backtrack window, and you pay an hourly rate for storing them. Both the target backtrack window and the workload on your DB cluster determine the number of change records you store. The workload is the number of changes you make to your DB cluster in a given amount of time. If your workload is heavy, you store more change records in your backtrack window than you do if your workload is light.

You can think of your target backtrack window as the goal for the maximum amount of time you want to be able to backtrack your DB cluster. In most cases, you can backtrack the maximum amount of time that you specified. However, in some cases, the DB cluster can't store enough change records to backtrack the maximum amount of time, and your actual backtrack window is smaller than your target. Typically, the actual backtrack window is smaller than the target when you have extremely heavy workload on your DB cluster. When your actual backtrack window is smaller than your target, we send you a notification.

When backtracking is enabled for a DB cluster, and you delete a table stored in the DB cluster, Aurora keeps that table in the backtrack change records. It does this so that you can revert back to a time before you deleted the table. If you don't have enough space in your backtrack window to store the table, the table might be removed from the backtrack change records eventually.

Backtracking Limitations

The following limitations apply to backtracking:

- Backtracking an Aurora DB cluster is available in certain AWS Regions and for specific Aurora MySQL versions only. For more information, see Backtracking in Aurora.
- Backtracking is only available for DB clusters that were created with the Backtrack feature enabled. You can enable the Backtrack feature when you create a new DB cluster or restore a snapshot of a DB cluster. For DB clusters that were created with the Backtrack feature enabled, you can create a clone DB cluster with the Backtrack feature enabled. Currently, you can't perform backtracking on DB clusters that were created with the Backtrack feature disabled.
- The limit for a backtrack window is 72 hours.
- Backtracking affects the entire DB cluster. For example, you can't selectively backtrack a single table or a single data update.
- Backtracking isn't supported with binary log (binlog) replication. Cross-Region replication must be disabled before you can configure or use backtracking.
- You can't backtrack a database clone to a time before that database clone was created. However, you can use the original database to backtrack to a time before the clone was created. For more information about database cloning, see Cloning an Aurora DB cluster volume.

- Backtracking causes a brief DB instance disruption. You must stop or pause your applications before starting a backtrack operation to ensure that there are no new read or write requests. During the backtrack operation, Aurora pauses the database, closes any open connections, and drops any uncommitted reads and writes. It then waits for the backtrack operation to complete.
- Backtracking isn't supported for the following AWS Regions:
  - Africa (Cape Town)
  - China (Ningxia)
  - Asia Pacific (Hong Kong)
  - Europe (Milan)
  - Europe (Stockholm)
  - Middle East (Bahrain)
  - South America (São Paulo)
- You can't restore a cross-Region snapshot of a backtrack-enabled cluster in an AWS Region that doesn't support backtracking.
- You can't use Backtrack with Aurora multi-master clusters.
- If you perform an in-place upgrade for a backtrack-enabled cluster from Aurora MySQL version 1 to version 2, you can't backtrack to a point in time before the upgrade happened.

For more information, see: Amazon Aurora Backtrack — Turn Back Time.

Migration Quick Tips

For more information, see Migration Quick Tips.

ANSI SQL

Topics

- Case Sensitivity Differences for SQL Server and PostgreSQL
- SQL Server Constraints and PostgreSQL Table Constraints
- Creating Tables
- Common Table Expressions
- Data Types
- Derived Tables
- GROUP BY
- Table JOIN
- SQL Server Temporal Tables and PostgreSQL Triggers
- Views
- Window Functions

Case Sensitivity Differences for SQL Server and PostgreSQL

Object name case sensitivity might be different for SQL Server and PostgreSQL. By default, SQL Server names are case insensitive. However, you can create a case sensitive SQL Server database by changing the `COLLATION` property. In PostgreSQL, object names are case insensitive.

By default, the AWS Schema Conversion Tool (AWS SCT) uses object names in lowercase for PostgreSQL. If your source code includes objects with identical names in different case, make sure that you keep unique names in your converted code. You can enclose object names in double quotation marks or change the names manually.

In addition to this, you can use AWS Database Migration Service transformation actions to change schema, table, and column names to lowercase. For more information, see Transformation rules and actions.

To use an uppercase name, enclose object names with double quotation marks. The following code example shows how to create the `EMPLOYEES` table in uppercase.

```
CREATE TABLE "EMPLOYEES" (
  EMP_ID NUMERIC PRIMARY KEY,
  EMP_FULL_NAME VARCHAR(60) NOT NULL,
  AVG_SALARY NUMERIC NOT NULL);
```

The following PostgreSQL command creates the `employees` table in lowercase.

```
CREATE TABLE EMPLOYEES (
  EMP_ID NUMERIC PRIMARY KEY,
  EMP_FULL_NAME VARCHAR(60) NOT NULL,
  AVG_SALARY NUMERIC NOT NULL);
```

If you don’t use double quotation marks, then PostgreSQL creates objects with lowercase names. To create, query, or manage PostgreSQL database objects with names in uppercase or mixed case, use double quotation marks.

SQL Server Constraints and PostgreSQL Table Constraints

Feature compatibi lity	**AWS SCT / AWS DMS automation level**	**AWS SCT action code index**	**Key differences**
		Constraints	The `SET DEFAULT` option is missing. Check constraint with subquery.

SQL Server Usage

Column and table constraints are defined by the SQL standard and enforce relational data consistency. You can use four types of SQL constraints: check, unique, primary key, and foreign key.

Check Constraints

Check constraints enforce domain integrity by limiting the data values stored in table columns. They are logical Boolean expressions that evaluate to one of the following three values: `TRUE`, `FALSE`, and `UNKNOWN`.

```
CHECK (<Logical Expression>)
```

> **Note**
>
> Check constraint expressions behave differently than predicates in other query clauses. For example, in a `WHERE` clause, a logical expression that evaluates to `UNKNOWN` is functionally equivalent to `FALSE` and the row is filtered out. For check constraints, an expression that evaluates to `UNKNOWN` is functionally equivalent to `TRUE` because the value is permitted by the constraint.

You can assign multiple check constraints to a column. Also, you can apply a single check constraint to multiple columns. In this case, it works as a table-level check constraint.

In ANSI SQL, check constraints can't access other rows as part of the expression. In SQL Server, you can use user-defined functions in constraints to access other rows, tables, or databases.

Unique Constraints

You can use unique constraints for all candidate keys. A candidate key is an attribute or a set of attributes or columns that uniquely identify each row in the relation (table data).

```
UNIQUE [CLUSTERED | NONCLUSTERED] (<Column List>)
```

Unique constraints guarantee that no rows with duplicate column values exist in a table.

A unique constraint can be simple or composite. Simple constraints are composed of a single column. Composite constraints are composed of multiple columns. A column may be a part of more than one constraint.

According to the ANSI SQL standard, you can have multiple rows with `NULL` values for unique constraints. However, in SQL Server, you can use a `NULL` value only for a single row. You can use a `NOT NULL` constraint in addition to a unique constraint to address this limitation.

To improve the efficiency, SQL Server creates a unique index to support unique constraints. Otherwise, every `INSERT` and `UPDATE` would require a full table scan to verify that the table doesn't include duplicates. The default index type for unique constraints is non-clustered.

Primary Key Constraints

A primary key is a candidate key serving as the unique identifier of a table row. Primary keys might consist of one or more columns. All columns that comprise a primary key must also have a NOT NULL constraint. Tables can have one primary key.

```
PRIMARY KEY [CLUSTERED | NONCLUSTERED] (<Column List>)
```

The default index type for primary keys is a clustered index.

Foreign Key Constraints

Foreign key constraints enforce domain referential integrity. Similar to check constraints, foreign keys limit the values stored in a column or set of columns.

```
FOREIGN KEY (<Referencing Column List>)
REFERENCES <Referenced Table>(<Referenced Column List>)
```

Foreign keys reference columns in other tables, which must be either primary keys or have unique constraints. The set of values that you can use for the referencing table is the set of values that exist in the referenced table.

Although the columns referenced in the parent table are indexed because they have either a primary key or unique constraint, no indexes are automatically created for the referencing columns in the child table. A best practice is to create appropriate indexes to support joins and constraint enforcement.

Foreign key constraints impose DML limitations for the referencing child and parent tables. The purpose of a constraint is to guarantee that no orphan rows, which don't have corresponding matching values in the parent table exist in the referencing table. The constraint limits INSERT and UPDATE to the child table and UPDATE and DELETE to the parent table. For example, you can't delete an order having associated order items.

Foreign keys support Cascading Referential Integrity (CRI). You can use CRI to enforce constraints and define action paths for DML statements that violate the constraints. There are four CRI options:

- **NO ACTION**. When the constraint is violated due to a DML operation, an error is raised and the operation is rolled back.

- **CASCADE**. Values in a child table are updated with values from the parent table when they are updated or deleted along with the parent.
- **SET NULL**. All columns that are part of the foreign key are set to NULL when the parent is deleted or updated.
- **SET DEFAULT**. All columns that are part of the foreign key are set to their DEFAULT value when the parent is deleted or updated.

You can customize these actions independently of others in the same constraint. For example, a cascading constraint may have `CASCADE` for `UPDATE`, but `NO ACTION` for `DELETE`.

Examples

Create a composite non-clustered primary key.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL,
Col2 INT NOT NULL,
Col3 VARCHAR(20) NULL,
CONSTRAINT PK_MyTable
PRIMARY KEY NONCLUSTERED (Col1, Col2)
);
```

Create a table-level check constraint.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL,
Col2 INT NOT NULL,
Col3 VARCHAR(20) NULL,
CONSTRAINT PK_MyTable
PRIMARY KEY NONCLUSTERED (Col1, Col2),
CONSTRAINT CK_MyTableCol1Col2
CHECK (Col2 >= Col1)
);
```

Create a simple non-null unique constraint.

```
CREATE TABLE MyTable
```

```
(
Col1 INT NOT NULL,
Col2 INT NOT NULL,
Col3 VARCHAR(20) NULL,
CONSTRAINT PK_MyTable
PRIMARY KEY NONCLUSTERED (Col1, Col2),
CONSTRAINT UQ_Col2Col3
UNIQUE (Col2, Col3)
);
```

Create a foreign key with multiple cascade actions.

```
CREATE TABLE MyParentTable
(
Col1 INT NOT NULL,
Col2 INT NOT NULL,
Col3 VARCHAR(20) NULL,
CONSTRAINT PK_MyTable
PRIMARY KEY NONCLUSTERED (Col1, Col2)
);
```

```
CREATE TABLE MyChildTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 INT NOT NULL,
Col3 INT NOT NULL,
CONSTRAINT FK_MyChildTable_MyParentTable
FOREIGN KEY (Col2, Col3)
REFERENCES MyParentTable (Col1, Col2)
ON DELETE NO ACTION
ON UPDATE CASCADE
);
```

For more information, see Unique Constraints and Check Constraints and Primary and Foreign Key Constraints in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL supports the following types of table constraints:

- `PRIMARY KEY`.

- `FOREIGN KEY`.
- `UNIQUE`.
- `NOT NULL`.
- `EXCLUDE` (unique to PostgreSQL).

Similar to constraint declaration in SQL Server, you can create constraints inline or out-of-line when you specify table columns in PostgreSQL.

You can specify PostgreSQL constraints using `CREATE TABLE` or `ALTER TABLE`. Constraints on views aren't supported.

Make sure that you have the `CREATE` and `ALTER` privileges on the table for which you create constraints. For foreign key constraints, make sure that you have the `REFERENCES` privilege.

Primary Key Constraints

- Uniquely identify each row and can't contain NULL values.
- Use the same ANSI SQL syntax as SQL Server.
- You can create primary key constraints on a single column or on multiple columns (composite primary keys) as the only primary key in a table.
- Creating a primary key constraint automatically creates a unique B-Tree index on the column or group of columns marked as the primary key of the table.
- You can generate constraint names automatically by PostgreSQL or explicitly specified during constraint creation.

Create an inline primary key constraint with a system-generated constraint name.

```
CREATE TABLE EMPLOYEES (
  EMPLOYEE_ID NUMERIC PRIMARY KEY,
  FIRST_NAME VARCHAR(20),
  LAST_NAME VARCHAR(25),
  EMAIL VARCHAR(25));
```

Create an inline primary key constraint with a user-specified constraint name.

```
CREATE TABLE EMPLOYEES (
```

```
  EMPLOYEE_ID NUMERIC CONSTRAINT PK_EMP_ID PRIMARY KEY,
  FIRST_NAME VARCHAR(20),
  LAST_NAME VARCHAR(25),
  EMAIL VARCHAR(25));
```

Create an out-of-line primary key constraint.

```
CREATE
```

```
CREATE TABLE EMPLOYEES(
  EMPLOYEE_ID NUMERIC,
  FIRST_NAME VARCHAR(20),
  LAST_NAME VARCHAR(25),
  EMAIL VARCHAR(25)),
  CONSTRAINT PK_EMP_ID PRIMARY KEY (EMPLOYEE_ID));
```

Add a primary key constraint to an existing table.

```
ALTER TABLE SYSTEM_EVENTS
  ADD CONSTRAINT PK_EMP_ID PRIMARY KEY (EVENT_CODE, EVENT_TIME);
```

Drop the primary key.

```
ALTER TABLE SYSTEM_EVENTS DROP CONSTRAINT PK_EMP_ID;
```

Foreign Key Constraints

- Enforce referential integrity in the database. Values in specific columns or a group of columns must match the values from another table or column.
- Creating a foreign key constraint in PostgreSQL uses the same ANSI SQL syntax as SQL Server.
- You can create foreign key constraints in-line or out-of-line during table creation.
- Use the `REFERENCES` clause to specify the table referenced by the foreign key constraint.
- When specifying `REFERENCES` in the absence of a column list in the referenced table, the primary key of the referenced table is used as the referenced column or columns.
- A table can have multiple foreign key constraints.
- Use the ON DELETE clause to handle foreign key parent record deletions such as cascading deletes.

- Foreign key constraint names are generated automatically by the database or specified explicitly during constraint creation.

ON DELETE Clause

PostgreSQL provides three main options to handle cases where data is deleted from the parent table and a child table is referenced by a FOREIGN KEY constraint. By default, without specifying any additional options, PostgreSQL uses the NO ACTION method and raises an error if the referencing rows still exist when the constraint is verified.

- `ON DELETE CASCADE`. Any dependent foreign key values in the child table are removed along with the referenced values from the parent table.
- `ON DELETE RESTRICT`. Prevents the deletion of referenced values from the parent table and the deletion of dependent foreign key values in the child table.
- `ON DELETE NO ACTION`. Performs no action (the default). The fundamental difference between `RESTRICT` and `NO ACTION` is that `NO ACTION` allows the check to be postponed until later in the transaction; `RESTRICT` doesn't.

ON UPDATE Clause

Handling updates on `FOREIGN KEY` columns is also available using the `ON UPDATE` clause, which shares the same options as the `ON DELETE` clause:

- `ON UPDATE CASCADE`.
- `ON UPDATE RESTRICT`.
- `ON UPDATE NO ACTION`.

Create an inline foreign key with a user-specified constraint name.

```
CREATE TABLE EMPLOYEES (
  EMPLOYEE_ID NUMERIC PRIMARY KEY,
  FIRST_NAME VARCHAR(20),
  LAST_NAME VARCHAR(25),
  EMAIL VARCHAR(25),
  DEPARTMENT_ID NUMERIC REFERENCES DEPARTMENTS(DEPARTMENT_ID));
```

Create an out-of-line foreign key constraint with a system-generated constraint name.

```
CREATE TABLE EMPLOYEES (
  EMPLOYEE_ID NUMERIC PRIMARY KEY,
  FIRST_NAME VARCHAR(20),
  LAST_NAME VARCHAR(25),
  EMAIL VARCHAR(25),
  DEPARTMENT_ID NUMERIC,
  CONSTRAINT FK_FEP_ID
  FOREIGN KEY(DEPARTMENT_ID) REFERENCES DEPARTMENTS(DEPARTMENT_ID));
```

Create a foreign key using the `ON DELETE CASCADE` clause.

```
CREATE TABLE EMPLOYEES (
  EMPLOYEE_ID NUMERIC PRIMARY KEY,
  FIRST_NAME VARCHAR(20),
  LAST_NAME VARCHAR(25),
  EMAIL VARCHAR(25),
  DEPARTMENT_ID NUMERIC,
  CONSTRAINT FK_FEP_ID
  FOREIGN KEY(DEPARTMENT_ID) REFERENCES DEPARTMENTS(DEPARTMENT_ID)
  ON DELETE CASCADE);
```

Add a foreign key to an existing table.

```
ALTER TABLE EMPLOYEES ADD CONSTRAINT FK_DEPT
  FOREIGN KEY (department_id)
  REFERENCES DEPARTMENTS (department_id) NOT VALID;

ALTER TABLE EMPLOYEES VALIDATE CONSTRAINT FK_DEPT;
```

ON UPDATE Clause

- Ensure that values in a column, or a group of columns, are unique across the entire table.
- PostgreSQL unique constraint syntax is ANSI SQL compatible.
- Automatically creates a B-Tree index on the respective column, or a group of columns, when creating a `UNIQUE` constraint.
- If duplicate values exist in the column, for which you create the unique constraint, the operation fails and returns an error message.
- Unique constraints in PostgreSQL accept multiple NULL values. This behavior is similar to SQL Server.

- You can use system-generated or explicitly specified naming for unique constraints.

Create an inline unique constraint ensuring uniqueness of values in the email column.

```
CREATE TABLE EMPLOYEES (
  EMPLOYEE_ID NUMERIC PRIMARY KEY,
  FIRST_NAME VARCHAR(20),
  LAST_NAME VARCHAR(25),
  EMAIL VARCHAR(25) CONSTRAINT UNIQ_EMP_EMAIL UNIQUE,
  DEPARTMENT_ID NUMERIC);
```

CHECK Constraints

- Enforce that values in a column satisfy a specific requirement.
- Check constraints in PostgreSQL use the same ANSI SQL syntax as SQL Server.
- Can only be defined using a Boolean data type to evaluate the values of a column.
- Check constraints naming can be system-generated or explicitly specified by the user during constraint creation.

Check constraints are using Boolean data type, therefore you can't use subqueries in the check constraint. To use this feature, you can create a Boolean function that will check the query results and return `TRUE` or `FALSE` values accordingly.

NOT NULL Constraints

- Enforce that a column can't accept NULL values. This behavior is different from the default column behavior in PostgreSQL where columns can accept NULL values.
- `NOT NULL` constraints can only be defined inline during table creation.
- You can explicitly specify names for `NOT NULL` constraints when used with a `CHECK` constraint.

Define two not null constraints on the `FIRST_NAME` and `LAST_NAME` columns. Define a check constraint with an explicitly user-specified name to enforce not null behavior on the `EMAIL` column.

```
CREATE TABLE EMPLOYEES (
  EMPLOYEE_ID NUMERIC PRIMARY KEY,
```

```
  FIRST_NAME VARCHAR(20) NOT NULL,
  LAST_NAME VARCHAR(25) NOT NULL,
  EMAIL VARCHAR(25) CONSTRAINT CHK_EMAIL
    CHECK(EMAIL IS NOT NULL));
```

SET Constraints Syntax

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

PostgreSQL provides controls for certain aspects of constraint behavior:

- `DEFERRABLE` | `NOT DEFERRABLE`. Using the PostgreSQL `SET CONSTRAINTS` statement. You can define constraints as:
  - `DEFERRABLE`. Allows you to use the `SET CONSTRAINTS` statement to set the behavior of constraint checking within the current transaction until transaction commit.
  - `IMMEDIATE`. Constraints are enforced only at the end of each statement. Note that each constraint has its own `IMMEDIATE` or `DEFERRED` mode.
  - `NOT DEFERRABLE`: This statement always runs as `IMMEDIATE` and isn't affected by the `SET CONSTRAINTS` command.
- `VALIDATE CONSTRAINT` | `NOT VALID`.
  - `VALIDATE CONSTRAINT`. Validates foreign key or check constraints only that were previously created as `NOT VALID`. This action performs a validation check by scanning the table to ensure all records satisfy the constraint definition.
  - `NOT VALID`. You can use this type only for foreign key or check constraints. When specified, new records aren't validated with the creation of the constraint. Only when the `VALIDATE CONSTRAINT` state is applied is the constraint state enforced on all records.

Using Existing Indexes During Constraint Creation

PostgreSQL can add a new primary key or unique constraints based on an existing unique index. PostgreSQL includes all index columns in the constraint. When you create constraints using this method, the index is owned by the constraint. If you delete the constraint, then PostgreSQL deletes the index.

Use an existing unique index to create a primary key constraint.

```
CREATE UNIQUE INDEX IDX_EMP_ID ON EMPLOYEES(EMPLOYEE_ID);
```

```
ALTER TABLE EMPLOYEES
  ADD CONSTRAINT PK_CON_UNIQ PRIMARY KEY USING INDEX IDX_EMP_ID;
```

Summary

The following table identifies similarities, differences, and key migration considerations.

Feature	SQL Server	Aurora PostgreSQL
Check constraints	CHECK	CHECK
Unique constraints	UNIQUE	UNIQUE
Primary key constraints	PRIMARY KEY	PRIMARY KEY
Foreign key constraints	FOREIGN KEY	FOREIGN KEY
Cascaded referential actions	NO ACTION, CASCADE, SET NULL, SET DEFAULT	RESTRICT, CASCADE, SET NULL, NO ACTION
Indexing of referencing columns	Not required	N/A
Indexing of referenced columns	PRIMARY KEY or UNIQUE	PRIMARY KEY or UNIQUE

For more information, see Constraints, SET CONSTRAINTS, and ALTER TABLE in the *PostgreSQL documentation*.

Creating Tables

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Creating Tables	Auto generated value column is different

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
			. Can't use physical attribute ON. Missing table variable and memory-optimized table.

SQL Server Usage

ANSI Syntax Conformity

You can create tables in SQL Server using the `CREATE TABLE` statement and conform to the ANSI/ISO entry level standard. The basic features of `CREATE TABLE` are similar for most relational database management engines and are well defined in the ANSI/ISO standards.

In its most basic form, the `CREATE TABLE` statement in SQL Server is used to define:

- Table names, the containing security schema, and database.
- Column names.
- Column data types.
- Column and table constraints.
- Column default values.
- Primary, candidate (UNIQUE), and foreign keys.

T-SQL Extensions

SQL Server extends the basic syntax and provides many additional options for the `CREATE TABLE` or `ALTER TABLE` statements. The most often used options are:

- Supporting index types for primary keys and unique constraints, clustered or non-clustered, and index properties such as `FILLFACTOR`.
- Physical table data storage containers using the `ON <File Group>` clause.
- Defining `IDENTITY` auto-enumerator columns.

- Encryption.
- Compression.
- Indexes.

For more information, see Data Types, Column Encryption, and Databases and Schemas.

Table Scope

SQL Server provides five scopes for tables

- Standard tables are created on disk, globally visible, and persist through connection resets and server restarts.
- Temporary tables are designated with the "# " prefix. They are persisted in TempDB and are visible to the run scope and any sub-scopes where they were created. Temporary tables are cleaned up by the server when the run scope terminates and when the server restarts.
- Global temporary tables are designated by the "## " prefix. They are similar in scope to temporary tables, but are also visible to concurrent scopes.
- Table variables are defined with the DECLARE statement, not with CREATE TABLE. They are visible only to the run scope where they were created.
- Memory-optimized tables are special types of tables used by the In-Memory Online Transaction Processing (OLTP) engine. They use a non-standard CREATE TABLE syntax.

Creating a Table Based on an Existing Table or Query

In SQL Server, you can create new tables based on SELECT queries as an alternate to the CREATE TABLE statement. You can use a SELECT statement that returns a valid set with unique column names to create a new table and populate data.

SELECT INTO is a combination of DML and DDL. The simplified syntax for SELECT INTO is shown following.

```
SELECT <Expression List>
INTO <Table Name>
[FROM <Table Source>]
[WHERE <Filter>]
[GROUP BY <Grouping Expressions>...];
```

When you create a new table using `SELECT INTO`, the only attributes created for the new table are column names, column order, and the data types of the expressions. Even a straight forward statement such as` SELECT * INTO <New Table> FROM <Source Table>` doesn't copy constraints, keys, indexes, identity property, default values, or any other related objects.

TIMESTAMP Syntax for ROWVERSION Deprecated Syntax

The `TIMESTAMP` syntax synonym for `ROWVERSION` has been deprecated as of SQL Server 2008R2 in accordance with Deprecated Database Engine Features in SQL Server 2008 R2.

Previously, you could use either the `TIMESTAMP` or the `ROWVERSION` keywords to denote a special data type that exposes an auto-enumerator. The auto-enumerator generates unique eight-byte binary numbers typically used to version-stamp table rows. Clients read the row, process it, and check the `ROWVERSION` value against the current row in the table before modifying it. If they are different, the row has been modified since the client read it. The client can then apply different processing logic.

Note that when migrating to Aurora PostgreSQL using the AWS Schema Conversion Tool, neither `ROWVERSION` nor `TIMESTAMP` are supported. You must add customer logic, potentially in the form of a trigger, to maintain this functionality.

Syntax

Simplified syntax for `CREATE TABLE` is shown following.

```
CREATE TABLE [<Database Name>.<Schema Name>].<Table Name> (<Column Definitions>)
  [ON{<Partition Scheme Name> (<Partition Column Name>)];
```

```
<Column Definition>:
<Column Name> <Data Type>
[CONSTRAINT <Column Constraint>
[DEFAULT <Default Value>]]
[IDENTITY [(<Seed Value>, <Increment Value>)]
[NULL | NOT NULL]
[ENCRYPTED WITH (<Encryption Specifications>)
[<Column Constraints>]
[<Column Index Specifications>]
```

```
<Column Constraint>:
```

```
[CONSTRAINT <Constraint Name>]
{{PRIMARY KEY | UNIQUE} [CLUSTERED | NONCLUSTERED]
[WITH FILLFACTOR = <Fill Factor>]
| [FOREIGN KEY]
REFERENCES <Referenced Table> (<Referenced Columns>)]
```

```
<Column Index Specifications>:
INDEX <Index Name> [CLUSTERED | NONCLUSTERED]
[WITH(<Index Options>]
```

Examples

Create a basic table.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
);
```

Create a table with column constraints and an identity.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL PRIMARY KEY IDENTITY (1,1),
Col2 VARCHAR(20) NOT NULL CHECK (Col2 <> ''),
Col3 VARCHAR(100) NULL
REFERENCES MyOtherTable (Col3)
);
```

Create a table with an additional index.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
INDEX IDX_Col2 NONCLUSTERED
);
```

For more information, see CREATE TABLE (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

As SQL Server, Aurora PostgreSQL provides ANSI/ISO syntax entry level conformity for CREATE TABLE and custom extensions to support Aurora PostgreSQL specific functionality.

In its most basic form, and very similar to SQL Server, the CREATE TABLE statement in Aurora PostgreSQL is used to define:

- Table names containing security schema and/or database.
- Column names.
- Column data types.
- Column and table constraints.
- Column default values.
- Primary, candidate (UNIQUE), and foreign keys.

Starting with PostgreSQL 12 support for generated columns has been added. Generated columns can be either calculated from other columns values on the fly or calculated and stored.

```
CREATE TABLE tst_gen(
n NUMERIC,
n_gen GENERATED ALWAYS AS (n*0.01)
);
```

Aurora PostgreSQL Extensions

Aurora PostgreSQL extends the basic syntax and allows many additional options to be defined as part of the CREATE TABLE or ALTER TABLE statements. The most often used option is in-line index definition.

Table Scope

Aurora PostgreSQL provides two table scopes:

- Standard tables are created on disk, visible globally, and persist through connection resets and server restarts.
- Temporary tables are created using the CREATE GLOBAL TEMPORARY TABLE statement. A TEMPORARY table is visible only to the session that creates it and is dropped automatically when the session is closed.

Creating a Table Based on an Existing Table or Query

Aurora PostgreSQL provides two ways to create standard or temporary tables based on existing tables and queries: `CREATE TABLE <New Table> LIKE <Source Table>` and `CREATE TABLE … AS <Query Expression> .`

`CREATE TABLE <New Table> LIKE <Source Table>` creates an empty table based on the definition of another table including any column attributes and indexes defined in the original table.

`CREATE TABLE … AS <Query Expression>` is very similar to `SELECT INTO` in SQL Server. You can use this query to create a new table and populate data in a single step.

The following code example creates a new empty table based on the definition of the SourceTable table.

```
CREATE TABLE SourceTable(Col1 INT);

INSERT INTO SourceTable VALUES (1)

CREATE TABLE NewTable(Col1 INT) AS SELECT Col1 AS Col2 FROM SourceTable;

INSERT INTO NewTable (Col1, Col2) VALUES (2,3);

SELECT * FROM NewTable

Col1  Col2
NULL  1
2     3
```

Converting TIMESTAMP and ROWVERSION Columns

The following code example shows how you can use SQL server to provide an automatic mechanism for stamping row versions for application concurrency control.

```
CREATE TABLE WorkItems
(
WorkItemID INT IDENTITY(1,1) PRIMARY KEY,
WorkItemDescription XML NOT NULL,
Status VARCHAR(10) NOT NULL DEFAULT ('Pending'),
-- other columns...
VersionNumber ROWVERSION
```

```
);
```

The VersionNumber column automatically updates when a row is modified. The actual value is meaningless. Just the fact that it changed is what indicates a row modification. The client can now read a work item row, process it, and ensure no other clients updated the row before updating the status.

```
SELECT @WorkItemDescription = WorkItemDescription,
@Status = Status,
@VersionNumber = VersionNumber
FROM WorkItems
WHERE WorkItemID = @WorkItemID;

EXECUTE ProcessWorkItem @WorkItemID, @WorkItemDescription, @Status OUTPUT;

IF (
  SELECT VersionNumber
  FROM WorkItems
  WHERE WorkItemID = @WorkItemID
  ) = @VersionNumber;
EXECUTE UpdateWorkItems @WorkItemID, 'Completed'; -- Success
ELSE
EXECUTE ConcurrencyExceptionWorkItem; -- Row updated while processing
```

In Aurora PostgreSQL, you can add a trigger to maintain the updated stamp for each row.

```
CREATE OR REPLACE FUNCTION IncByOne()
  RETURNS TRIGGER
  AS $$
  BEGIN
    UPDATE WorkItems SET VersionNumber = VersionNumber+1
    WHERE WorkItemID = OLD.WorkItemID;
  END; $$
  LANGUAGE PLPGSQL;

CREATE TRIGGER MaintainWorkItemVersionNumber
  AFTER UPDATE OF WorkItems
  FOR EACH ROW
  EXECUTE PROCEDURE IncByOne();
```

For more information, see Triggers.

Syntax

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ]
table_name ( [
{ column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
| table_constraint
| LIKE source_table [ like_option ... ] }
[, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ PARTITION BY { RANGE | LIST } ( { column_name | ( expression ) } [ COLLATE collation
] [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ]
table_name
OF type_name [ (
{ column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
| table_constraint }
[, ... ]
) ]
[ PARTITION BY { RANGE | LIST } ( { column_name | ( expression ) } [ COLLATE collation
] [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ]
table_name
PARTITION OF parent_table [ (
{ column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
| table_constraint }
[, ... ]
) ] FOR VALUES partition_bound_spec
[ PARTITION BY { RANGE | LIST } ( { column_name | ( expression ) } [ COLLATE collation
] [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

The `column_constraint` is:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
NULL |
CHECK ( expression ) [ NO INHERIT ] |
DEFAULT default_expr |
GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
UNIQUE index_parameters |
PRIMARY KEY index_parameters |
REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
[ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

The `table_constraint` is:

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
UNIQUE ( column_name [, ... ] ) index_parameters |
PRIMARY KEY ( column_name [, ... ] ) index_parameters |
EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] )
 index_parameters
[ WHERE ( predicate ) ] |
FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
[ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE
action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

The `like_option` is:

```
{ INCLUDING | EXCLUDING } { COMMENTSDEFAULTS | CONSTRAINTS | DEFAULTS | IDENTITY |
INDEXES | STATISTICS | STORAGE |COMMENTS | ALL }
```

The `partition_bound_spec` is:

```
IN ( { numeric_literal | string_literal | TRUE | FALSE | NULL } [, ...] ) |
FROM ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE } [,
...] )
TO ( { numeric_literal | string_literal | TRUE | FALSE | MINVALUE | MAXVALUE } [,
...] )
```

The `index_parameters` in `UNIQUE`, `PRIMARY KEY`, and `EXCLUDE` constraints are:

```
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

The `exclude_element` in an `EXCLUDE` constraint is:

```
{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

Examples

Create a basic table.

```
CREATE TABLE MyTable
(
Col1 INT PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
);
```

Create a table with column constraints.

```
CREATE TABLE MyTable
(
Col1 INT PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
  CHECK (Col2 <> ''),
Col3 VARCHAR(100) NULL
  REFERENCES MyOtherTable (Col3)
);
```

Summary

Feature	SQL Server	Aurora PostgreSQL
ANSI compliance	Entry level	Entry level
Auto generated enumerator	`IDENTITY`	`SERIAL`
Reseed auto generated value	`DBCC CHECKIDENT`	N/A

Feature	SQL Server	Aurora PostgreSQL
Index types	CLUSTERED or NONCLUSTERED	See Indexes.
Physical storage location	ON <File Group>	Not supported
Temporary tables	#TempTable	CREATE TEMPORARY TABLE
Global temporary tables	##GlobalTempTable	CREATE GLOBAL TEMPORARY TABLE
Table variables	DECLARE @Table	Not supported
Create table as query	SELECT… INTO	CREATE TABLE… AS
Copy table structure	Not supported	CREATE TABLE… LIKE
Memory-optimized tables	Supported	N/A

For more information, see CREATE TABLE in the *PostgreSQL documentation*.

Common Table Expressions

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Use RECURSIVE keyword for recursive CTE queries.

SQL Server Usage

Common Table Expressions (CTE) are part of the ANSI standard since SQL:1999, simplify queries and make them more readable by defining a temporary view, or derived table, that a subsequent query can reference. You can use SQL Server CTEs as the target of DML modification statements. They have similar restrictions as updateable views.

SQL Server CTEs provide recursive functionality in accordance with the ANSI 99 standard. Recursive CTEs can reference themselves and re-run queries until the data set is exhausted, or the maximum number of iterations is exceeded.

CTE Syntax

```
WITH <CTE NAME>
AS
(
SELECT ....
)
SELECT ...
FROM CTE
```

Recursive CTE Syntax

```
WITH <CTE NAME>
AS (
<Anchor SELECT query>
UNION ALL
<Recursive SELECT query with reference to <CTE NAME>>
)
SELECT ... FROM <CTE NAME>...
```

Examples

Create and populate an OrderItems table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
```

```
(3, 'M8 Washer', 200),
(3, 'M6 Washer', 100);
```

Define a CTE to calculate the total quantity in every order and then join to the OrderItems table to obtain the relative quantity for each item.

```
WITH AggregatedOrders
AS
( SELECT OrderID, SUM(Quantity) AS TotalQty
FROM OrderItems
GROUP BY OrderID
)
SELECT O.OrderID, O.Item,
O.Quantity,
(O.Quantity / AO.TotalQty) * 100 AS PercentOfOrder
FROM OrderItems AS O
INNER JOIN
AggregatedOrders AS AO
ON O.OrderID = AO.OrderID;
```

The preceding example produces the following results.

```
OrderID  Item       Quantity  PercentOfOrder
1        M8 Bolt    100       100.0000000000
2        M8 Nut     100       100.0000000000
3        M8 Washer  100       33.3333333300
3        M6 Washer  200       66.6666666600
```

Using a recursive CTE, create and populate the `Employees` table with the `DirectManager` for each employee.

```
CREATE TABLE Employees
(
Employee VARCHAR(5) NOT NULL PRIMARY KEY,
DirectManager VARCHAR(5) NULL
);
```

```
INSERT INTO Employees(Employee, DirectManager)
VALUES
('John', 'Dave'),
('Jose', 'Dave'),
```

```
('Fred', 'John'),
('Dave', NULL);
```

Use a recursive CTE to display the employee-management hierarchy.

```
WITH EmpHierarchyCTE AS
(
  -- Anchor query retrieves the top manager
  SELECT 0 AS LVL,
    Employee,
    DirectManager
  FROM Employees AS E
  WHERE DirectManager IS NULL
  UNION ALL
-- Recursive query gets all Employees managed by the previous level
SELECT LVL + 1 AS LVL,
  E.Employee,
  E.DirectManager
FROM EmpHierarchyCTE AS EH
INNER JOIN
Employees AS E
ON E.DirectManager = EH.Employee
)
SELECT *
FROM EmpHierarchyCTE;
```

The preceding example produces the following results.

```
LVL  Employee  DirectManager
0    Dave      NULL
1    John      Dave
1    Jose      Dave
2    Fred      John
```

For more information, see Recursive Queries Using Common Table Expressions in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL conforms to the ANSI SQL-99 standard and implementing CTEs in PostgreSQL is similar to SQL Server.

CTE is also known as `WITH` query. This type of query helps you to simplify long queries, it is similar to defining temporary tables that exist only for the running of the query. The statement in a WITH clause can be a `SELECT`, `INSERT`, `UPDATE`, or `DELETE`, and the `WITH` clause itself is attached to a primary statement that can also be a `SELECT`, `INSERT`, `UPDATE`, or `DELETE`.

CTE Syntax

```
WITH <CTE NAME>
AS
(
SELECT OR DML
)
SELECT OR DML
Recursive CTE
```

Recursive CTE Syntax

```
WITH RECURSIVE <CTE NAME>
AS (
<Anchor SELECT query>
UNION ALL
<Recursive SELECT query with reference to <CTE NAME>>
)
SELECT OR DML
```

Examples

Create and populate an OrderItems table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
```

```
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Washer', 100);
```

Create a CTE.

```
WITH DEPT_COUNT
  (DEPARTMENT_ID, DEPT_COUNT) AS (
    SELECT DEPARTMENT_ID, COUNT(*) FROM EMPLOYEES GROUP BY DEPARTMENT_ID)
    SELECT E.FIRST_NAME ||' '|| E.LAST_NAME AS EMP_NAME,
    D.DEPT_COUNT AS EMP_DEPT_COUNT
    FROM EMPLOYEES E JOIN DEPT_COUNT D USING (DEPARTMENT_ID) ORDER BY 2;
```

PostgreSQL provides an additional feature when using a CTE as a recursive modifier. The following example uses a recursive WITH clause to access its own result set.

```
WITH RECURSIVE t(n) AS (
  VALUES (0)
  UNION ALL
  SELECT n+1 FROM t WHERE n < 5)
  SELECT * FROM t;

WITH RECURSIVE t(n) AS (
VALUES (0)
UNION ALL
SELECT n+1 FROM t WHERE n < 5)

SELECT * FROM t;

n
0
...
5
```

Note that using the SQL Server example will get undesired results.

Define a CTE to calculate the total quantity in every order and then join to the OrderItems table to obtain the relative quantity for each item.

```
WITH AggregatedOrders
```

```
AS
( SELECT OrderID, SUM(Quantity) AS TotalQty
  FROM OrderItems
  GROUP BY OrderID
)
SELECT O.OrderID, O.Item,
  O.Quantity,
    (O.Quantity / AO.TotalQty) * 100 AS PercentOfOrder
FROM OrderItems AS O
  INNER JOIN
  AggregatedOrders AS AO
  ON O.OrderID = AO.OrderID;
```

The preceding example produces the following results.

```
OrderID  Item       Quantity  PercentOfOrder
1        M8 Bolt    100       100
2        M8 Nut     100       100
3        M8 Washer  100       0
3        M6 Washer  200       0
```

This is because when you divide `INT` by `INT`, you get a round result. If you use another data type such as `DECIMAL`, there will be no problem. To fix the current issue, cast the columns using `::decimal`.

```
AS
( SELECT OrderID, SUM(Quantity) AS TotalQty
  FROM OrderItems
  GROUP BY OrderID
)
SELECT O.OrderID, O.Item,
  O.Quantity,
  trunc((O.Quantity::decimal / AO.TotalQty::decimal)*100,2) AS PercentOfOrder
  FROM OrderItems AS O
  INNER JOIN
  AggregatedOrders AS AO
  ON O.OrderID = AO.OrderID;
```

The preceding example produces the following results.

```
OrderID  Item       Quantity  PercentOfOrder
```

```
1         M8 Bolt    100       100
2         M8 Nut     100       100
3         M8 Washer  100       33.33
3         M6 Washer  200       66.66
```

Unlike in SQL Server, for `RECURSIVE WITH` query, use the `RECURSIVE` keyword in PostgreSQL.

Use a recursive CTE to display the employee-management hierarchy.

```
WITH RECURSIVE EmpHierarchyCTE AS
(
-- Anchor query retrieves the top manager
SELECT 0 AS LVL,
  Employee,
  DirectManager
FROM Employees AS E
WHERE DirectManager IS NULL
UNION ALL

-- Recursive query gets all Employees managed by the previous level
SELECT LVL + 1 AS LVL,
  E.Employee,
  E.DirectManager
FROM EmpHierarchyCTE AS EH
INNER JOIN
Employees AS E
ON E.DirectManager = EH.Employee
)
SELECT *
FROM EmpHierarchyCTE;
```

The preceding example produces the following results.

```
LVL  Employee  DirectManager
0    Dave
1    John      Dave
1    Jose      Dave
2    Fred      John
```

For more information, see [WITH Queries (Common Table Expressions)] in the *PostgreSQL documentation*.

Data Types

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Data Types	Syntax and handling differences.

SQL Server Usage

In SQL Server, each table column, variable, expression, and parameter has an associated data type. SQL Server provides a rich set of built-in data types as summarized in the following table.

Category	Data types
Numeric	`BIT`, `TINYINT`, `SMALLINT`, `INT`, `BIGINT`, `NUMERIC`, `DECIMAL`, `MONEY`, `SMALLMONEY`, `FLOAT`, `REAL`
String and Character	`CHAR`, `VARCHAR`, `NCHAR`, `NVARCHAR`
Temporal	`DATE`, `TIME`, `SMALLDATETIME`, `DATETIME`, `DATETIME2`, `DATETIMEOFFSET`
Binary	`BINARY`, `VARBINARY`
Large Object (LOB)	`TEXT`, `NTEXT`, `IMAGE`, `VARCHAR(MAX)`, `NVARCHAR(MAX)`, `VARBINARY(MAX)`
Cursor	`CURSOR`
GUID	`UNIQUEIDENTIFIER`
Hierarchical identifier	`HIERARCHYID`
Spatial	`GEOMETRY`, `GEOGRAPHY`

Category	Data types
Sets (table type)	TABLE
XML	XML
Other specialty types	ROW VERSION, SQL_VARIANT

You can create custom user defined data types using T-SQL, and the .NET Framework. Custom data types are based on the built-in system data types and are used to simplify development. For more information, see User-Defined Types.

TEXT, NTEXT, and IMAGE Deprecated Data Types

The TEXT, NTEXT, and IMAGE data types have been deprecated as of SQL Server 2008R2. For more information, see Deprecated Database Engine Features in SQL Server 2008 R2 in the *SQL Server documentation*.

These data types are legacy types for storing BLOB and CLOB data. The TEXT data type was used to store ASCII text CLOBS, the NTEXT data type to store UNICODE CLOBS, and IMAGE was used as a generic data type for storing all BLOB data. In SQL Server 2005, Microsoft introduced the new and improved VARCHAR (MAX), NVARCHAR(MAX), and VARBINARY(MAX) data types as the new BLOB and CLOB standard. These new types support a wider range of functions and operations. They also provide enhanced performance over the legacy types.

If your code uses TEXT, NTEXT or IMAGE data types, AWS SCT automatically converts them to the appropriate Aurora PostgreSQL BYTEA data type. Also, AWS SCT converts TEXT and NTEXT data types to LONGTEXT and IMAGE to LONGBLOB. Make sure you use the proper collations. For more information, see the SQL Server Collations and PostgreSQL Encoding.

Examples

Define table columns.

```
CREATE TABLE MyTable
(
Col1 AS INTEGER NOT NULL PRIMARY KEY,
Col2 AS NVARCHAR(100) NOT NULL
```

```
);
```

Define variable types.

```
DECLARE @MyXMLType AS XML,
  @MyTemporalType AS DATETIME2
```

```
DECLARE @MyTableType
AS TABLE
(
Col1 AS BINARY(16) NOT NULL PRIMARY KEY,
Col2 AS XML NULL
);
```

For more information, see Data types (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL provides multiple data types equivalent to certain SQL Server data types. The following tables include the full list of PostgreSQL data types.

Character data types

SQL Server data type	SQL Server data type characteristic	PostgreSQL identical compatibility	PostgreSQL corresponding data type
CHAR	Fixed length 1-8,000	Yes	CHAR
VARCHAR	Variable length 1-8,000	Yes	VARCHAR
NCHAR	Fixed length 1-4,000	Yes	NCHAR
NVARCHAR	Variable length 1-4,000	Yes	NVARCHAR

Numeric data types

SQL Server data type	SQL Server data type characteristic	PostgreSQL identical compatibility	PostgreSQL corresponding data type
BIT	First 8 BIT column will consume 1 byte, 9 to 16 BIT columns will be 2 bytes, and so on.	Yes	BIT
TINYINT	8-bit unsigned integer, 0 to 255	No	SMALLINT
SMALLINT	16-bit integer	Yes	SMALLINT
INT, INTEGER	32-bit integer	Yes	INT, INTEGER
BIGINT	64-bit integer	Yes	BIGINT
NUMERIC	Fixed-point number	Yes	NUMERIC
DECIMAL	Fixed-point number	Yes	DECIMAL
MONEY	64-bit currency amount	Yes	MONEY
SMALLMONEY	32-bit currency amount	No	MONEY
FLOAT	Floating-point number	Yes	FLOAT
REAL	Single-precision floating-point number	Yes	REAL

Temporal data types

SQL Server data type	SQL Server data type characteristic	PostgreSQL identical compatibility	PostgreSQL corresponding data type
DATE	Date (year, month and day)	Yes	DATE
TIME	Time (hour, minute, second and fraction)	Yes	TIME
SMALLDATETIME	Date and time	No	TIMESTAMP(0)
DATETIME	Date and time with fraction	No	TIMESTAMP(3)
DATETIME2	Date and time with fraction	No	TIMESTAMP(p)
DATETIMEOFFSET	Date and time with fraction and time zone	No	TIMESTAMP(p) WITH TIME ZONE

Binary data types

SQL Server data type	SQL Server data type characteristic	PostgreSQL identical compatibility	PostgreSQL corresponding data type
BINARY	Fixed-length byte string	No	BYTEA
VARBINARY	Variable length 1-8,000	No	BYTEA

LOB data types

SQL Server data type	SQL Server data type characteristic	PostgreSQL identical compatibility	PostgreSQL corresponding data type
TEXT	Variable-length character data up to 2 GB	Yes	TEXT
NTEXT	Variable-length Unicode UCS-2 data up to 2 GB	No	TEXT
IMAGE	Variable-length character data up to 2 GB	No	BYTEA
VARCHAR(MAX)	Variable-length character data up to 2 GB	Yes	TEXT
NVARCHAR(MAX)	Variable-length Unicode UCS-2 data up to 2 GB	No	TEXT
VARBINARY(MAX)	Variable-length character data up to 2 GB	No	BYTEA

Spatial data types

SQL Server data type	SQL Server data type characteristic	PostgreSQL identical compatibility	PostgreSQL corresponding data type
GEOMETRY	Euclidean (flat) coordinate system	Yes	GEOMETRY

SQL Server data type	SQL Server data type characteristic	PostgreSQL identical compatibility	PostgreSQL corresponding data type
GEOGRAPHY	Round-earth coordinate system	Yes	GEOGRAPHY
SQL_VARIANT	Maximum length of 8016	No	No equivalent

Other data types

SQL Server data type	SQL Server data type characteristic	PostgreSQL identical compatibility	PostgreSQL corresponding data type
XML	XML data	Yes	XML
UNIQUEIDENTIFIER	16-byte GUID (UUID)	No	CHAR(16)
HIERARCHYID	Approximately 5 bytes	No	NVARCHAR (4000)
ROWVERSION	8 bytes	No	TIMESTAMP(p)

PostgreSQL Character Column Semantics

PostgreSQL only supports `CHAR` for column size semantics. If you define a field as `VARCHAR (10)`, PostgreSQL can store 10 characters regardless of how many bytes it takes to store each non-English character. `VARCHAR(n)` stores strings up to n characters, not bytes, in length.

Migration of SQL Server Data Types to PostgreSQL Data Types

You can use AWS Schema Conversion Tool (AWS SCT) for automatic migration and conversion of SQL Server tables and data types.

Examples

To demonstrate AWS SCT capability for migrating SQL Server tables to their PostgreSQL equivalents, a table containing columns representing the majority of SQL Server data types was created and converted using AWS SCT.

Source SQL Server compatible DDL for creating the DATATYPES table

```
CREATE TABLE "DataTypes"(
  "BINARY_FLOAT" REAL,
  "BINARY_DOUBLE" FLOAT,
  "BLOB" VARBINARY(4000),
  "CHAR" CHAR(10),
  "CHARACTER" CHAR(10),
  "CLOB" VARCHAR(4000),
  "DATE" DATE,
  "DECIMAL" NUMERIC(3,2),
  "DOUBLE_PRECISION" FLOAT(52),
  "FLOAT" FLOAT(3),
  "INTEGER" INTEGER,
  "LONG" TEXT,
  "NCHAR" NCHAR(10),
  "NUMBER" NUMERIC(9,9),
  "NUMBER1" NUMERIC(9,0),
  "NUMERIC" NUMERIC(9,9),
  "RAW" BINARY(10),
  "REAL" FLOAT(52),
  "SMALLINT" SMALLINT,
  "TIMESTAMP" TIMESTAMP,
  "TIMESTAMP_WITH_TIME_ZONE" DATETIMEOFFSET(5),
  "VARCHAR" VARCHAR(10),
  "VARCHAR2" VARCHAR(10),
  "XMLTYPE" XML
);
```

Target PostgreSQL compatible DDL for creating the DATATYPES table migrated from SQL Server with AWS SCT.

```
CREATE TABLE IF NOT EXISTS datatypes(
  binary_float real DEFAULT NULL,
  binary_double double precision DEFAULT NULL,
  blob bytea DEFAULT NULL,
```

```
  char character(10) DEFAULT NULL,
  character character(10) DEFAULT NULL,
  clob text DEFAULT NULL,
  date TIMESTAMP(0) without time zone DEFAULT NULL,
  decimal numeric(3,2) DEFAULT NULL,
  dec numeric(3,2) DEFAULT NULL,
  double_precision double precision DEFAULT NULL,
  float double precision DEFAULT NULL,
  integer numeric(38,0) DEFAULT NULL,
  long text DEFAULT NULL,
  nchar character(10) DEFAULT NULL,
  number numeric(9,9) DEFAULT NULL,
  number1 numeric(9,0) DEFAULT NULL,
  numeric numeric(9,9) DEFAULT NULL,
  raw bytea DEFAULT NULL,
  real double precision DEFAULT NULL,
  smallint numeric(38,0) DEFAULT NULL,
  timestamp TIMESTAMP(5) without time zone DEFAULT NULL,
  timestamp_with_time_zone TIMESTAMP(5) with time zone DEFAULT NULL,
  varchar character varying(10) DEFAULT NULL,
  varchar2 character varying(10) DEFAULT NULL,
  xmltype xml DEFAULT NULL
)
WITH (
  OIDS=FALSE
);
```

Summary

AWS SCT converts all incompatible data types.

SQL Server CREATE TABLE command:

```
CREATE TABLE scttest(
SMALLDATETIMEcol SMALLDATETIME,
datetimecol DATETIME,
datetime2col DATETIME2,
datetimeoffsetcol DATETIMEOFFSET,
binarycol BINARY,
varbinarycol VARBINARY,
ntextcol NTEXT,
imagecol IMAGE,
nvarcharmaxcol NVARCHAR(MAX),
```

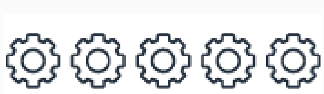
```
varbinarymaxcol VARBINARY(MAX),
uniqueidentifiercol UNIQUEIDENTIFIER,
hierarchyiDcol HIERARCHYID,
sql_variantcol SQL_VARIANT,
rowversioncol ROWVERSION);
```

The equivalent command that was created by AWS SCT:

```
CREATE TABLE scttest(
smalldatetimecol TIMESTAMP WITHOUT TIME ZONE,
datetimecol TIMESTAMP WITHOUT TIME ZONE,
datetime2col TIMESTAMP(6) WITHOUT TIME ZONE,
datetimeoffsetcol TIMESTAMP(6) WITH TIME ZONE,
binarycol BYTEA,
varbinarycol BYTEA,
ntextcol TEXT,
imagecol BYTEA,
nvarcharmaxcol TEXT,
varbinarymaxcol BYTEA,
uniqueidentifiercol UUID,
hierarchyidcol VARCHAR(8000),
sql_variantcol VARCHAR(8000),
rowversioncol VARCHAR(8000) NOT NULL);
```

For more information, see System Columns and Data Types in the *PostgreSQL documentation*, and Schema Conversion Tool Documentation.

Derived Tables

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	N/A

SQL Server Usage

SQL Server implements derived tables as specified in ANSI SQL:2011. Derived tables are similar to CTEs, but the reference to another query is used inside the `FROM` clause of a query.

This feature enables you to write more sophisticated, complex join queries.

Examples

```
SELECT name, salary, average_salary
FROM (SELECT AVG(salary)
  FROM employee) AS workers (average_salary), employee
WHERE salary > average_salary
ORDER BY salary DESC;
```

For more information, see [FROM clause plus JOIN, APPLY, PIVOT (Transact-SQL)] in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL implements derived tables and is fully compatible with SQL Server derived tables.

Examples

```
SELECT name, salary, average_salary
FROM (SELECT AVG(salary)
  FROM employee) AS workers (average_salary), employee
WHERE salary > average_salary
ORDER BY salary DESC;
```

For more information, see [Table Expressions] in the *PostgreSQL documentation*.

GROUP BY

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	N/A

SQL Server Usage

GROUP BY is an ANSI SQL query clause used to group individual rows that have passed the WHERE filter clause into groups to be passed on to the HAVING filter and then to the SELECT list. This grouping supports the use of aggregate functions such as SUM, MAX, AVG, and others.

Syntax

ANSI compliant GROUP BY Syntax.

```
GROUP BY
[ROLLUP | CUBE]
<Column Expression> ...n
[GROUPING SETS (<Grouping Set>)...n
```

Backward compatibility syntax.

```
GROUP BY
  [ ALL ] <Column Expression> ...n
  [ WITH CUBE | ROLLUP ]
```

The basic ANSI syntax for GROUP BY supports multiple grouping expressions, the CUBE and ROLLUP keywords, and the GROUPING SETS clause; all used to add super-aggregate rows to the output.

Up to SQL Server 2008 R2, the database engine supported a legacy, proprietary syntax (not ANSI Compliant) using the WITH CUBE and WITH ROLLUP clauses. These clauses added super-aggregates to the output.

Also, up to SQL Server 2008 R2, SQL Server supported the GROUP BY ALL syntax, which was used to create an empty group for rows that failed the WHERE clause.

SQL Server supports the following aggregate functions: AVG, CHECKSUM_AGG, COUNT, COUNT_BIG, GROUPING, GROUPING_ID, STDEV, STDEVP, STRING_AGG, SUM, MIN, MAX, VAR, VARP.

Examples

Legacy CUBE and ROLLUP Syntax

```
CREATE TABLE Orders
```

```
(
  OrderID INT IDENTITY(1,1) NOT NULL
  PRIMARY KEY,
  Customer VARCHAR(20) NOT NULL,
  OrderDate DATE NOT NULL
);
```

```
INSERT INTO Orders(Customer, OrderDate)
VALUES ('John', '20180501'), ('John', '20180502'), ('John', '20180503'),
  ('Jim', '20180501'), ('Jim', '20180503'), ('Jim', '20180504')
```

```
SELECT Customer,
  OrderDate,
  COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY Customer, OrderDate
WITH ROLLUP
```

The preceding example produces the following results.

```
Customer  OrderDate   NumOrders
Jim       2018-05-01  1
Jim       2018-05-03  1
Jim       2018-05-04  1
Jim       NULL        3
John      2018-05-01  1
John      2018-05-02  1
John      2018-05-03  1
John      NULL        3
NULL      NULL        6
```

The rows with NULL were added as a result of the `WITH ROLLUP` clause and contain super aggregates for the following:

- All orders for Jim and John regardless of OrderDate.
- A super aggregated for all customers and all dates.

Using `CUBE` instead of `ROLLUP` adds super aggregates in all possible combinations, not only in `GROUP BY` expression order.

```
SELECT Customer,
  OrderDate,
  COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY Customer, OrderDate
WITH CUBE
```

The preceding example produces the following results.

```
Customer  OrderDate   NumOrders
Jim       2018-05-01  1
John      2018-05-01  1
NULL      2018-05-01  2
John      2018-05-02  1
NULL      2018-05-02  1
Jim       2018-05-03  1
John      2018-05-03  1
NULL      2018-05-03  2
Jim       2018-05-04  1
NULL      2018-05-04  1
NULL      NULL        6
Jim       NULL        3
John      NULL        3
```

Four additional rows were added by the CUBE. They provide super aggregates for every date for all customers that were not part of the ROLLUP results in the preceding example.

Legacy GROUP BY ALL

Use the Orders table from the previous example.

```
SELECT Customer, OrderDate, COUNT(*) AS NumOrders
FROM Orders AS O
WHERE OrderDate <= '20180503'
GROUP BY ALL Customer, OrderDate
```

The preceding example produces the following results.

```
Customer  OrderDate   NumOrders
Jim       2018-05-01  1
```

```
John      2018-05-01  1
John      2018-05-02  1
Jim       2018-05-03  1
John      2018-05-03  1
Jim       2018-05-04  0
Warning: Null value is eliminated by an aggregate or other SET operation.
```

The last row failed the WHERE clause and was returned as an empty group as indicated by the warning for the empty `COUNT(*) = 0`.

Use GROUPING SETS

The following query uses the ANSI compliant `GROUPING SETS` syntax to provide all possible aggregate combinations for the Orders table, similar to the result of the `CUBE` syntax. This syntax requires specifying each dimension that needs to be aggregated.

```
SELECT Customer, OrderDate, COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY GROUPING SETS (
  (Customer, OrderDate),
  (Customer),
  (OrderDate),
  ()
)
```

The preceding example produces the following results.

```
Customer  OrderDate   NumOrders
Jim       2018-05-01  1
John      2018-05-01  1
NULL      2018-05-01  2
John      2018-05-02  1
NULL      2018-05-02  1
Jim       2018-05-03  1
John      2018-05-03  1
NULL      2018-05-03  2
Jim       2018-05-04  1
NULL      2018-05-04  1
NULL      NULL        6
Jim       NULL        3
John      NULL        3
```

For more information, see Aggregate Functions (Transact-SQL) and SELECT - GROUP BY- Transact-SQL in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) supports the basic ANSI syntax for `GROUP BY` and also supports `GROUPING SETS CUBE`, and `ROLLUP`.

In Aurora PostgreSQL, you can use `ROLLUP` and `ORDER BY` clauses in the same query, but the syntax is different from SQL Server. There is no `WITH` clause in the statement.

```
SELECT Customer, OrderDate, COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY ROLLUP (Customer, OrderDate)
```

The main difference is the need to move from writing the column to `GROUP BY` after the `ROLLUP`.

For the `CUBE` option, it's the same change.

```
SELECT Customer, OrderDate, COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY CUBE (Customer, OrderDate);
```

For the `GROUPING SET`, use the following query.

```
SELECT Customer, OrderDate, COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY GROUPING SETS (
  (Customer, OrderDate),
  (Customer),
  (OrderDate),
  ()
);
```

For more information, see Table Expressions in the *PostgreSQL documentation*.

Syntax

```
SELECT <Select List>
```

```
FROM <Table Source>
WHERE <Row Filter>
GROUP BY
  [ROLLUP | CUBE | GROUPING SETS]
<Column Name> | <Expression> | <Position>
```

Migration Considerations

The GROUP BY functionality exists except for the ALL option.

Convert every query to use the column name after the GROUP BY option, such as CUBE, ROLLUP, or CUBE.

Examples

Rewrite SQL Server WITH CUBE modifier for migration.

```
CREATE TABLE Orders
(
  OrderID serial NOT NULL
  PRIMARY KEY,
  Customer VARCHAR(20) NOT NULL,
  OrderDate DATE NOT NULL
);
```

```
INSERT INTO Orders(Customer, OrderDate)
VALUES ('John', '20180501'), ('John', '20180502'), ('John', '20180503'),
  ('Jim', '20180501'), ('Jim', '20180503'), ('Jim', '20180504');
```

```
SELECT Customer, OrderDate, COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY CUBE (Customer, OrderDate);
```

The preceding example produces the following results.

```
Customer  OrderDate   NumOrders
Jim       2018-05-01  1
Jim       2018-05-03  1
Jim       2018-05-04  1
```

```
Jim       NULL        3
John      2018-05-01  1
John      2018-05-02  1
John      2018-05-03  1
John      NULL        3
NULL      NULL        6
NULL      2018-05-01  2
NULL      2018-05-02  1
NULL      2018-05-03  2
NULL      2018-05-04  1
```

Rewrite SQL Server `GROUP BY ALL` for migration.

```
SELECT Customer, OrderDate, COUNT(*) AS NumOrders
FROM Orders AS O
WHERE OrderDate <= '20180503'
GROUP BY Customer, OrderDate
UNION ALL -- Add the empty groups
SELECT DISTINCT Customer, OrderDate, 0
FROM Orders AS O
WHERE OrderDate > '20180503';
```

The preceding example produces the following results.

```
Customer  OrderDate   NumOrders
Jim       2018-05-01  1
Jim       2018-05-03  1
John      2018-05-01  1
John      2018-05-02  1
John      2018-05-03  1
Jim       2018-05-04  0
```

Summary

The following table shows similarities, differences, and key migration considerations.

SQL Server feature	Aurora PostgreSQL feature	Comments
`MAX`, `MIN`, `AVG`, `COUNT`, `COUNT_BIG`	`MAX`, `MIN`, `AVG`, `COUNT`	In Aurora PostgreSQL, `COUNT` returns a `BIGINT` and is

SQL Server feature	Aurora PostgreSQL feature	Comments
		compatible with SQL Server COUNT and COUNT_BIG .
CHECKSUM_AGG	N/A	Use a loop to calculate checksums.
GROUPING, GROUPING_ID	GROUPING	Reconsider the query logic to avoid having NULL groups that are ambiguous with the super aggregates.
STDEV, STDEVP, VAR, VARP	STDDEV, STDDEV_POP , VARIANCE, VAR_POP	Rewrite keywords only.
STRING_AGG	STRING_AGG	
WITH ROLLUP	ROLLUP	Remove WITH and change the columns names to be after the ROLLUP keyword.
WITH CUBE	CUBE	Remove WITH and change the columns names to be after the CUBE keyword.
GROUPING SETS	GROUPING SETS	

For more information, see Aggregate Functions in the *PostgreSQL documentation*.

Table JOIN

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	OUTER JOIN with commas. CROSS

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
			APPLY and OUTER APPLY aren't supported.

SQL Server Usage

ANSI JOIN

SQL Server supports the standard ANSI join types.

- `<Set A> CROSS JOIN <Set B>`. Results in a Cartesian product of the two sets. Every JOIN starts as a Cartesian product.
- `<Set A> INNER JOIN <Set B> ON <Join Condition>`. Filters the Cartesian product to only the rows where the join predicate evaluates to TRUE.
- `<Set A> LEFT OUTER JOIN <Set B> ON <Join Condition>`. Adds to the INNER JOIN all the rows from the reserved left set with NULL for all the columns that come from the right set.
- `<Set A> RIGHT OUTER JOIN <Set B> ON <Join Condition>` Adds to the INNER JOIN all the rows from the reserved right set with NULL for all the columns that come from the left set.
- `<Set A> FULL OUTER JOIN <Set B> ON <Join Condition>`. Designates both sets as reserved and adds non-matching rows from both, similar to a LEFT OUTER JOIN and a RIGHT OUTER JOIN.

APPLY

SQL Server also supports the APPLY operator, which is somewhat similar to a join. However, APPLY operators enable the creation of a correlation between <Set A> and <Set B> such that <Set B> may consist of a sub query, a VALUES row value constructor, or a table valued function that is evaluated for each row of <Set A> where the <Set B> query can reference columns from the current row in <Set A>. This functionality isn't possible with any type of standard JOIN operator.

There are two APPLY types:

- `<Set A> CROSS APPLY <Set B>`. Similar to a `CROSS JOIN` in the sense that every row from `<Set A>` is matched with every row from `<Set B>`.
- `<Set A> OUTER APPLY <Set B>`. Similar to a `LEFT OUTER JOIN` in the sense that rows from `<Set A>` are returned even if the sub query for `<Set B>` produces an empty set. In that case, NULL is assigned to all columns of `<Set B>`.

ANSI SQL 89 JOIN

Up until version 2008R2, SQL Server also supported the old-style `JOIN` syntax including `LEFT` and `RIGHT OUTER JOIN`.

The ANSI syntax for a `CROSS JOIN` operator was to list the sets in the `FROM` clause using commas as separators.

```
SELECT * FROM Table1,
  Table2,
  Table3...
```

To perform an `INNER JOIN`, you only needed to add the `JOIN` predicate as part of the `WHERE` clause.

```
SELECT * FROM Table1,
  Table2
WHERE Table1.Column1 = Table2.Column1
```

Although the ANSI standard didn't specify outer joins at the time, most RDBMS supported them in one way or another. T-SQL supported outer joins by adding an asterisk to the left or the right of equality sign of the join predicate to designate the reserved table.

```
SELECT * FROM Table1,
  Table2
WHERE Table1.Column1 *= Table2.Column1
```

To perform a `FULL OUTER JOIN`, asterisks were placed on both sides of the equality sign of the join predicate.

As of SQL Server 2008R2, outer joins using this syntax have been deprecated. For more information, see Deprecated Database Engine Features in SQL Server 2008 R2 in the *SQL Server documentation*.

> **Note**
>
> Even though `INNER JOIN` using the ANSI SQL 89 syntax is still supported, they are highly discouraged due to being notorious for introducing hard-to-catch programming bugs.

Syntax

CROSS JOIN

```
FROM <Table Source 1>
  CROSS JOIN
  <Table Source 2>
```

```
-- ANSI 89
FROM <Table Source 1>,
  <Table Source 2>
```

INNER / OUTER JOIN

```
FROM <Table Source 1>
  [ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } }] JOIN
  <Table Source 2>
  ON <JOIN Predicate>
```

```
-- ANSI 89
FROM <Table Source 1>,
  <Table Source 2>
WHERE <Join Predicate>
<Join Predicate>:: <Table Source 1 Expression> | = | *= | =* | *=* <Table Source 2
 Expression>
```

APPLY

```
FROM <Table Source 1>
  { CROSS | OUTER } APPLY
```

```
  <Table Source 2>
<Table Source 2>:: <SELECT sub-query> | <Table Valued UDF> | <VALUES clause>
```

Examples

Create the Orders and Items tables.

```
CREATE TABLE Items
(
  Item VARCHAR(20) NOT NULL
  PRIMARY KEY
  Category VARCHAR(20) NOT NULL,
  Material VARCHAR(20) NOT NULL
);
```

```
INSERT INTO Items (Item, Category, Material)
VALUES
('M8 Bolt', 'Metric Bolts', 'Stainless Steel'),
('M8 Nut', 'Metric Nuts', 'Stainless Steel'),
('M8 Washer', 'Metric Washers', 'Stainless Steel'),
('3/8" Bolt', 'Imperial Bolts', 'Brass')
```

```
CREATE TABLE OrderItems
(
  OrderID INT NOT NULL,
  Item VARCHAR(20) NOT NULL
  REFERENCES Items(Item),
  Quantity SMALLINT NOT NULL,
  PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200)
```

INNER JOIN

```
SELECT *
```

```
FROM Items AS I
  INNER JOIN
  OrderItems AS OI
  ON I.Item = OI.Item;
-- ANSI SQL 89
SELECT *
FROM Items AS I,
  OrderItems AS OI
WHERE I.Item = OI.Item;
```

LEFT OUTER JOIN

Find Items that were never ordered.

```
SELECT I.Item
FROM Items AS I
  LEFT OUTER JOIN
  OrderItems AS OI
  ON I.Item = OI.Item
WHERE OI.OrderID IS NULL;

-- ANSI SQL 89
SELECT Item
FROM
(
  SELECT I.Item, O.OrderID
  FROM Items AS I,
    OrderItems AS OI
  WHERE I.Item *= OI.Item
) AS LeftJoined
WHERE LeftJoined.OrderID IS NULL;
```

FULL OUTER JOIN

```
CREATE TABLE T1(Col1 INT, COl2 CHAR(2));
CREATE TABLE T2(Col1 INT, COl2 CHAR(2));

INSERT INTO T1 (Col1, Col2)
VALUES (1, 'A'), (2,'B');

INSERT INTO T2 (Col1, Col2)
VALUES (2,'BB'), (3,'CC');
```

```
SELECT *
FROM T1
  FULL OUTER JOIN
  T2
  ON T1.Col1 = T2.Col1;
```

The preceding example produces the following results.

```
Col1  COl2  Col1  COl2
1     A     NULL  NULL
2     B     2     BB
NULL  NULL  3     CC
```

For more information, see FROM clause plus JOIN, APPLY, PIVOT (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) supports all types of joins in the same way as SQL Server.

- `<Set A> CROSS JOIN <Set B>`. Results in a Cartesian product of the two sets. Every `JOIN` starts as a Cartesian product.
- `<Set A> INNER JOIN <Set B> ON <Join Condition>`. Filters the Cartesian product to only the rows where the join predicate evaluates to TRUE.
- `<Set A> LEFT OUTER JOIN <Set B> ON <Join Condition>`. Adds to the `INNER JOIN` all the rows from the reserved left set with NULL for all the columns that come from the right set.
- `<Set A> RIGHT OUTER JOIN <Set B> ON <Join Condition>` Adds to the `INNER JOIN` all the rows from the reserved right set with NULL for all the columns that come from the left set.
- `<Set A> FULL OUTER JOIN <Set B> ON <Join Condition>`. Designates both sets as reserved and adds non-matching rows from both, similar to a `LEFT OUTER JOIN` and a `RIGHT OUTER JOIN`.

PostgreSQL doesn't support `APPLY` options. You can replace them with `INNER JOIN LATERAL` and `LEFT JOIN LATERAL`.

Syntax

```
FROM
    <Table Source 1> CROSS JOIN <Table Source 2>
  | <Table Source 1> INNER JOIN <Table Source 2>
    ON <Join Predicate>
  | <Table Source 1> {LEFT|RIGHT|FULL} [OUTER] JOIN <Table Source 2>
    ON <Join Predicate>
```

Migration Considerations

For most `JOIN` statements, the syntax should be equivalent and no rewrites should be needed. Find the differences following.

- ANSI SQL 89 isn't supported.
- `FULL OUTER JOIN` and `OUTER JOIN` using the pre-ANSI SQL 92 syntax aren't supported, but you can use workarounds.
- `CROSS APPLY` and `OUTER APPLY` aren't supported. You can rewrite these statements using `INNER JOIN LATERAL` and `LEFT JOIN LATERAL`.

Examples

Create the Orders and Items tables.

```
CREATE TABLE Items
(
  Item VARCHAR(20) NOT NULL
  PRIMARY KEY
  Category VARCHAR(20) NOT NULL,
  Material VARCHAR(20) NOT NULL
);
```

```
INSERT INTO Items (Item, Category, Material)
VALUES
('M8 Bolt', 'Metric Bolts', 'Stainless Steel'),
('M8 Nut', 'Metric Nuts', 'Stainless Steel'),
('M8 Washer', 'Metric Washers', 'Stainless Steel'),
('3/8" Bolt', 'Imperial Bolts', 'Brass')
```

```
CREATE TABLE OrderItems
(
  OrderID INT NOT NULL,
  Item VARCHAR(20) NOT NULL
  REFERENCES Items(Item),
  Quantity SMALLINT NOT NULL,
  PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200)
```

INNER JOIN

```
SELECT *
FROM Items AS I
  INNER JOIN
  OrderItems AS OI
  ON I.Item = OI.Item;
```

LEFT OUTER JOIN

Find Items that were never ordered.

```
SELECT Item
FROM Items AS I
  LEFT OUTER JOIN
  OrderItems AS OI
  ON I.Item = OI.Item
WHERE OI.OrderID IS NULL;
```

FULL OUTER JOIN

```
CREATE TABLE T1(Col1 INT, COl2 CHAR(2));
CREATE TABLE T2(Col1 INT, COl2 CHAR(2));

INSERT INTO T1 (Col1, Col2)
VALUES (1, 'A'), (2,'B');
```

```
INSERT INTO T2 (Col1, Col2)
VALUES (2,'BB'), (3,'CC');

SELECT *
FROM T1
FULL OUTER JOIN
T2
ON T1.Col1 = T2.Col1;
```

The preceding example produces the following results.

```
Col1  COl2  Col1  COl2
1     A     NULL  NULL
2     B     2     BB
NULL  NULL  3     CC
```

Summary

The following table shows similarities, differences, and key migration considerations.

SQL Server feature	Aurora PostgreSQL feature	Comments
`INNER JOIN` with `ON` clause or commas.	Supported.	
`OUTER JOIN` with `ON` clause.	Supported.	
`OUTER JOIN` with commas.	Not supported.	Requires T-SQL rewrite post SQL Server 2008R2.
`CROSS JOIN` or using commas.	Supported.	
`CROSS APPLY` and `OUTER APPLY`.	Not supported.	Rewrite required.

For more information, see Controlling the Planner with Explicit JOIN Clauses and Joins Between Tables in the *PostgreSQL documentation*.

SQL Server Temporal Tables and PostgreSQL Triggers

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	N/A

SQL Server Usage

Temporal database tables were introduced in ANSI SQL 2011. T-SQL began supporting system versioned temporal tables in SQL Server 2016.

Each temporal table has two explicitly defined `DATETIME2` columns known as period columns. The system uses these columns to record the period of availability for each row when it is modified. An additional history table retains the previous version of the data. The system can automatically create the history table, or a user can specify an existing table.

To query the history table, use `FOR SYSTEM TIME` after the table name in the `FROM` clause and combine it with the following options:

- `ALL` — all changes.
- `CONTAINED IN` — change is valid only within a period.
- `AS OF` — change was valid somewhere in a specific period.
- `BETWEEN` — change was valid from a time range.

Temporal Tables are mostly used when to track data change history as described in the following scenarios.

Anomaly Detection

Use this option when searching for data with unusual values. For example, detecting when a customer returns items too often.

```
CREATE TABLE Products_returned
(
```

```
  ProductID int NOT NULL PRIMARY KEY CLUSTERED,
  ProductName varchar(60) NOT NULL,
  return_count INT NOT NULL,
  ValidFrom datetime2(7) GENERATED ALWAYS AS ROW START NOT NULL,
  ValidTo datetime2(7) GENERATED ALWAYS AS ROW END NOT NULL,
  PERIOD FOR SYSTEM_TIME (ValidFrom, ValidTo)
)
WITH( SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.ProductHistory,
  DATA_CONSISTENCY_CHECK = ON ))
```

Query the Product table and run calculations on the data.

```
SELECT
  ProductId,
  LAG (return_count, 1, 1)
  over (partition by ProductId order by ValidFrom) as PrevValue,
  return_count,
  LEAD (return_count, 1, 1)
  over (partition by ProductId order by ValidFrom) as NextValue ,
  ValidFrom, ValidTo from Product
FOR SYSTEM_TIME ALL
```

Audit

Track changes to critical data such as salaries or medical data.

```
CREATE TABLE Employee
(
  EmployeeID int NOT NULL PRIMARY KEY CLUSTERED,
  Name nvarchar(60) NOT NULL,
  Salary decimal (6,2) NOT NULL,
  ValidFrom datetime2 (2) GENERATED ALWAYS AS ROW START,
  ValidTo datetime2 (2) GENERATED ALWAYS AS ROW END,
  PERIOD FOR SYSTEM_TIME (ValidFrom, ValidTo)
)
WITH (SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.EmployeeTrackHistory));
```

Use `FOR SYSTEM_TIME ALL` to retrieve changes from the history table.

```
SELECT * FROM Employee
  FOR SYSTEM_TIME ALL WHERE
```

```
    EmployeeID = 1000 ORDER BY ValidFrom;
```

Other Scenarios

Additional scenarios include the following:

- Fixing row-level corruption.
- Slowly changing dimension.
- Over time changes analysis.

For more information, see Temporal tables in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL provides an extension for supporting temporal tables, but it's not supported by Amazon Aurora. A workaround will be to create table triggers to update a custom history table to track changes to data. For more information, see Triggers.

Views

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	PostgreSQL doesn't support indexed and partitioned views.

SQL Server Usage

Views are schema objects that provide stored definitions for virtual tables. Similar to tables, views are data sets with uniquely named columns and rows. With the exception of indexed views, view objects don't store data. They consist only of a query definition and are reevaluated for each invocation.

Views are used as abstraction layers and security filters for the underlying tables. They can `JOIN` and `UNION` data from multiple source tables and use aggregates, window functions, and other SQL features as long as the result is a semi-proper set with uniquely identifiable columns and no order to the rows. You can use distributed views to query other databases and data sources using linked servers.

As an abstraction layer, a view can decouple application code from the database schema. You can change the underlying tables without the need to modify the application code as long as the expected results of the view don't change. You can use this approach to provide backward compatible views of data.

As a security mechanism, a view can screen and filter source table data. You can perform permission management at the view level without explicit permissions to the base objects, provided the ownership chain is maintained. For more information, see Overview of SQL Server Security.

View definitions are evaluated when they are created and aren't affected by subsequent changes to the underlying tables. For example, a view that uses `SELECT *` doesn't display columns that were added later to the base table. Similarly, if a column was dropped from the base table, invoking the view results in an error. Use the `SCHEMABINDING` option to prevent changes to base objects.

Modifying Data Through Views

Updatable Views can both select and modify data. Updatable views meet the following conditions:

- The DML targets only one base table.
- Columns being modified must be directly referenced from the underlying base tables. Computed columns, set operators, functions, aggregates, or any other expressions aren't permitted.
- If a view is created with the CHECK OPTION, rows being updated can't be filtered out of the view definition as the result of the update.

Special View Types

SQL Server provides three types of specialized views:

- **Indexed views**. These views are also known as materialized views or persisted views. Indexed vires are standard views that have been evaluated and persisted in a unique clustered index, much like a normal clustered primary key table. Each time the source data changes, SQL Server re-evaluates the indexed views automatically and updates them. Indexed views are typically used as a means to optimize performance by pre-processing operators such as aggregations, joins, and others. Queries needing this pre-processing don't have to wait for it to be reevaluated on every query run.

- **Partitioned views** rejoin horizontally partitioned data sets from multiple underlying tables, each containing only a subset of the data. The view uses a `UNION ALL` query where the underlying tables can reside locally or in other databases (or even other servers). These types of views are called Distributed Partitioned Views (DPV).
- **System views** access server and object meta data. SQL Server also supports a set of standard `INFORMATION_SCHEMA` views for accessing object meta data.

Syntax

```
CREATE [OR ALTER] VIEW [<Schema Name>.] <View Name> [(<Column Aliases> ])]
[WITH [ENCRYPTION][SCHEMABINDING][VIEW_METADATA]]
AS <SELECT Query>
[WITH CHECK OPTION][;]
```

Examples

The following example creates a view that aggregates items for each customer.

```
CREATE TABLE Orders
(
  OrderID INT NOT NULL PRIMARY KEY,
  OrderDate DATETIME NOT NULL
  DEFAULT GETDATE()
);
```

```
CREATE TABLE OrderItems
(
  OrderID INT NOT NULL
  REFERENCES Orders(OrderID),
  Item VARCHAR(20) NOT NULL,
  Quantity SMALLINT NOT NULL,
  PRIMARY KEY(OrderID, Item)
);
```

```
CREATE VIEW SalesView
```

```
AS
SELECT O.Customer,
  OI.Product,
  SUM(CAST(OI.Quantity AS BIGINT)) AS TotalItemsBought
FROM Orders AS O
  INNER JOIN
  OrderItems AS OI
  ON O.OrderID = OI.OrderID;
```

The following example creates an indexed view that pre-aggregates items for each customer

```
CREATE VIEW SalesViewIndexed
AS
SELECT O.Customer,
  OI.Product,
  SUM_BIG(OI.Quantity) AS TotalItemsBought
FROM Orders AS O
  INNER JOIN
  OrderItems AS OI
  ON O.OrderID = OI.OrderID;
```

```
CREATE UNIQUE CLUSTERED INDEX IDX_SalesView
ON SalesViewIndexed (Customer, Product);
```

The following example creates a partitioned view.

```
CREATE VIEW dbo.PartitioneView
WITH SCHEMABINDING
AS
SELECT *
FROM Table1
UNION ALL
SELECT *
FROM Table2
UNION ALL
SELECT *
FROM Table3
```

For more information, see Views, Modify Data Through a View, and CREATE VIEW (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

The basic form of views is similar between PostgreSQL and SQL Server. A view defines a stored query based on one or more physical database tables that runs every time the view is accessed.

More complex option such as indexed views or partitioned views aren't supported, and may require a redesign or might application rewrite.

> **Note**
>
> For Amazon Relational Database Service (Amazon RDS), starting with PostgreSQL 13, you can rename view columns using `ALTER VIEW` command. This option helps DBAs avoid dropping and recreating the view to change a column name.
> Use the following syntax to rename a column name in a view: `ALTER VIEW [ IF EXISTS ] name RENAME [ COLUMN ] column_name TO new_column_name`.
> For PostgreSQL versions lower than 13, you can change the column name in a view using the `ALTER TABLE` command.

PostgreSQL View Privileges

To create a view, make sure that you grant `SELECT` and `DML` privileges on the base tables or views to your role or user. For more information, see [GRANT](#) in the *PostgreSQL documentation*.

PostgreSQL View Parameters

CREATE [OR REPLACE] VIEW

When you re-create an existing view, make sure that the new view has the same column structure as generated by the original view. The column structure includes column names, column order, and data types. It is sometimes preferable to drop the view and use the `CREATE VIEW` statement instead.

```
hr=# CREATE [OR REPLACE] VIEW VW_NAME AS
SELECT COLUMNS
FROM TABLE(s)
[WHERE CONDITIONS];

hr=# DROP VIEW [IF EXISTS] VW_NAME;
```

In the example preceding, the `IF EXISTS` parameter is optional.

WITH [CASCADED | LOCAL] CHECK OPTION

DML `INSERT` and `UPDATE` operations are verified against the view-based tables to ensure new rows satisfy the original structure conditions or the view-defining condition. If a conflict is detected, the DML operation fails.

- `LOCAL`. Verifies the view without a hierarchical check.
- `CASCADED`. Verifies all underlying base views using a hierarchical check.

Running DML Commands On Views

PostgreSQL simple views are automatically updatable. No restrictions exist when performing DML operations on views. An updatable view may contain a combination of updatable and non-updatable columns. A column is updatable if it references an updatable column of the underlying base table. If not, the column is read-only and an error is raised if an `INSERT` or `UPDATE` statement is attempted on the column.

Syntax

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name
 [,...] ) ]
[ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
AS query
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Examples

The following example creates and updates a view without the `CHECK OPTION` parameter.

```
CREATE OR REPLACE VIEW VW_DEP AS
  SELECT DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID
  FROM DEPARTMENTS
  WHERE LOCATION_ID=1700;

view VW_DEP created.

UPDATE VW_DEP SET LOCATION_ID=1600;
```

```
21 rows updated.
```

The following example creates and updates a view with the LOCAL CHECK OPTION parameter.

```
CREATE OR REPLACE VIEW VW_DEP AS
  SELECT DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID
  FROM DEPARTMENTS
  WHERE LOCATION_ID=1700
  WITH LOCAL CHECK OPTION;

view VW_DEP created.

UPDATE VW_DEP SET LOCATION_ID=1600;

SQL Error: ERROR: new row violates check option for view "vw_dep"
```

Summary

Feature	SQL Server	Aurora PostgreSQL
Indexed views	Supported	N/A
Partitioned views	Supported	N/A
Updateable views	Supported	Supported
Prevent schema conflicts	`SCHEMABINDING` option	N/A
Triggers on views	`INSTEAD OF`	`INSTEAD OF`
Temporary Views	`CREATE VIEW #View…`	`CREATE [ OR REPLACE ] [ TEMP ] [ TEMPORARY ] VIEW`
Refresh view definition	`sp_refreshview` / `ALTER VIEW`	`ALTER VIEW`

For more information, see [Views](#) and [CREATE VIEW](#) in the *PostgreSQL documentation*.

Window Functions

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	N/A

SQL Server Usage

Window functions use an OVER clause to define the window and frame for a data set to be processed. They are part of the ANSI standard and are typically compatible among various SQL dialects. However, most RDBMS don't yet support the full ANSI specification.

Window functions are a relatively new, advanced, and efficient T-SQL programming tool. They are highly utilized by developers to solve numerous programming challenges.

SQL Server currently supports the following window functions:

- Ranking functions: ROW_NUMBER, RANK, DENSE_RANK, and NTILE.
- Aggregate functions: AVG, MIN, MAX, SUM, COUNT, COUNT_BIG, VAR, STDEV, STDEVP, STRING_AGG, GROUPING, GROUPING_ID, VAR, VARP, and CHECKSUM_AGG.
- Analytic functions: LAG, LEAD, FIRST_Value, LAST_VALUE, PERCENT_RANK, PERCENTILE_CONT, PERCENTILE_DISC, and CUME_DIST.
- Other functions: NEXT_VALUE_FOR. For more information, see Sequences and Identity.

Syntax

```
<Function()>
OVER
(
[ <PARTITION BY clause> ]
[ <ORDER BY clause> ]
[ <ROW or RANGE clause> ]
)
```

Examples

The following example creates and populates an OrderItems table.

```
CREATE TABLE OrderItems
(
  OrderID INT NOT NULL,
  Item VARCHAR(20) NOT NULL,
  Quantity SMALLINT NOT NULL,
  PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Locking Nut', 300);
```

The following example uses a window ranking function to rank items based on the ordered quantity.

```
SELECT Item,
  Quantity,
RANK() OVER(ORDER BY Quantity) AS QtyRank
FROM OrderItems;
```

The preceding example produces the following results.

```
Item            Quantity  QtyRank
M8 Bolt         100       1
M8 Nut          100       1
M8 Washer       200       3
M6 Locking Nut  300       4
```

The following example uses a partitioned window aggregate function to calculate the total quantity for each order. This statement doesn't use a `GROUP BY` clause.

```
SELECT Item,
Quantity,
OrderID,
```

```
SUM(Quantity)
OVER (PARTITION BY OrderID) AS TotalOrderQty
FROM OrderItems;
```

The preceding example produces the following results.

```
Item            Quantity  QtyRank  TotalOrderQty
M8 Bolt         100       1        100
M8 Nut          100       2        100
M6 Locking Nut  300       3        500
M8 Washer       200       3        500
```

The following example uses an analytic LEAD function to get the next largest quantity for the order.

```
SELECT Item,
  Quantity,
  OrderID,
  LEAD(Quantity)
  OVER (PARTITION BY OrderID ORDER BY Quantity) AS NextQtyOrder
FROM OrderItems;
```

The preceding example produces the following results.

```
Item            Quantity  OrderID  NextQtyOrder
M8 Bolt         100       1        NULL
M8 Nut          100       2        NULL
M8 Washer       200       3        300
M6 Locking Nut  300       3        NULL
```

For more information, see SELECT - OVER Clause (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL refers to ANSI SQL analytical functions as window functions. They provide the same core functionality as SQL Server analytical functions. Window functions in PostgreSQL operate on a logical partition or window of the result set and return a value for rows in that window.

From a database migration perspective, you should examine PostgreSQL window functions by type and compare them with the equivalent SQL Server window functions to verify compatibility of syntax and output.

Note

Even if a PostgreSQL window function provides the same functionality of a specific SQL Server window function, the returned data type may be different and require application changes.

PostgreSQL provides support for two main types of window functions: aggregation functions and ranking functions.

PostgreSQL Window Functions by Type

Function type	Related functions
Aggregate	`avg`, `count`, `max`, `min`, `sum`, `string_agg`
Ranking	`row_number` , `rank`, `dense_rank` , `percent_rank` , `cume_dist` , `ntile`, `lag`, `lead`, `first_value` , `last_value` , `nth_value`

PostgreSQL Window Functions

PostgreSQL window function	Returned data type	Compatible syntax
Count	bigint	Yes
Max	numeric, string, date/time, network or enum type	Yes
Min	numeric, string, date/time, network or enum type	Yes
Avg	numeric, double, otherwise same data type as the argument	Yes

PostgreSQL window function	Returned data type	Compatible syntax
Sum	bigint, otherwise same data type as the argument	Yes
rank()	bigint	Yes
row_number()	bigint	Yes
dense_rank()	bigint	Yes
percent_rank()	double	Yes
cume_dist()	double	Yes
ntile()	integer	Yes
lag()	Same type as value	Yes
lead()	Same type as value	Yes
first_value()	Same type as value	Yes
last_value()	Same type as value	Yes

Examples

The following example uses he PostgreSQL `rank()` function.

```
SELECT department_id, last_name, salary, commission_pct,
RANK() OVER (PARTITION BY department_id
ORDER BY salary DESC, commission_pct) "Rank"
FROM employees WHERE department_id = 80;

DEPARTMENT_ID  LAST_NAME  SALARY    COMMISSION_PCT  Rank
80             Russell    14000.00  0.40            1
80             Partners   13500.00  0.30            2
80             Errazuriz  12000.00  0.30            3
```

The returned formatting for certain numeric data types is different.

The following example calculates the total salary for the department 80.

```
SELECT SUM(salary)
FROM employees WHERE department_id = 80;

SUM(SALARY)
39500.00
```

The following example creates and populates an OrderItems table.

```
CREATE TABLE OrderItems
(
  OrderID INT NOT NULL,
  Item VARCHAR(20) NOT NULL,
  Quantity SMALLINT NOT NULL,
  PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity) VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Locking Nut', 300);
```

The following example uses a window ranking function to rank items based on the ordered quantity.

```
SELECT Item, Quantity, RANK()
  OVER(ORDER BY Quantity) AS QtyRank
FROM OrderItems;

Item            Quantity  QtyRank
M8 Bolt         100       1
M8 Nut          100       1
M8 Washer       200       3
M6 Locking Nut  300       4
```

The following example uses a partitioned window aggregate function to calculate the total quantity for each order. This statement doesn't use a `GROUP BY` clause.

```
SELECT Item, Quantity, OrderID, SUM(Quantity)
```

```
  OVER (PARTITION BY OrderID) AS TotalOrderQty
FROM OrderItems;

Item            Quantity  OrderID  TotalOrderQty
M8 Bolt         100       1        100
M8 Nut          100       2        100
M6 Locking Nut  300       3        500
M8 Washer       200       3        500
```

The following example uses an analytic LEAD function to get the next largest quantity for the order.

```
SELECT Item, Quantity, OrderID, LEAD(Quantity)
  OVER (PARTITION BY OrderID ORDER BY Quantity) AS NextQtyOrder
FROM OrderItems;

Item            Quantity  OrderID  NextQtyOrder
M8 Bolt         100       1        NULL
M8 Nut          100       2        NULL
M8 Washer       200       3        300
M6 Locking Nut  300       3        NULL
```

For more information, see Window Functions in the *PostgreSQL documentation*.

T-SQL

Topics

- SQL Server Service Broker Essentials and PostgreSQL AWS Lambda or DB links
- SQL Server Cast and Convert and PostgreSQL CAST and CONVERSION
- SQL Server Common Library Runtime and PostgreSQL PL/Perl
- SQL Server Collations and PostgreSQL Encoding
- Cursors
- Date and Time Functions
- String Functions
- Databases and Schemas
- SQL Server Dynamic SQL and PostgreSQL EXECUTE and PREPARE
- Transactions
- SQL Server Synonyms and PostgreSQL Views, Types, and Functions
- DELETE and UPDATE FROM
- Stored Procedures
- Error Handling
- SQL Server Flow Control and PostgreSQL Control Structures
- Full-Text Search
- SQL Server Graph
- JSON and XML
- MERGE
- PIVOT and UNPIVOT
- Triggers
- SQL Server TOP and FETCH and PostgreSQL LIMIT and OFFSET
- User-Defined Functions
- User-Defined Types
- Sequences and Identity

SQL Server Service Broker Essentials and PostgreSQL AWS Lambda or DB links

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Service Broker	Use Amazon Lambda for similar functionality.

SQL Server Usage

SQL Server Service Broker provides native support for messaging and queuing applications. Developers use Server Broker to create complex applications that use the database engine components to communicate between several SQL Server databases. Developers can use Service Broker to easily build distributed and more reliable applications.

Benefits of using messaging queues:

- Decouple dependencies between applications by communicating through messages.
- Scale out your architecture by moving queues or message processors to separate servers as needed.
- Maintain individual parts with a minimal impact to the end users.
- Control when the messages are processed, for example, off-peak hours.
- Process queued messages on multiple servers or processes or threads.

The following sections describe the Service Broker commands.

CREATE MESSAGE TYPE

The following example creates a message with name and structure.

```
CREATE MESSAGE TYPE message_type_name
  [ AUTHORIZATION owner_name ]
```

```
  [ VALIDATION = { NONE
    | EMPTY
    | WELL_FORMED_XML
    | VALID_XML WITH SCHEMA COLLECTION schema_collection_name
  } ]
[ ; ]
```

For more information, see [CREATE MESSAGE TYPE (Transact-SQL)]() in the *SQL Server documentation*.

CREATE QUEUE

The following example creates a queue to store messages.

```
CREATE QUEUE <object>
  [ WITH
    [ STATUS = { ON | OFF } [ , ] ]
    [ RETENTION = { ON | OFF } [ , ] ]
    [ ACTIVATION (
      [ STATUS = { ON | OFF } , ]
        PROCEDURE_NAME = <procedure> ,
        MAX_QUEUE_READERS = max_readers ,
        EXECUTE AS { SELF | 'user_name' | OWNER }
        ) [ , ] ]
    [ POISON_MESSAGE_HANDLING (
      [ STATUS = { ON | OFF } ] ) ]
    ]
      [ ON { filegroup | [ DEFAULT ] } ]
[ ; ]

<object> ::=
{
  [ database_name. [ schema_name ] . | schema_name. ]
    queue_name
}

<procedure> ::=
{
  [ database_name. [ schema_name ] . | schema_name. ]
    stored_procedure_name
}
```

For more information, see [CREATE QUEUE (Transact-SQL)]() in the *SQL Server documentation*.

CREATE CONTRACT

The following example specifies the role and what type of messages a service can handle.

```
CREATE CONTRACT contract_name
  [ AUTHORIZATION owner_name ]
    ( { { message_type_name | [ DEFAULT ] }
      SENT BY { INITIATOR | TARGET | ANY }
    } [ ,...n] )
[ ; ]
```

For more information, see [CREATE CONTRACT (Transact-SQL)]() in the *SQL Server documentation*.

CREATE SERVICE

The following example creates a named Service Broker for a specified task or set of tasks.

```
CREATE SERVICE service_name
  [ AUTHORIZATION owner_name ]
  ON QUEUE [ schema_name. ]queue_name
  [ ( contract_name | [DEFAULT][ ,...n ] ) ]
[ ; ]
```

For more information, see [CREATE SERVICE (Transact-SQL)]() in the *SQL Server documentation*.

BEGIN DIALOG CONVERSATION

The following example starts the interaction between Service Brokers.

```
BEGIN DIALOG [ CONVERSATION ] @dialog_handle
  FROM SERVICE initiator_service_name
  TO SERVICE 'target_service_name'
    [ , { 'service_broker_guid' | 'CURRENT DATABASE' }]
  [ ON CONTRACT contract_name ]
  [ WITH
  [ { RELATED_CONVERSATION = related_conversation_handle
    | RELATED_CONVERSATION_GROUP = related_conversation_group_id } ]
  [ [ , ] LIFETIME = dialog_lifetime ]
  [ [ , ] ENCRYPTION = { ON | OFF } ] ]
[ ; ]
```

For more information, see [BEGIN DIALOG CONVERSATION (Transact-SQL)]() in the *SQL Server documentation*.

WAITFOR(RECEIVE TOP(1))

The following example specifies that a code block has to wait until one message is received.

```
[ WAITFOR ( ]
  RECEIVE [ TOP ( n ) ]
  <column_specifier> [ ,...n ]
  FROM <queue>
  [ INTO table_variable ]
  [ WHERE { conversation_handle = conversation_handle
    | conversation_group_id = conversation_group_id } ]
  [ ) ] [ , TIMEOUT timeout ]
[ ; ]

<column_specifier> ::=
{ *
  | { column_name | [ ] expression } [ [ AS ] column_alias ]
  | column_alias = expression
} [ ,...n ]

<queue> ::=
{
  [ database_name . [ schema_name ] . | schema_name . ]
    queue_name
}
```

For more information, see [RECEIVE (Transact-SQL)]() in the *SQL Server documentation*.

You can combine all of the preceding commands to achieve your architecture goals.

For more information, see [Service Broker]() in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) doesn't provide a compatible solution to the SQL Server Service Broker. However, you can use DB Links and AWS Lambda to achieve similar functionality.

You can combine AWS Lambda with AWS SQS to reduce costs and remove some loads from the database into the AWS Lambda and Amazon Simple Queue Service (Amazon SQS). This will be much more efficient. For more information, see Using Lambda with Amazon SQS.

For example, you can create a table in each database and connect each database with a DB link to read the tables and process the data. For more information, see DB Links.

You can also use AWS Lambda to query a table from the database, process the data, and insert it to another database (even another database type). This approach is the best option for moving workloads out of the database to a less expensive instance type.

For even more decoupling and reducing workloads from the database, you can use Amazon SQS with Lambda.

For more information, see Database Mail.

SQL Server Cast and Convert and PostgreSQL CAST and CONVERSION

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	CONVERT is used only to convert between collations. CAST uses different syntax.

SQL Server Usage

The CAST and CONVERT functions are commonly used to convert one data type to another. CAST and CONVERT behave mostly the same and share the same topic in MSDN. They have the following differences:

- CAST is part of the ANSI-SQL specification, but CONVERT isn't.
- CONVERT accepts an optional style parameter used for formatting.

For more information, see Date and Time styles in the *SQL Server documentation*.

Conversion Matrix

For a list of available conversion data types, see Implicit conversions in the *SQL Server documentation*.

Syntax

```
-- CAST Syntax:
CAST ( expression AS data_type [ ( length ) ] )

-- CONVERT Syntax:
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

Examples

The following example casts a `string` to `int` and `int` to `decimal`.

```
SELECT CAST('23.7' AS varchar) AS int, CAST(23.7 AS int) AS decimal;
```

The following example converts `string` to `int` and `int` to `decimal`.

```
SELECT CONVERT(VARCHAR, '23.7') AS int, CONVERT(int, 23.7) AS decimal;
```

For these two preceding examples, the result looks as shown following.

```
int   decimal
23.7  23
```

The following example converts a date with option style input (`109 - mon dd yyyy hh:mi:ss:mmmAM (or PM)`).

```
SELECT CONVERT(nvarchar(30), GETDATE(), 109);

Jul 25 2018 5:20:10.8975085PM
```

For more information, see CAST and CONVERT (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) provides the same CAST function as SQL Server for conversion between data types. It also provides a CONVERSION function, but it isn't equivalent to SQL Server CONVERT. PostgreSQL CONVERSION is used to convert between character set encoding.

CREATE A CAST defines a new cast on how to convert between two data types.

Cast can be EXPLICITLY or IMPLICIT.

The behavior is similar to SQL Server's casting, but in PostgreSQL, you can also create your own casts to change the default behavior. For example, checking if a string is a valid credit card number by creating the CAST with the WITHOUT FUNCTION clause.

CREATE CONVERSION is used to convert between encoding such as UTF8 and LATIN. If CONVERT is currently in use in SQL Server code, rewrite it to use CAST instead.

Note

Not all SQL Server data types are supported on Aurora PostgreSQL, besides changing the CAST or CONVERT commands, you might need to also change the source of the target data type. For more information, see Data Types.

Another way to convert between data types in PostgreSQL will be to use the :: characters. This option is useful and can make your PL/pgSQL code look cleaner and simpler, see the following examples.

Syntax

```
CREATE CAST (source_type AS target_type)
WITH FUNCTION function_name (argument_type [, ...]) [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
WITHOUT FUNCTION [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
WITH INOUT [ AS ASSIGNMENT | AS IMPLICIT ]
```

Examples

The following example converts a numeric value to float.

```
SELECT 23 + 2.0;

or

SELECT CAST ( 23 AS numeric ) + 2.0;
```

The following example converts a date with format input ('mon dd yyyy hh:mi:ss:mmmAM (or PM)').

```
SELECT TO_CHAR(NOW(),'Mon DD YYYY HH:MI:SS:MSAM');

Jul 25 2018 5:20:10.8975085PM
```

The following example uses the `::` characters.

```
SELECT '2.35'::DECIMAL + 4.5 AS results;

results
6.85
```

Summary

Option	SQL Server	Aurora PostgreSQL
Explicit CAST	`SELECT CAST('23.7' AS varchar) AS int`	`SELECT CAST('23.7' AS varchar) AS int`
Explicit CONVERT	`SELECT CONVERT (VARCHAR, '23.7')`	Need to use CAST:
`SELECT CAST('23.7' AS varchar) AS int`	Implicit casting	`SELECT 23 + 2.0 SELECT 23 + 2.0`

Option	SQL Server	Aurora PostgreSQL
Convert to a specific date format: `'mon dd yyyy hh:mi:ss:mmmAM'`	`SELECT CONVERT(nvarchar (30), GETDATE(), 109)`	`SELECT TO_CHAR(NOW(),'Mon DD YYYY HH:MI:SS:MSAM')`

For more information, see CREATE CAST, Type Conversion, and CREATE CONVERSION in the *PostgreSQL documentation*.

SQL Server Common Library Runtime and PostgreSQL PL/Perl

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Migrating CLR objects requires a full code rewrite.

SQL Server Usage

SQL Server provides the capability of implementing .NET objects in the database using the Common Runtime Library (CLR). The CLR enables development of functionality that would be complicated using T-SQL.

The CLR provides robust solutions for string manipulation, date manipulation, and calling external services such as Windows Communication Foundation (WCF) services and web services.

You can create the following objects with the `EXTERNAL NAME` clause:

- Procedures. For more information, see CLR Stored Procedures in the *SQL Server documentation*.
- Functions. For more information, see Create CLR Functions in the *SQL Server documentation*.
- Triggers. For more information, see Create CLR Triggers in the *SQL Server documentation*.
- Types. For more information, see CLR User-Defined Types in the *SQL Server documentation*.
- User-defined aggregate functions. For more information, see CLR User-Defined Aggregates in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) doesn't support .NET code. However, you can create Perl functions. In this case, convert all C# code to PL/pgSQL or PL/Perl.

To use PL/Perl language, install the Perl extension:

```
CREATE EXTENSION plperl;
```

After you install the Perl extension, you can create functions using Perl code. Specify `plperl` in the `LANGUAGE` clause.

You can create the following objects with Perl:

- Functions.
- Void functions or procedures.
- Triggers.
- Event Triggers.
- Values for session level.

Examples

The following example creates a function that returns the greater value of two integers.

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
  if ($_[0] > $_[1]) { return $_[0]; }
  return $_[1];
$$ LANGUAGE plperl;
```

For more information, see PL/Perl — Perl Procedural Language in the *PostgreSQL documentation*.

SQL Server Collations and PostgreSQL Encoding

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Collations	UTF16, NCHAR, and NVARCHAR data types aren't supported.

SQL Server Usage

SQL Server collations define the rules for string management and storage in terms of sorting, case sensitivity, accent sensitivity, and code page mapping. SQL Server supports both ASCII and UCS-2 UNICODE data.

UCS-2 UNICODE data uses a dedicated set of UNICODE data types denoted by the prefix N: Nchar and Nvarchar. Their ASCII counterparts are CHAR and VARCHAR.

Choosing a collation and a character set has significant implications on data storage, logical predicate evaluations, query results, and query performance.

To view all collations supported by SQL Server, use the `fn_helpcollations` function:

```
SELECT * FROM sys.fn_helpcollations()
```

Collations define the actual bitwise binary representation of all string characters and the associated sorting rules. SQL Server supports multiple collations down to the column level. A table may have multiple string columns that use different collations. Collations for non-UNICODE character sets determine the code page number representing the string characters.

UNICODE and non-UNICODE data types in SQL Server aren't compatible. A predicate or data modification that introduces a type conflict is resolved using predefined collation precedence rules. For more information, see Collation Precedence.

Collations define sorting and matching sensitivity for the following string characteristics:

- Case

- Accent
- Kana
- Width
- Variation selector

SQL Server uses a suffix naming convention that appends the option name to the collation name. For example, the collation Azeri_Cyrillic_100_CS_AS_KS_WS_SC, is an Azeri-Cyrillic-100 collation that is case-sensitive, accent-sensitive, kana type-sensitive, width-sensitive, and has supplementary characters.

SQL Server supports three types of collation sets:

- **Windows collations** use the rules defined for collations by the operating system locale where UNICODE and non-UNICODE data use the same comparison algorithms.
- **Binary collations** use the binary bit-wise code for comparison. Therefore, the locale doesn't affect sorting.
- **SQL Server collations** provide backward compatibility with previous SQL Server versions. They aren't compatible with the windows collation rules for non-UNICODE data.

You can define collations at various levels:

- **Server-level collations** determine the collations used for all system databases and is the default for future user databases. While the system databases collation can't be changed, you can specify an alternative collation as part of the `CREATE DATABASE` statement.
- **Database-level collations** inherit the server default unless the `CREATE DATABASE` statement explicitly sets a different collation. This collation is used as a default for all `CREATE TABLE` and `ALTER TABLE` statements.
- **Column-level collations** can be specified as part of the `CREATE TABLE` or `ALTER TABLE` statements to override the default collation setting of your database.
- **Expression-level collations** can be set for individual string expressions using the COLLATE function. For example, `SELECT * FROM MyTable ORDER BY StringColumn COLLATE Latin1_General_CS_AS`.

SQL Server supports UCS-2 UNICODE only.

SQL Server 2019 adds support for UTF-8 for import and export encoding, and as database-level or column-level collation for string data. Support includes PolyBase external tables, and Always Encrypted when not used with Enclaves. For more information, see Collation and Unicode Support.

Syntax

```
CREATE DATABASE <Database Name>
[ ON <File Specifications> ]
COLLATE <Collation>
[ WITH <Database Option List> ];
```

```
CREATE TABLE <Table Name>
(
<Column Name> <String Data Type>
COLLATE <Collation> [ <Column Constraints> ]...
);
```

Examples

The following example creates a database with a default Bengali_100_CS_AI collation.

```
CREATE DATABASE MyBengaliDatabase
ON
( NAME = MyBengaliDatabase_Datafile,
  FILENAME = 'C:\Program Files\Microsoft SQL Server-\MSSQL13.MSSQLSERVER\MSSQL\DATA
\MyBengaliDatabase.mdf', SIZE = 100)
LOG ON
( NAME = MyBengaliDatabase_Logfile,
FILENAME = 'C:\Program Files\Microsoft SQL Server-\MSSQL13.MSSQLSERVER\MSSQL\DATA
\MyBengaliDblog.ldf', SIZE = 25)
COLLATE Bengali_100_CS_AI;
```

The following example creates a table with two different collations.

```
CREATE TABLE MyTable
(
Col1 CHAR(10) COLLATE Hungarian_100_CI_AI_SC NOT NULL PRIMARY KEY,
COL2 VARCHAR(100) COLLATE Sami_Sweden_Finland_100_CS_AS_KS NOT NULL
);
```

For more information, see Collation and Unicode support in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL supports a variety of different character sets, also known as encoding, including support for both single-byte and multi-byte languages. The default character set is specified when initializing a PostgreSQL database cluster with `initdb`. Each individual database created on the PostgreSQL cluster supports individual character sets defined as part of database creation.

> **Note**
>
> For Amazon Relational Database Service (Amazon RDS), starting with PostgreSQL 13, the Windows version now supports obtaining version information for collations or ordering rules from the operating system.

When you query the collation in PostgreSQL running on Windows, prior to version 13 there wasn’t any value to reflect the OS collation version. For example, for PostgreSQL version 11 running on Windows, the result is shown following:

```
CREATE COLLATION german (provider = libc, locale = 'de_DE');

CREATE COLLATION

select oid,collname,collversion from pg_collation
where collprovider='c' and collname='german';

oid    collname  collversion
16394  german
(1 row)

select pg_collation_actual_version (16394);

pg_collation_actual_version
(1 row)
```

For PostgreSQL version 13 running on Windows, the result is shown following:

```
CREATE COLLATION german (provider = libc, locale = 'de_DE');

CREATE COLLATION
```

```
select oid,collname,collversion from pg_collation
where collprovider='c' and collname='german';

oid    collname  collversion
32769  german    1539.5,1539.5
(1 row)

select pg_collation_actual_version (32769);

pg_collation_actual_version
1539.5,1539.5
(1 row)
```

Clients can use all supported character sets. However, some client-side only characters aren't supported for use within the server.

Unlike SQL Server, PostgreSQL doesn't natively support an NVARHCHAR data type and doesn't provide support for UTF-16.

Type	Function	Implementation level
Encoding	Defines the basic rules on how alphanumeric character s are represented in binary format. For example, Unicode encoding.	Database
Locale	A superset that includes `LC_COLLATE` and `LC_CTYPE` among others. For example, `LC_COLLATE` defines how strings are sorted and must be a subset supported by the database encoding.	Table-Column

Examples

The following example creates a database named test01 which uses the Korean EUC_KR Encoding the and the `ko_KR` locale.

```
CREATE DATABASE test01 WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr'
 LC_CTYPE='ko_KR.euckr' TEMPLATE=template0;
```

The following example shows how to view the character sets configured for each database by querying the system catalog.

```
select datname, datcollate, datctype from pg_database;
```

Changing Character Sets or Encoding

In-place modification of the database encoding isn't recommended nor supported. Instead, export all data, create a new database with the new encoding, and import the data.

Export the data using the pg_dump utility.

```
pg_dump mydb1 > mydb1_export.sql
```

Rename or delete a database.

```
ALTER DATABASE mydb1 TO mydb1_backup;
```

Create a new database using the modified encoding.

```
CREATE DATABASE mydb1_new_encoding WITH ENCODING 'UNICODE' TEMPLATE=template0;
```

Import data using the pg_dump file previously created. Verify that you set your client encoding to the encoding of your old database.

```
PGCLIENTENCODING=OLD_DB_ENCODING psql -f mydb1_export.sql mydb1_new_encoding
```

The `client_encoding` parameter overrides the use of `PGCLIENTENCODING`.

Client-Server Character Set Conversions

PostgreSQL supports conversion of character sets between servers and clients for specific character set combinations as described in the `pg_conversion` system catalog.

PostgreSQL includes predefined conversions. For more information, see Available Character Set Conversions in the *PostgreSQL documentation*.

You can create a new conversion using the SQL command `CREATE CONVERSION`.

Examples

The following example creates a conversion from UTF8 to LATIN1 using the custom `myfunc1` function.

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc1;
```

The following example configures the PostgreSQL client character set.

```
Method 1
========
psql \encoding SJIS

Method 2
========
SET CLIENT_ENCODING TO 'value';
```

View the client character set and reset it back to the default value.

```
SHOW client_encoding;

RESET client_encoding;
```

Table Level Collation

PostgreSQL supports specifying the sort order and character classification behavior on a per-column level.

Example

Specify specific collations for individual table columns.

```
CREATE TABLE test1 (col1 text COLLATE "de_DE", col2 text COLLATE "es_ES");
```

Summary

Feature	SQL Server	Aurora PostgreSQL
View database character set	`SELECT collation_name FROM sys.databases;`	`select datname, pg_encoding_to_char(encoding), datcollate, datctype from pg_database;`
Modify the database character set	RECRATE the database	• Export the database. • Drop or rename the database. • Re-create the database with the desired new character set. • Import database data from the exported file into the new database.
Character set granularity	Database	Database
UTF8	Supported	Supported
UTF16	Supported	Not Supported
NCHAR or NVARCHAR data types	Supported	Not Supported

For more information, see Character Set Support in the *PostgreSQL documentation*.

Cursors

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Cursors	Different cursor options.

SQL Server Usage

A *set* is a fundamental concept of the relation data model from which SQL is derived. SQL is a declarative language that operates on whole sets, unlike most procedural languages that operate on individual data elements. A single invocations of an SQL statements can return a whole set or modify millions of rows.

Many developers are accustomed to using procedural or imperative approaches to develop solutions that are difficult to implement using set-based querying techniques. Also, operating on row data sequentially may be a more appropriate approach in certain situations.

Cursors provide an alternative mechanism for operating on result sets. Instead of receiving a table object containing rows of data, applications can use cursors to access the data sequentially, row-by-row. Cursors provide the following capabilities:

- Positioning the cursor at specific rows of the result set using absolute or relative offsets.
- Retrieving a row, or a block of rows, from the current cursor position.
- Modifying data at the current cursor position.
- Isolating data modifications by concurrent transactions that affect the cursor's result.
- T-SQL statements can use cursors in scripts, stored procedures, and triggers.

Syntax

```
DECLARE <Cursor Name>
CURSOR [LOCAL | GLOBAL]
  [FORWARD_ONLY | SCROLL]
  [STATIC | KEYSET | DYNAMIC | FAST_FORWARD]
```

```
  [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]
  [TYPE_WARNING]
  FOR <SELECT statement>
  [ FOR UPDATE [ OF <Column List>]][;]
```

```
FETCH [NEXT | PRIOR | FIRST | LAST | ABSOLUTE <Value> | RELATIVE <Value>]
FROM <Cursor Name> INTO <Variable List>;
```

Examples

Process data in a cursor.

```
DECLARE MyCursor CURSOR FOR
  SELECT *
  FROM Table1 AS T1
    INNER JOIN
    Table2 AS T2
    ON T1.Col1 = T2.Col1;
  OPEN MyCursor;
  DECLARE @VarCursor1 VARCHAR(20);
  FETCH NEXT
    FROM MyCursor INTO @VarCursor1;
  WHILE @@FETCH_STATUS = 0
  BEGIN
    EXEC MyPRocessingProcedure
      @InputParameter = @VarCursor1;
    FETCH NEXT
    FROM product_cursor INTO @VarCursor1;
END

CLOSE MyCursor;
DEALLOCATE MyCursor ;
```

For more information, see SQL Server Cursors and Cursors (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Similar to T-SQL Cursors in SQL Server, PostgreSQL has PL/pgSQL cursors that you can use to iterate business logic on rows read from the database. They can encapsulate the query and read the query results a few rows at a time. All access to cursors in PL/pgSQL is performed through cursor variables, which are always of the `refcursor` data type.

Examples

Declare a Cursor

The following table includes the `DECLARE..CURSOR` options that are Transact-SQL extended syntax have no equivalent in PostgreSQL.

SQL Server option	Use	Comments
`FORWARD_ONLY`	Defining that `FETCH NEXT` is the only supported fetching option.	Using `FOR LOOP` might be a relevant solution for this option.
`STATIC`	Cursor will make a temporary copy of the data.	For small data sets temporary tables can be created and declare a cursor that will select these tables.
`KEYSET`	Determining that membershi p and order of rows in the cursor are fixed.	N/A
`DYNAMIC`	Cursor will reflect all data changes made on the selected rows.	Default for PostgreSQL.
`FAST_FORWARD`	Will use `FORWARD_ONLY` and `READ_ONLY` to optimize performance.	N/A
`SCROLL_LOCKS`	Determine that positioned updates or deletes made by the cursor are guaranteed to succeed.	N/A

SQL Server option	Use	Comments
OPTIMISTIC	Determine that positioned updates or deletes made by the cursor will not succeed if the rows has been updated.	N/A
TYPE_WARNING	Will send warning messages to the client if the cursor is implicitly converted from the requested type.	N/A

Declare a Cursor in PL/pgSQL to be used with any query. The variable c1 is unbounded because it isn't bound to any particular query.

```
DECLARE c1 refcursor;
```

Declare a Cursor in PL/pgSQL with a bounded query.

```
DECLARE c2 CURSOR FOR SELECT * FROM employees;
```

Declare a Cursor with a parametrized bound query:

- The id variable is replaced by an integer parameter value when the cursor is opened.
- When declaring a Cursor with SCROLL specified, the Cursor can scroll backwards.
- If `NO SCROLL` is specified, backward fetches are rejected.

```
DECLARE c3 CURSOR (var1 integer) FOR SELECT * FROM employees where id = var1;
```

Declare a backward-scrolling compatible Cursor using the `SCROLL` option.

- `SCROLL` specifies that rows can be retrieved backwards. `NO SCROLL` specifies that rows can't be retrieved backwards.
- Depending upon the complexity of the run plan for the query, `SCROLL` might create performance issues.

- Backward fetches aren't allowed when the query includes `FOR UPDATE` or `FOR SHARE`.

```
DECLARE c3 SCROLL CURSOR FOR SELECT id, name FROM employees;
```

Open a Cursor

The `OPEN` command is fully compatible between SQL Server and PostgreSQL.

Open a cursor variable that was declared as unbound and specify the query to run.

```
OPEN c1 FOR SELECT * FROM employees WHERE id = emp_id;
```

Open a Cursor variable that was declared as Unbound and specify the query to run as a string expression. This approach provides greater flexibility.

```
OPEN c1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tabname) USING keyvalue;
```

You can insert parameter values into the dynamic command with `format()` and `USING`. For example, the table name is inserted into the query with `format()`. The comparison value for `col1` is inserted with a `USING` parameter.

Open a Cursor that was bound to a query when the cursor was declared and was declared to take arguments.

```
DO $$
DECLARE
  c3 CURSOR (var1 integer) FOR SELECT * FROM employees where id = var1;
BEGIN
  OPEN c3(var1 := 42);
END$$;
```

For the `c3` cursor, supply the argument value expressions.

If the cursor wasn't declared to take arguments, you can specify the arguments outside the cursor.

```
DO $$
DECLARE
  var1 integer;
  c3 CURSOR FOR SELECT * FROM employees where id = var1;
BEGIN
```

```
  var1 := 1;
  OPEN c3;
END$$;
```

Fetch a Cursor

Use the following syntax to fetch a cursor.

```
FETCH [ direction [ FROM | IN ] ] cursor_name
```

The following table shows additional PostgreSQL options as a direction for the FETCH command.

PostgreSQL option	Use
ALL	Get all remaining rows
FORWARD	Same as NEXT
FORWARD	(n) Fetch the next n rows
FORWARD	ALL Same as ALL
BACKWARD	Same as PRIOR
BACKWARD	(n) Fetch the prior n rows
BACKWARD	ALL Fetch all prior rows

The PL/pgSQL `FETCH` command retrieves the next row from the cursor into a variable.

Fetch the values returned from the `c3` cursor into a row variable.

```
DO $$
DECLARE
  c3 CURSOR FOR SELECT * FROM employees;
  rowvar employees%ROWTYPE;
BEGIN
  OPEN c3;
  FETCH c3 INTO rowvar;
END$$;
```

Fetch the values returned from the c3 Cursor into two scalar data types.

```
DO $$
DECLARE
  c3 CURSOR FOR SELECT id, name FROM employees;
  emp_id integer;
  emp_name varchar;
BEGIN
  OPEN c3;
  FETCH FROM c3 INTO emp_id, emp_name;
END$$;
```

PL/pgSQL supports a special direction clause when fetching data from a cursor using the NEXT, PRIOR, FIRST, LAST, ABSOLUTE count, RELATIVE count, FORWARD, or BACKWARD arguments. Omitting direction is equivalent to specifying NEXT. For example, fetch the last row from the cursor into the declared variables.

```
DO $$
DECLARE
  c3 CURSOR FOR SELECT id, name FROM employees;
  emp_id integer;
  emp_name varchar;
BEGIN
  OPEN c3;
  FETCH LAST FROM c3 INTO emp_id, emp_name;
END$$;
```

Summary

Feature	SQL Server	Aurora PostgreSQL
Cursor options	`[FORWARD_ONLY | SCROLL] [STATIC | KEYSET | DYNAMIC | FAST_FORWARD] [READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]`	`[ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT } HOLD ]`

Feature	SQL Server	Aurora PostgreSQL
Updateable cursors	`DECLARE CURSOR... FOR UPDATE`	`DECLARE cur_name CURSOR... FOR UPDATE`
Cursor declaration	`DECLARE CURSOR`	`DECLARE cur_name CURSOR`
Cursor open	`OPEN`	`OPEN`
Cursor fetch	`FETCH NEXT | PRIOR | FIRST | LAST | ABSOLUTE | RELATIVE`	`FETCH [ direction [ FROM | IN ] ] cursor_name` The direction can be empty or one of the following : `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE count`, `RELATIVE count`, `count`, `ALL FORWARD`, `FORWARD count`, `FORWARD ALL`, `BACKWARD`, `BACKWARD count`, `BACKWARD ALL`.
Cursor close	`CLOSE`	`CLOSE`
Cursor deallocate	`DEALLOCATE`	Same effect as CLOSE (not required)
Cursor end condition	`@@FETCH_STATUS` system variable	Not supported

For more information, see FETCH in the *PostgreSQL documentation*.

Date and Time Functions

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Data Types	PostgreSQL is using different function names.

SQL Server Usage

Date and Time Functions are scalar functions that perform operations on temporal or numeric input and return temporal or numeric values.

System date and time values are derived from the operating system of the server on which SQL Server is running.

This section doesn't address time zone considerations and time zone aware functions. For more information about time zone handling, see Data Types.

Syntax and Examples

The following table includes the most commonly used date and time functions.

Function	Purpose	Example	Result	Comments
`GETDATE` and `GETUTCDATE`	Return a datetime value that contains the current local or UTC date and time.	`SELECT GETDATE()`	2018-04-05 15:53:01.380	
`DATEPART`, `DAY`, `MONTH`, and `YEAR`	Return an integer value representing the specified	`SELECT MONTH(GETDATE()),`	4, 2018	

Function	Purpose	Example	Result	Comments
	DATEPART of a specified date.	YEAR(GETDATE())		
DATEDIFF	Returns an integer value of DATEPART boundaries that are crossed between two dates.	SELECT DATEDIFF(DAY, GETDATE() , EOMONTH(GETDATE()))	25	How many days left until end of the month.
DATEADD	Returns a datetime value that is calculated with an offset interval to the specified DATEPART of a date.	SELECT DATEADD(DAY, 25, GETDATE())	2018-04-30 15:55:52.147	
CAST and CONVERT	Converts datetime values to and from string literals and to and from other datetime formats.	SELECT CAST (GETDATE() AS DATE) SELECT CONVERT (VARCHAR(20), GETDATE(), 112)	2018-04-05 20180405	Default date format. Style 112 (ISO) with no separtors.

For more information, see Date and Time functions in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) provides a very rich set of scalar date and time functions; more than SQL Server.

While some of the functions appear to be similar to those in SQL Server, the functionality is significantly different. Take extra care when migrating temporal logic to Aurora PostgreSQL paradigms.

Functions and Definition

PostgreSQL function	Function definition
`AGE`	Subtract from `current_date` .
`CLOCK_TIMESTAMP`	Current date and time.
`CURRENT_DATE`	Current date.
`CURRENT_TIME`	Current time of day.
`CURRENT_TIMESTAMP`	Current date and time (start of current transaction).
`DATE_PART`	Get subfield (equivalent to extract).
`DATE_TRUNC`	Truncate to specified precision.
`EXTRACT`	Get subfield.
`ISFINITE`	Test for finite interval.
`JUSTIFY_DAYS`	Adjust interval so 30-day time periods are represented as months.
`JUSTIFY_HOURS`	Adjust interval so 24-hour time periods are represented as days.

PostgreSQL function	Function definition
`JUSTIFY_INTERVAL`	Adjust interval using `justify_days` and `justify_hours` , with additional sign adjustments.
`LOCALTIME`	Current time of day.
`MAKE_DATE`	Create date from year, month and day fields.
`MAKE_INTERVAL`	Create interval from years, months, weeks, days, hours, minutes and seconds fields.
`MAKE_TIME`	Create time from hour, minute and seconds fields.
`MAKE_TIMESTAMP`	Create timestamp from year, month, day, hour, minute, and seconds fields.
`MAKE_TIMESTAMPTZ`	Create timestamp with time zone from year, month, day, hour, minute, and seconds fields. If the time zone isn't specified, the current time zone is used.
NOW	Current date and time.
`STATEMENT_TIMESTAMP`	Current date and time.
`TIMEOFDAY`	Current date and time (like clock_timestamp, but as a text string).
`TRANSACTION_TIMESTAMP`	Current date and time.
`TO_TIMESTAMP`	Convert Unix epoch (seconds since 1970-01-01 00:00:00+00) to timestamp.

Summary

SQL Server function	Aurora PostgreSQL function
GETDATE, CURRENT_TIMESTAMP	NOW, CURRENT_DATE , CURRENT_TIME , CURRENT_TIMESTAMP
GETUTCDATE	current_timestamp at time zone 'utc'
DAY, MONTH, and YEAR	EXTRACT(DAY/MONTH/YEAR FROM TIMESTAMP timestamp_value)
DATEPART	EXTRACT, DATE_PART
DATEDIFF	DATE_PART
DATEADD	+ INTERVAL 'X days/months/years'
CAST and CONVERT	CAST

For more information, see Date/Time Functions and Operators in the *PostgreSQL documentation*.

String Functions

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Syntax and option differences.

SQL Server Usage

String functions are typically scalar functions that perform an operation on string input and return a string or a numeric value.

Syntax and Examples

The following table includes the most commonly used string functions.

Function	Purpose	Example	Result	Comments
ASCII and UNICODE	Convert an ASCII or UNICODE character to its ASCII or UNICODE code.	SELECT ASCII ('A')	65	Returns a numeric integer value.
CHAR and NCHAR	Convert between ASCII or UNICODE code to a string character.	SELECT CHAR(65)	'A'	Numeric integer value as input.
CHARINDEX and PATINDEX	Find the starting position of one string expression or string pattern within another string expressio n.	SELECT CHARINDEX ('ab','xa bcdy')	2	Returns a numeric integer value.
CONCAT and CONCAT_WS	Combine multiple string input expressio ns into a single string with, or without, a separator character (WS).	SELECT CONCAT ('a','b') , CONCAT_WS (',','a', 'b')	'ab', 'a,b'	
LEFT, RIGHT, and SUBSTRING	Return a partial string from another string	SELECT LEFT ('abs',2) ,SUBSTRIN	'ab', 'bc'	

Function	Purpose	Example	Result	Comments
	expression based on position and length.	`G ('abcd',2 ,2)`		
`LOWER` and `UPPER`	Return a string with all characters in lower or upper case. Use for presentation or to handle case insensitive expressions.	`SELECT LOWER('AB cd')`	'abcd'	
`LTRIM`, `RTRIM`, and `TRIM`	Remove leading and trailing spaces.	`SELECT LTRIM ('abc d ')`	'abc d '	
`STR`	Convert a numeric value to a string.	`SELECT STR(3.141 5927,5,3)`	3.142	Numeric expressions as input.
`REVERSE`	Return a string in reverse order.	`SELECT REVERSE(' abcd')`	'dcba'	
`REPLICATE`	Return a string that consists of zero or more concatenated copies of another string expression.	`SELECT REPLICATE ('abc', 3)`	'abcabcabc'	

Function	Purpose	Example	Result	Comments
REPLACE	Replace all occurrences of a string expression with another.	SELECT REPLACE(' abcd', 'bc', 'xy')	'axyd'	
STRING_SPLIT	Parse a list of values with a separator and return a set of all individual elements.	SELECT * FROM STRING_SPLIT('1,2' ,',') AS X©	12	STRING_SPLIT is a table-valued function.
STRING_AGG	Return a string that consists of concatenated string values in row groups.	SELECT STRING_AGG(C, ',') FROM VALUES(1, 'a'), (1, 'b'), (2,'c') AS X (ID,C) GROUP BY I	1 'ab'	2 'c'

For more information, see String Functions (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Most of SQL Server string functions are supported in PostgreSQL, there are few which aren't:

- UNICODE returns the integer value of the first character as defined by the Unicode standard. If you will use UTF8 input, ASCII can be used to get the same results.
- PATINDEX returns the starting position of the first occurrence of a pattern in a specified expression, or zeros if the pattern isn't found, there is no equivalent function for that but you can create the same function with the same name so it will be fully compatible.

Some of the functions aren't supported but they have an equivalent function in PostgreSQL that you can use to get the same functionality.

Some of the functions such as regular expressions don't exist in SQL Server and may be useful for your application.

Syntax and Examples

The following table includes the most commonly used string functions.

PostgreSQL function	Function definition
`CONCAT`	Concatenate the text representations of all the arguments: `concat('a', 1)` → a1. Also, can use the (||) operators: `select 'a' ||' '|| 'b'` → a b.
`LOWER` or `UPPER`	Returns char, with all letters lowercase or uppercase: `lower ('MR. Smith')` → mr. smith.
`LPAD` or `RPAD`	Returns `expr1`, left or right padded to length n characters with the sequence of character s in `expr2`: `LPAD('Log-1',10,'@')` → @@@@@Log-1.
`REGEXP_REPLACE`	Replace substrings matching a POSIX regular expression: `regexp_replace('John', '[hn].', '1')` → Jo1.
`REGEXP_MATCHES` or `SUBSTRING`	Return all captured substrings resulting from matching a POSIX regular expression against the string: `REGEXP_MATCHES ('http://www.aws.com/products', '(http://[[: alnum:]]+.*/)')`

PostgreSQL function	Function definition
	The result is `{http://www.aws.com/}` . You can use the following example `SUBSTRING ('http://www.aws.com/products', '(http://[[: alnum:]]+.*/)')` The result is `http://www.aws.com/` .
`REPLACE`	Returns char with every occurrence of search string replaced with a replacement string: `replace ('abcdef', 'abc', '123') →` 123def.
`LTRIM` or `RTRIM`	Remove the longest string containing only characters from characters (a space by default) from the start of string: `ltrim('zzzyaws', 'xyz')` → aws.
`SUBSTRING`	Extract substring: `substring ( 'John Smith', 6 ,1)` → S.
`TRIM`	Remove the longest string containing only characters from characters (a space by default) from the start, end, or both ends: `trim (both from 'yxJohnxx', 'xyz') →` John.
`ASCII`	Returns the decimal representation in the database character set of the first character of char: `ascii('a')` → 97.
`LENGTH`	Return the length of char: `length ('John S.')` → 7.

To create the `PATINDEX` function, use the following code snippet. Note the 0 means that the expression doesn't exist so the first position will be 1.

```
CREATE OR REPLACE FUNCTION "patindex"( "pattern" VARCHAR, "expression" VARCHAR )
RETURNS INT AS $BODY$
SELECT COALESCE(STRPOS($2,(
  SELECT(REGEXP_MATCHES($2,'(' ||
  REPLACE( REPLACE(TRIM( $1, '%' ), '%', '.*?' ), '_', '.' )
    || ')','i') )[ 1 ] LIMIT 1)),0);
$BODY$ LANGUAGE 'sql' IMMUTABLE;

SELECT patindex( 'Lo%', 'Long String' );

patindex
1

SELECT patindex( '%rin%', 'Long String' );
patindex
8

SELECT patindex( '%g_S%', 'Long String' );
patindex
4
```

Summary

SQL Server function	Aurora PostgreSQL function
ASCII	ASCII
UNICODE	For UTF8 inputs, you can use only ASCII.
CHAR and NCHAR	CHR
CHARINDEX	POSITION
PATINDEX	See examples
CONCAT and CONCAT_WS	CONCAT and CONCAT_WS
LEFT, RIGHT, and SUBSTRING	LEFT, RIGHT, and SUBSTRING
LOWER and UPPER	LOWER and UPPER

SQL Server function	Aurora PostgreSQL function
`LTRIM`, `RTRIM` and `TRIM`	`LTRIM`, `RTRIM` and `TRIM`
`STR`	`TO_CHAR`
`REVERSE`	`REVERSE`
`REPLICATE`	`LPAD`
`REPLACE`	`REPLACE`
`STRING_SPLIT`	`regexp_split_to_array` or `regexp_split_to_table`
`STRING_AGG`	`CONCAT_WS`

For more information, see String Functions and Operators in the *PostgreSQL documentation*.

Databases and Schemas

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	N/A

SQL Server Usage

Databases and schemas are logical containers for security and access control. Administrators can grant permissions collectively at both the databases and the schema levels. SQL Server instances provide security at three levels: individual objects, schemas (collections of objects), and databases (collections of schemas). For more information, see Data Control Language.

> **Note**
>
> In previous versions of SQL server, the term user was interchangeable with the term schema. For backward compatibility, each database has several built-in security schemas including `guest`, `dbo`, `db_datareaded`, `sys`, `INFORMATION_SCHEMA`, and others. Most likely, you don't need to migrate these schemas.

Each SQL Server instance can host and manage a collection of databases, which consists of SQL Server processes and the Master, Model, TempDB, and MSDB system databases.

The most common SQL Server administrator tasks at the database level are:

- Managing physical files: add, remove, change file growth settings, and re-size files.
- Managing filegroups: partition schemes, object distribution, and read-only protection of tables.
- Managing default options.
- Creating database snapshots.

Unique object identifiers within an instance use three-part identifiers: <Database name>.<Schema name>.<Objectname>.

The recommended way to view database object meta data, including schemas, is to use the ANSI standard information schema views. In most cases, these views are compatible with other ANSI-compliant Relational Database Management Systems (RDBMS).

To view a list of all databases on the server, use the sys.databases table.

Syntax

Simplified syntax for `CREATE DATABASE`.

```
CREATE DATABASE <database name>
[ ON [ PRIMARY ] <file specifications>[,<filegroup>]
[ LOG ON <file specifications>
[ WITH <options specification> ] ;
```

Simplified syntax for `CREATE SCHEMA`.

```
CREATE SCHEMA <schema name> | AUTHORIZATION <owner name>;
```

Examples

The following example adds a file to a database and creates a table using the new file.

```
USE master;
```

```
ALTER DATABASE NewDB
ADD FILEGROUP NewGroup;
```

```
ALTER DATABASE NewDB
ADD FILE (
  NAME = 'NewFile',
  FILENAME = 'D:\NewFile.ndf',
  SIZE = 2 MB
)
TO FILEGROUP NewGroup;
```

```
USE NewDB;
CREATE TABLE NewTable
(
  Col1 INT PRIMARY KEY
)
ON NewGroup;
```

```
SELECT Name
FROM sys.databases
WHERE database_id > 4;
```

The following example creates a table within a new schema and database.

```
USE master
CREATE DATABASE NewDB;

USE NewDB;
CREATE SCHEMA NewSchema;

CREATE TABLE NewSchema.NewTable
```

```
(
  NewColumn VARCHAR(20) NOT NULL PRIMARY KEY
);
```

This example uses default settings for the new database and schema.

For more information, see sys.databases (Transact-SQL), CREATE SCHEMA (Transact-SQL), and CREATE DATABASE in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) supports both the `CREATE SCHEMA` and `CREATE DATABASE` statements.

As with SQL Server, Aurora PostgreSQL does have the concept of an instance hosting multiple databases, which in turn contain multiple schemas. Objects in Aurora PostgreSQL are referenced as a three-part name: `<database>.<schema>.<object>`.

A schema is essentially a namespace that contains named objects.

When database is created, it is cloned from a template.

Syntax

Syntax for `CREATE DATABASE`.

```
CREATE DATABASE name
  [ [ WITH ] [ OWNER [=] user_name ]
    [ TEMPLATE [=] template ]
    [ ENCODING [=] encoding ]
    [ LC_COLLATE [=] lc_collate ]
    [ LC_CTYPE [=] lc_ctype ]
    [ TABLESPACE [=] tablespace_name ]
    [ ALLOW_CONNECTIONS [=] allowconn ]
    [ CONNECTION LIMIT [=] connlimit ]
    [ IS_TEMPLATE [=] istemplate ] ]
```

Syntax for `CREATE SCHEMA`.

```
CREATE SCHEMA schema_name [ AUTHORIZATION role_specification ] [ schema_element
 [ ... ] ]
```

```
CREATE SCHEMA AUTHORIZATION role_specification [ schema_element [ ... ] ]
CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION role_specification ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION role_specification

where role_specification can be:
user_name | CURRENT_USER | SESSION_USER
```

Migration Considerations

Unlike SQL Server, Aurora PostgreSQL doesn't support the USE command to specify the default database or schema for missing object qualifiers. To use a different database, use a new connection, obtain the required permissions, and refer to the object using the database name.

For applications using a single database and multiple schemas, the migration path is the same and requires fewer rewrites because two-part names are already being used.

Query the `postgres.pg_catalog.pg_database` table to view databases in Aurora PostgreSQL.

```
SELECT datname, datcollate, datistemplate, datallowconn
FROM postgres.pg_catalog.pg_database;

datname    datcollate   datistemplate  datallowconn
template0  en_US.UTF-8  true           false
rdsadmin   en_US.UTF-8  false          true
template1  en_US.UTF-8  true           true
postgres   en_US.UTF-8  false          true
```

Examples

The following example creates a new database.

```
CREATE DATABASE NewDatabase;
```

The following example creates a schema for user testing.

```
CREATE SCHEMA AUTHORIZATION joe;
```

The following example creates a schema, a table and a view.

```
CREATE SCHEMA world_flights
```

```
CREATE TABLE flights (flight_id VARCHAR(10), departure DATE, airport VARCHAR(30))
CREATE VIEW us_flights AS
  SELECT flight_id, departure FROM flights WHERE airport='United States';
```

For more information, see CREATE DATABASE and CREATE SCHEMA in the *PostgreSQL documentation*.

SQL Server Dynamic SQL and PostgreSQL EXECUTE and PREPARE

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Different paradigm and syntax require rewriting the application.

SQL Server Usage

Dynamic SQL is a feature that helps minimize hard-coded SQL. The SQL engine optimizes code, which leads to less hard parses.

Developers can use dynamic SQL to construct and run SQL queries at run time as a string, using some logic in SQL to construct varying query strings, without having to pre-construct them during development.

There are two options for running dynamic SQL: use the `EXECUTE` command or the `sp_executesql` function.

EXECUTE Command

Use this option to run a command string within a T-SQL block, procedure, or function. You can also use the `EXECUTE` command with linked servers. You can define metadata for the result set using the `WITH RESULT SETS` options.

For parameters, use either the value or `@parameter_name=value`.

Note

Make sure that you validate the structure of the string command before running it with the EXECUTE command.

Syntax

The following example shows the SQL Server syntax that runs a stored procedure or function.

```
[ { EXEC | EXECUTE } ]
  {
    [ @return_status = ]
    { module_name [ ;number ] | @module_name_var }
      [ [ @parameter = ] { value
        | @variable [ OUTPUT ]
        | [ DEFAULT ]
        }
      ]
    [ ,...n ]
    [ WITH <execute_option> [ ,...n ] ]
  }
[;]
```

The following example shows the SQL Server syntax that runs a character string.

```
{ EXEC | EXECUTE }
  ( { @string_variable | [ N ]'tsql_string' } [ + ...n ] )
  [ AS { LOGIN | USER } = ' name ' ]
[;]
```

The following example shows the SQL Server syntax that runs a pass-through command against a linked server.

```
{ EXEC | EXECUTE }
  ( { @string_variable | [ N ] 'command_string [ ? ]' } [ + ...n ]
    [ { , { value | @variable [ OUTPUT ] } } [ ...n ] ]
  )
  [ AS { LOGIN | USER } = ' name ' ]
  [ AT linked_server_name ]
[;]
```

```
<execute_option>::=
{
  RECOMPILE
  | { RESULT SETS UNDEFINED }
  | { RESULT SETS NONE }
  | { RESULT SETS ( <result_sets_definition> [,...n ] ) }
  }

<result_sets_definition> ::=
{
  (
    { column_name
    data_type
    [ COLLATE collation_name ]
    [ NULL | NOT NULL ] }
    [,...n ]
    )
  | AS OBJECT
    [ db_name . [ schema_name ] . | schema_name . ]
    {table_name | view_name | table_valued_function_name }
  | AS TYPE [ schema_name.]table_type_name
  | AS FOR XML
}
```

Example

The following example shows how to use EXECUTE to run a `tsql_string` function with a variable.

```
DECLARE @scm_name sysname;
DECLARE @tbl_name sysname;
EXECUTE ('DROP TABLE ' + @scm_name + '.' + @tbl_name + ';');
```

The following example shows how to use `EXECUTE AS USER` to switch context to another user.

```
DECLARE @scm_name sysname;
DECLARE @tbl_name sysname;
EXECUTE ('DROP TABLE ' + @scm_name + '.' + @tbl_name + ';') AS USER = 'SchemasAdmin';
```

The following example shows how to use EXECUTE with a result set.

```
EXEC GetMaxSalByDeptID 23
```

```
WITH RESULT SETS
(
  ([Salary] int NOT NULL)
);
```

sp_executesql System Stored Procedure

This option runs a T-SQL command or block that you can run several times and build dynamically. You can also use this option with embedded parameters.

Syntax

The following example shows the `sp_executesql` syntax for SQL Server, Azure SQL Database, Azure SQL Data Warehouse, and Parallel Data Warehouse.

```
sp_executesql [ @stmt = ] statement
[
  { , [ @params = ] N'@parameter_name data_type [ OUT | OUTPUT ][ ,...n ]' }
    { , [ @param1 = ] 'value1' [ ,...n ] }
]
```

Example

The following example shows how to use `sp_executesql` to run a SELECT statement.

```
EXECUTE sp_executesql
  N'SELECT * FROM HR.Employees
  WHERE DeptID = @DID',
  N'@DID int',
  @DID = 23;
```

For more information, see sp_executesql (Transact-SQL) and EXECUTE (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

The PostgreSQL EXECUTE command prepares and runs commands dynamically. The EXECUTE command can also run DDL statements and retrieve data using SQL commands. Similar to SQL Server, you can use the PostgreSQL EXECUTE command with bind variables.

Converting SQL Server dynamic SQL to PostgreSQL requires significant efforts.

Examples

The following example runs a SQL SELECT query with the table name as a dynamic variable using bind variables. This query returns the number of employees under a manager with a specific ID.

```
DO $$DECLARE
Tabname varchar(30) := 'employees';
num integer := 1;
cnt integer;
BEGIN
EXECUTE format('SELECT count(*) FROM %I WHERE manager = $1', tabname)
INTO cnt USING num;
RAISE NOTICE 'Count is % int table %', cnt, tabname;
END$$;
;
```

The following example runs a DML command; first with no variables and then with variables.

```
DO $$DECLARE
BEGIN
EXECUTE 'INSERT INTO numbers (a) VALUES (1)';
EXECUTE format('INSERT INTO numbers (a) VALUES (%s)', 42);
END$$;
;
```

Note

`%s` formats the argument value as a simple string. A null value is treated as an empty string. `%I` treats the argument value as an SQL identifier and double-quotes it if necessary. It is an error for the value to be null.

The following example runs a DDL command.

```
DO $$DECLARE
BEGIN
EXECUTE 'CREATE TABLE numbers (num integer)';
END$$;
;
```

For more information, see String Functions and Operators in the *PostgreSQL documentation*.

Prepare

Using a PREPARE statement can improve performance of reusable SQL statements.

The PREPARE command can receive a SELECT, INSERT, UPDATE, DELETE, or VALUES statement and parse it with a user-specified qualifying name so you can use the EXECUTE command later without the need to re-parse the SQL statement for each run.

- When using PREPARE to create a prepared statement, it will be viable for the scope of the current session.
- If a DDL command is run on a database object referenced by the prepared SQL statement, the next EXECUTE command requires a hard parse of the SQL statement.

Example

Use PREPARE and EXECUTE commands together. The SQL command is prepared with a user-specified qualifying name. You can run the SQL command several times8 without the need for re-parsing.

```
PREPARE numplan (int, text, bool) AS
INSERT INTO numbers VALUES($1, $2, $3);
EXECUTE numplan(100, 'New number 100', 't');
EXECUTE numplan(101, 'New number 101', 't');
EXECUTE numplan(102, 'New number 102', 'f');
EXECUTE numplan(103, 'New number 103', 't');
```

Summary

Functionality	SQL Server dynamic SQL	PostgreSQL EXECUTE and PREPARE
Run SQL with results and bind variables	`DECLARE @sal int;` `EXECUTE getSalary @sal OUTPUT;`	`EXECUTE format('select salary from employees WHERE %I = $1', col_name)`

Functionality	SQL Server dynamic SQL	PostgreSQL EXECUTE and PREPARE
		`INTO amount USING col_val;`
Run DML with variables and bind variables	`DECLARE @amount int DECLARE @col_val int DECLARE @col_name carchar(70) DECLARE @sqlCommand varchar(1000) SET @sqlCommand = 'UPDATE employees SET salary=salary' + @amount + ' WHERE ' + @col_name + '=' + @col_val EXECUTE (@sqlCommand)`	`EXECUTE format('UPDATE employees SET salary = salary + $1 WHERE %I = $2', col_name) USING amount, col_val;`
Run DDL	`EXECUTE ('CREATE TABLE link_emp (idemp1 integer, idemp2 integer);');`	`EXECUTE 'CREATE TABLE link_emp (idemp1 integer, idemp2 integer)';`
Run anonymous block	`BEGIN ... END; DO $ $DECLARE`	`BEGIN ... END$$;`

For more information, see Basic Statements in the *PostgreSQL documentation*.

Transactions

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Transaction Isolation	Nested transactions aren't supported and syntax differences for initializing a transaction.

SQL Server Usage

A transaction is a unit of work performed on a database and typically represents a change in the database. Transactions serve the following purposes:

- Provide units of work that enable recovery from logical or physical system failures while keeping the database in a consistent state.
- Provide units of work that enable recovery from failures while keeping a database in a consistent state when a logical or physical system failure occurs.
- Provide isolation between users and programs accessing a database concurrently.

Transactions are an all-or-nothing unit of work. Each transactional unit of work must either complete, or it must rollback all data changes. Also, transactions must be isolated from other transactions. The results of the view of data for each transaction must conform to the defined database isolation level.

Database transactions must comply with ACID properties.

- **Atomic** — Transactions are all or nothing. If any part of the transaction fails, the entire transaction fails and the database remains unchanged.

  There are exceptions to this rule. For example, some constraint violations, for ANSI definitions, should not cause a transaction rollback.
- **Consistent** — All transactions must bring the database from one valid state to another valid state. Data must be valid according to all defined rules, constraints, triggers, and so on.

- **Isolation** — Concurrent run of transactions must result in a system state that would occur if transactions were run sequentially.

  There are several exceptions to this rule based on the lenience of the required isolation level.
- **Durable** — After a transaction commits successfully and is acknowledged to the client, the engine must guarantee that its changes are persisted in the event of power loss, system crashes, or any other errors.

  By default, SQL Server uses the auto commit or implicit transactions mode set to `ON`. Every statement is treated as a transaction on its own unless a transaction was explicitly defined. This behavior is different than other engines like Oracle where, by default, every DML requires an explicit `COMMIT` statement to be persisted.

Syntax

The following examples show the simplified syntax for the commands defining transaction boundaries.

Define the beginning of a transaction.

```
BEGIN TRAN | TRANSACTION [<transaction name>]
```

Commit work and the end of a transaction.

```
COMMIT WORK | [ TRAN | TRANSACTION [<transaction name>]]
```

Rollback work at the end of a transaction.

```
ROLLBACK WORK | [ TRAN | TRANSACTION [<transaction name>]]
```

SQL Server supports the standard ANSI isolation levels defined by the ANSI/ISO SQL standard (SQL92).

Each level provides a different approach for managing the concurrent run of transactions. The main purpose of a transaction isolation level is to manage the visibility of changed data as seen by other running transactions. Additionally, when concurrent transactions access the same data, the level of transaction isolation affects the way they interact with each other.

- **Read uncommitted** — A current transaction can see uncommitted data from other transactions. If a transaction performs a rollback, all data is restored to its previous state.
- **Read committed** — A transaction only sees data changes that were committed. Therefore, dirty reads aren't possible. However, after issuing a commit, it would be visible to the current transaction while it's still in a running state.
- **Repeatable read** — A transaction sees data changes made by the other transactions only after both transactions issue a commit or are rolled back.
- **Serializable** — This isolation level is the strictest because it doesn't permit transaction overwrites of another transaction's actions. Concurrent run of a set of serializable transactions is guaranteed to produce the same effect as running them sequentially in the same order.

The main difference between isolation levels is the phenomena they prevent from appearing. The three preventable phenomena are:

- **Dirty reads** — A transaction can read data written by another transaction but not yet committed.
- **Non-repeatable or fuzzy reads** — When reading the same data several times, a transaction can find the data has been modified by another transaction that has just committed. The same query ran twice can return different values for the same rows.
- **Phantom or ghost reads** — Similar to a non-repeatable read, but it is related to new data created by another transaction. The same query ran twice can return different numbers of records.

The following table summarizes the four ANSI/ISO SQL standard (SQL92) isolation levels and indicates which phenomena are allowed or disallowed.

Transaction isolation level	Dirty reads	Non-repeatable reads	Phantom reads
Read uncommitted	Allowed	Allowed	Allowed
Read committed	Disallowed	Allowed	Allowed
Repeatable read	Disallowed	Disallowed	Allowed

Transaction isolation level	Dirty reads	Non-repeatable reads	Phantom reads
Serializable	Disallowed	Disallowed	Disallowed

There are two common implementations for transaction isolation:

- **Pessimistic isolation or locking** — Resources accessed by a transaction are locked for the duration of the transaction. Depending on the operation, resource, and transaction isolation level, other transactions can see changes made by the locking transaction, or they must wait for it to complete. With this mechanism, there is only one copy of the data for all transactions, which minimizes memory and disk resource consumption at the expense of transaction lock waits.
- **Optimistic isolation (MVCC)** — Every transaction owns a set of the versions of the resources (typically rows) that it accessed. In this mode, transactions don't have to wait for one another at the expense of increased memory and disk utilization. In this isolation mechanism, there is a chance that conflicts will arise when transactions attempt to commit. In case of a conflict, the application needs to be able to handle the rollback, and attempt a retry.

SQL Server implements both mechanisms. You can use them concurrently.

For optimistic isolation, SQL Server introduced two additional isolation levels: read-committed snapshot and snapshot.

Set the transaction isolation level using SET command. It affects the current run scope only.

```
SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ |
 SNAPSHOT | SERIALIZABLE }
```

Examples

The following example runs two DML statements within a serializable transaction.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION;
INSERT INTO Table1
VALUES (1, 'A');
UPDATE Table2
```

```
  SET Column1 = 'Done'
WHERE KeyColumn = 1;
COMMIT TRANSACTION;
```

For more information, see Transaction Isolation Levels (ODBC) and SET TRANSACTION ISOLATION LEVEL (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

As with SQL Server, the same ANSI/ISO SQL (SQL92) isolation levels apply to PostgreSQL, but with several similarities and some differences.

Transaction isolation level	Dirty reads	Non-repeatable reads	Phantom reads
Read uncommitted	Permitted but not implemented	Permitted	Permitted
Read committed	Not permitted	Permitted	Permitted
Repeatable read	Not permitted	Not permitted	Permitted but not implemented
Serializable	Not permitted	Not permitted	Not permitted

PostgreSQL technically supports the use of any of the four transaction isolation levels, but only three can practically be used. The Read-Uncommitted isolation level serves as read-committed.

The way the repeatable-read isolation-level is implemented doesn't allow for phantom reads, which is similar to the Serializable isolation-level. The primary difference between repeatable-read and serializable is that serializable guarantees that the result of concurrent transactions are precisely the same as if they were run serially, which isn't always true for repeatable-reads.

Starting with PostgreSQL 12, you can add the `AND CHAIN` option to `COMMIT` or `ROLLBACK` commands to immediately start another transaction with the same parameters as preceding transaction.

Multiversion Concurrency Control

In PostgreSQL, the multiversion concurrency control (MVCC) mechanism allows transactions to work with a consistent snapshot of data ignoring changes made by other transactions that have not yet committed or rolled back. Each transaction sees a snapshot of accessed data accurate to its run start time regardless of what other transactions are doing concurrently.

Isolation Levels

PostgreSQL supports the read-committed, repeatable reads, and serializable isolation levels. Read-committed is the default isolation level.

- **Read-committed** — The default PostgreSQL transaction isolation level. It prevents sessions from seeing data from concurrent transactions until it is committed. Dirty reads aren't permitted.
- **Repeatable read** — Queries can only see rows committed before the first query or DML statement was run in the transaction.
- **Serializable** — Provides the strictest transaction isolation level. The Serializable isolation level assures that the result of the concurrent transactions will be the same as if they run serially. This isn't always the case for the repeatable read isolation level.

Setting Isolation Levels in Aurora PostgreSQL

You can configure isolation levels at several levels.

- Session level.
- Transaction level.
- Instance level using Aurora parameter groups.

Syntax

```
SET TRANSACTION transaction_mode [...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [...]

where transaction_mode is one of:

ISOLATION LEVEL {
```

```
  SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED
}
READ WRITE | READ ONLY [ NOT ] DEFERRABLE
```

Examples

The following example configures the isolation level for a specific transaction.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

The following example configures the isolation level for a specific session.

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Use the following example to view the current isolation level.

```
SELECT CURRENT_SETTING('TRANSACTION_ISOLATION'); -- Session
SHOW DEFAULT_TRANSACTION_ISOLATION; -- Instance
```

You can use parameter groups to modify instance-level parameters for Aurora PostgreSQL. For example, you can alter the `default_transaction_isolation` parameter using the AWS Console or the AWS CLI. For more information, see Working with parameter groups.

Comparison table of relevant database features related to transactions

Database feature	SQL Server	PostgreSQL
AutoCommit	Off	Autocommit is turned off by default, however, some client tools like psql and more are setting this to ON by default. Check your client tool defaults or run the following

Database feature	SQL Server	PostgreSQL
		command to check current configuration in psql: `\echo :AUTOCOMMIT` .
MVCC	Yes	Yes
Default isolation level	Read-committed	Read-committed
Supported isolation levels	REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED, SERIALIZA BLE	Repeatable reads, serializable, read-only
Configure session isolation levels	Yes	Yes
Configure transaction isolation levels	Yes	Yes

Read-Committed Isolation Level

TX1	TX2	Comment

`SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100;`
 	
`employee_id  salary`
`100          24000.00`
	`select employee_id, salary from EMPLOYEES where employee_id=100;`
	
	`employee_id  salary`
	`100          24000.00`
		Same results returned from both sessions.

`begin;`
`UPDATE employees`
`SET salary=27000`
`WHERE employee_id=100;`
	`begin;`
	`set transaction isolation level`
	`read committed;`
		TX1 starts a transaction and performs an update. TX2 starts a transaction with read-committed isolation level.

TX1	TX2	Comment

`SELECT employee_id,`
` salary`
`FROM EMPLOYEES`
`WHERE employee_id=100;`
 	
`employee_id  salary`
`100          27000.00`
	`SELECT employee_id,`
	` salary`
	`FROM EMPLOYEES`
	`WHERE employee_id=100;`
	
	`employee_id  salary`
	`100          24000.00`
		TX1 will see the modified results while TX2 sees the original data.

	`UPDATE employees`
	`SET salary=29000`
	`WHERE employee_id=100;`
		Waits because TX2 is blocked by TX1.

`Commit;`		TX1 issues a commit, and the lock is released.
	`Commit;`	TX2 issues a commit.

`SELECT employee_id,`
` salary`
`FROM EMPLOYEES`
`WHERE employee_id=100;`
 	
`employee_id  salary`
`100          29000.00`
	`SELECT employee_id,`
	` salary`
	`FROM EMPLOYEES`
	`WHERE employee_id=100;`
	
	`employee_id  salary`
	`100          29000.00`
		Both queries return the updated value.

Serializable Isolation Level

TX1	TX2	Comment

`SELECT employee_id,`
` salary`
`FROM EMPLOYEES`
`WHERE employee_id=100;`
	`SELECT employee_id,`
	` salary`
	`FROM EMPLOYEES`
	`WHERE employee_id=100;`
		Same results returned from both sessions.

TX1	TX2	Comment
`employee_id salary` `100 24000.00`	`employee_id salary` `100 24000.00`	

`begin;`
`UPDATE employees`
`SET salary=27000`
`WHERE employee_id=100;`
	`begin;`
	`set transaction`
	` isolation level`
	` serializable;`
		TX1 starts a transaction and performs an update. TX2 starts a transaction with isolation level of serializable.

`SELECT employee_id,`
` salary`
`FROM EMPLOYEES`
`WHERE employee_id=100;`
 	
`employee_id salary`
`100 27000.00`
	`SELECT employee_id,`
	` salary`
	`FROM EMPLOYEES`
	`WHERE employee_id=100;`
	
	`employee_id salary`
	`100 24000.00`
		TX1 will see the modified results while TX2 sees the original data.

	`update employees`
	`set salary=29000`
	`where employee_id=100;`
		Waits because TX2 is blocked by TX1.

`Commit;`		TX1 issues a commit, and the lock is released.
	ERROR: couldn’t serialize access due to concurrent update.	TX2 received an error message.
	`Commit;` `ROLLBACK`	TX2 trying to issue a commit but receives a rollback message, the transaction failed due to the serializable isolation level.

TX1	TX2	Comment

`SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100;`
 	
`employee_id salary`
`100 27000.00`
	`SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100;`
	
	`employee_id salary`
	`100 27000.00`
		Both queries return the value updated according to TX1.

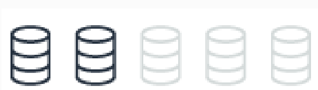
Summary

Transaction property	SQL Server	Aurora PostgreSQL
Default isolation level	`READ COMMITTED`	`READ COMMITTED`
Initialize transaction syntax	`BEGIN TRAN` or `TRANSACTION`	`SET TRANSACTION`
Default isolation mechanism	Pessimistic lock based	Lock based for writes, consistent read for selects
Commit transaction	`COMMIT [WORK|TRAN|TRANSACTION]`	`COMMIT [ WORK | TRANSACTION ]`
Rollback transaction	`ROLLBACK [WORK |[ TRAN | TRANSACTION]`	`ROLLBACK [ WORK | TRANSACTION ]`
Set autocommit off/on	`SET IMPLICIT_TRANSACTIONS OFF | ON`	`SET AUTOCOMMIT { = | TO } { ON | OFF }`
ANSI isolation	`REPEATABLE READ | READ COMMITTED |`	`REPEATABLE READ | READ COMMITTED |`

Transaction property	SQL Server	Aurora PostgreSQL
	`READ UNCOMMITTED | SERIALIZABLE`	`READ UNCOMMITTED | SERIALIZABLE`
MVCC	`SNAPSHOT and READ COMMITTED SNAPSHOT`	`READ COMMITTED SNAPSHOT`
Nested transactions	Supported, view level with `@@trancount`	Not Supported

For more information, see Transactions, Transaction Isolation, and SET TRANSACTION in the *PostgreSQL documentation*.

SQL Server Synonyms and PostgreSQL Views, Types, and Functions

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	PostgreSQL doesn't support synonyms. There is an available workaround.

SQL Server Usage

Synonyms are database objects that serve as alternative identifiers for other database objects. The referenced database object is called the base object and may reside in the same database, another database on the same instance, or a remote server.

Synonyms provide an abstraction layer to isolate client application code from changes to the name or location of the base object.

In SQL Server, synonyms are often used to simplify the use of four-part identifiers when accessing remote instances.

For example, the table A resides on the server A, and the client application accesses it directly. For scale out reasons, the table A needs to be moved to the server B to offload resource consumption on the server A. Without synonyms, the client application code must be rewritten to access the server B. Instead, you can create a synonym called Table A and it will transparently redirect the calling application to the server B without any code changes.

You can create synonyms for the following objects:

- Assembly (CLR) stored procedures, table-valued functions, scalar functions, and aggregate functions.
- Replication-filter-procedures.
- Extended stored procedures.
- SQL scalar functions, table-valued functions, inline-tabled-valued functions, views, and stored procedures.
- User-defined tables including local and global temporary tables.

Syntax

```
CREATE SYNONYM [ <Synonym Schema> ] . <Synonym Name>
FOR [ <Server Name> ] . [ <Database Name> ] . [ Schema Name> ] . <Object Name>
```

Examples

The following example creates a synonym for a local object in a separate database.

```
CREATE TABLE DB1.Schema1.MyTable
(
KeyColumn INT IDENTITY PRIMARY KEY,
DataColumn VARCHAR(20) NOT NULL
);

USE DB2;
CREATE SYNONYM Schema2.MyTable
FOR DB1.Schema1.MyTable
```

The following example creates a synonym for a remote object.

```
-- On ServerA
CREATE TABLE DB1.Schema1.MyTable
(
KeyColumn INT IDENTITY PRIMARY KEY,
DataColumn VARCHAR(20) NOT NULL
);

-- On Server B
USE DB2;
CREATE SYNONYM Schema2.MyTable
FOR ServerA.DB1.Schema1.MyTable;
```

The example preceding assumes a linked server named ServerA exists on Server B that points to Server A.

For more information, see [CREATE SYNONYM (Transact-SQL)]() in the *SQL Server documentation*.

PostgreSQL Usage

SQL Server synonyms are often used to give another name for an object. PostgreSQL doesn't provide a feature comparable to SQL Server Synonyms. However, you can achieve similar functionality by using a few PostgreSQL objects.

AWS SCT converts different source databases into one target database. Each source database becomes a schema in the new target database. AWS SCT adds the name of the source schemas as a prefix to the name of the target database schema. If you migrate several databases as part of one migration project, then you can avoid using synonyms because all converted objects are in the same database.

This lack of functionality in PostgreSQL adds a manual dimension to the migration process of SQL Server synonyms. Make sure that your database user has privileges on the base object and the relevant PostgreSQL options.

Examples

To create a synonym of a table in PostgreSQL, use views.

The first step is to create a table that will be used as the base object, and on top of it, a view that will be used as synonym.

```
CREATE TABLE target_db_name.DB1_Schema1.MyTable
(
  KeyColumn NUMERIC PRIMARY KEY,
  DataColumn VARCHAR(20) NOT NULL
);

CREATE VIEW target_db_name.DB2_Schema2.MyTable_Syn
AS SELECT * FROM target_db_name.DB1_Schema1.MyTable
```

For more information, see Views.

To create a synonym of a user-defined type in PostgreSQL, another user-defined type should be used to wrap the source type.

The first step is to create the user-defined type that will be used as the base object, and on top of it, a user-defined type that will be used as the synonym.

```
CREATE TYPE DB1.Schema1.MyType AS (
ID NUMERIC,
name CHARACTER VARYING(100));

CREATE TYPE DB2.Schema2.MyType_Syn AS (
udt DB1.Schema1.MyT);
```

For more information, see User-Defined Types.

To create a synonym for a function in PostgreSQL, another function should be used to wrap the source type.

As before, the first step is to create the function that will be used as the base object. And then, on top of it, create a function that will be used as the synonym.

```
CREATE OR REPLACE FUNCTION DB1.Schema1.MyFunc (P_NUM NUMERIC)
RETURNS numeric AS $$
begin
  RETURN P_NUM * 2;
END; $$
LANGUAGE PLPGSQL;

CREATE OR REPLACE FUNCTION DB2.Schema2.MyFunc_Syn (P_NUM NUMERIC)
RETURNS numeric AS $$
```

```
begin
  RETURN DB1.Schema1.MyFunc(P_NUM);
END; $$
LANGUAGE PLPGSQL;
```

For more information, see User-Defined Functions.

DELETE and UPDATE FROM

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	PostgreSQL doesn't support `DELETE … FROM from_list`. Rewrite to use subqueries.

SQL Server Usage

SQL Server supports an extension to the ANSI standard that allows using an additional `FROM` clause in `UPDATE` and `DELETE` statements.

You can use this additional `FROM` clause to limit the number of modified rows by joining the table being updated, or deleted from, to one or more other tables. This functionality is similar to using a WHERE clause with a derived table sub-query. For `UPDATE`, you can use this syntax to set multiple column values simultaneously without repeating the sub-query for every column.

However, these statements can introduce logical inconsistencies if a row in an updated table is matched to more than one row in a joined table. The current implementation chooses an arbitrary value from the set of potential values and is non-deterministic.

Syntax

```
UPDATE <Table Name>
SET <Column Name> = <Expression> ,...
```

```
FROM <Table Source>
WHERE <Filter Predicate>;
```

```
DELETE FROM <Table Name>
FROM <Table Source>
WHERE <Filter Predicate>;
```

Examples

The following example deletes customers with no orders.

```
CREATE TABLE Customers
(
  Customer VARCHAR(20) PRIMARY KEY
);
```

```
INSERT INTO Customers VALUES
('John'),
('Jim'),
('Jack')
```

```
CREATE TABLE Orders
(
  OrderID INT NOT NULL PRIMARY KEY,
  Customer VARCHAR(20) NOT NULL,
  OrderDate DATE NOT NULL
);
```

```
INSERT INTO Orders (OrderID, Customer, OrderDate) VALUES
(1, 'Jim', '20180401'),
(2, 'Jack', '20180402');
```

```
DELETE FROM Customers
FROM Customers AS C
  LEFT OUTER JOIN
  Orders AS O
  ON O.Customer = C.Customer
WHERE O.OrderID IS NULL;
```

```
SELECT *
FROM Customers;
```

```
Customer
Jim
Jack
```

The following example updates multiple columns in Orders based on the values in OrderCorrections.

```
CREATE TABLE OrderCorrections
(
  OrderID INT NOT NULL PRIMARY KEY,
  Customer VARCHAR(20) NOT NULL,
  OrderDate DATE NOT NULL
);
```

```
INSERT INTO OrderCorrections
VALUES (1, 'Jack', '20180324');
```

```
UPDATE O
SET Customer = OC.Customer,
  OrderDate = OC.OrderDate
FROM Orders AS O
  INNER JOIN
  OrderCorrections AS OC
  ON O.OrderID = OC.OrderID;
```

```
SELECT *
FROM Orders;
```

```
Customer  OrderDate
Jack      2018-03-24
Jack      2018-04-02
```

For more information, see UPDATE (Transact-SQL), DELETE (Transact-SQL), and FROM clause plus JOIN, APPLY, PIVOT (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Aurora PostgreSQL doesn't support the `DELETE..FROM` syntax, but it support the `UPDATE FROM` syntax.

Syntax

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
  SET { column_name = { expression | DEFAULT } |
    ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) |
    ( column_name [, ...] ) = ( sub-SELECT )
  } [, ...]
  [ FROM from_list ]
  [ WHERE condition | WHERE CURRENT OF cursor_name ]
  [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

Migration Considerations

You can rewrite the `DELETE` statements as subqueries. Place the subqueries in the WHERE clause. This workaround is simple and, in most cases, easier to read and understand.

Examples

The following example deletes customers with no orders.

```
CREATE TABLE Customers
(
  Customer VARCHAR(20) PRIMARY KEY
);

INSERT INTO Customers
VALUES
('John'),
('Jim'),
('Jack')

CREATE TABLE Orders
(
  OrderID INT NOT NULL PRIMARY KEY,
  Customer VARCHAR(20) NOT NULL,
  OrderDate DATE NOT NULL
```

```
);

INSERT INTO Orders (OrderID, Customer, OrderDate)
VALUES
(1, 'Jim', '20180401'),
(2, 'Jack', '20180402');

DELETE FROM Customers
WHERE Customer NOT IN (
  SELECT Customer
  FROM Orders
);

SELECT * FROM Customers;

Customer
Jim
Jack
```

The following example updates multiple columns in Orders based on the values in OrderCorrections

```
CREATE TABLE OrderCorrections
(
  OrderID INT NOT NULL PRIMARY KEY,
  Customer VARCHAR(20) NOT NULL,
  OrderDate DATE NOT NULL
);

INSERT INTO OrderCorrections
VALUES (1, 'Jack', '20180324');

UPDATE orders
SET Customer = OC.Customer,
  OrderDate = OC.OrderDate
FROM Orders AS O
  INNER JOIN
  OrderCorrections AS OC
  ON O.OrderID = OC.OrderID;

SELECT *
FROM Orders;
```

```
Customer  OrderDate
Jack      2018-03-24
Jack      2018-04-02
```

Summary

The following table identifies similarities, differences, and key migration considerations.

Feature	SQL Server	Aurora PostgreSQL
Join as part of `DELETE`	`DELETE FROM … FROM`	Not available. Rewrite to use WHERE clause with a sub-query.
Join as part of `UPDATE`	`UPDATE … FROM`	`UPDATE … FROM`

For more information, see DELETE and UPDATE in the *PostgreSQL documentation*.

Stored Procedures

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Stored Procedures	Syntax and option differences.

SQL Server Usage

Stored procedures are encapsulated, persisted code modules that you can run using the `EXECUTE` T-SQL statement. They may have multiple input (`IN`) and output (`OUT`) parameters. Table-valued user-defined types can be used as input parameters. `IN` is the default direction for parameters, but `OUT` must be explicitly specified. You can specify parameters as both `IN` and `OUT`.

SQL Server allows you to run stored procedures in any security context using the `EXECUTE AS` option. You can explicitly recompile them for every run using the `RECOMPILE` option. You can encrypt them in the database using the ENCRYPTION option to prevent unauthorized access to the source code.

SQL Server provides a unique feature that allows you to use a stored procedure as an input to an INSERT statement. When using this feature, only the first row in the data set returned by the stored procedure is evaluated.

Syntax

```
CREATE [ OR ALTER ] { PROC | PROCEDURE } <Procedure Name>
[<Parameter List>
[ WITH [ ENCRYPTION ]|[ RECOMPILE ]|[ EXECUTE AS ...]]
AS {
[ BEGIN ]
<SQL Code Body>
[ END ] }[;]
```

Examples

Create and run a stored procedure

The following example creates a simple parameterized stored procedure to validate the basic format of an email.

```
CREATE PROCEDURE ValidateEmail
@Email VARCHAR(128), @IsValid BIT = 0 OUT
AS
BEGIN
IF @Email LIKE N'%@%' SET @IsValid = 1
ELSE SET @IsValid = 0
RETURN @IsValid
END;
```

The following example runs this stored procedure.

```
DECLARE @IsValid BIT
EXECUTE [ValidateEmail]
@Email = 'X@y.com', @IsValid = @IsValid OUT;
SELECT @IsValid;

-- Returns 1
```

```
EXECUTE [ValidateEmail]
@Email = 'Xy.com', @IsValid = @IsValid OUT;
SELECT @IsValid;

-- Returns 0
```

The following example creates a stored procedure that uses RETURN to pass an error value to the application.

```
CREATE PROCEDURE ProcessImportBatch
@BatchID INT
AS
BEGIN
BEGIN TRY
EXECUTE Step1 @BatchID
EXECUTE Step2 @BatchID
EXECUTE Step3 @BatchID
END TRY
BEGIN CATCH
IF ERROR_NUMBER() = 235
RETURN -1 -- indicate special condition
ELSE
THROW -- handle error normally
END CATCH
END
```

Using a table-valued input parameter

The following example creates and populates an OrderItems table.

```
CREATE TABLE OrderItems(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
```

```
(3, 'M8 Washer', 200),
(3, 'M6 Washer', 100);
```

The following example creates a table-valued type for the `OrderItem` table-valued parameter.

```
CREATE TYPE OrderItems
AS TABLE
(
  OrderID INT NOT NULL,
  Item VARCHAR(20) NOT NULL,
  Quantity SMALLINT NOT NULL,
  PRIMARY KEY(OrderID, Item)
);
```

The following example creates a procedure to process order items.

```
CREATE PROCEDURE InsertOrderItems
@OrderItems AS OrderItems READONLY
AS
BEGIN
  INSERT INTO OrderItems(OrderID, Item, Quantity)
  SELECT OrderID,
    Item,
    Quantity
  FROM @OrderItems
END;
```

The following example populates the table-valued variable and passes the data set to the stored procedure.

```
DECLARE @OrderItems AS OrderItems;

INSERT INTO @OrderItems ([OrderID], [Item], [Quantity])
VALUES
(1, 'M8 Bolt', 100),
(1, 'M8 Nut', 100),
(1, M8 Washer, 200);

EXECUTE [InsertOrderItems]
@OrderItems = @OrderItems;
```

```
(3 rows affected)
   Item       Quantity
1  M8 Bolt    100
2  M8 Nut     100
3  M8 Washer  200
```

INSERT… EXEC Syntax

```
INSERT INTO <MyTable>
EXECUTE <MyStoredProcedure>;
```

For more information, see CREATE PROCEDURE (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL version 10 provides support for both stored procedures and stored functions using the `CREATE FUNCTION` statement. To emphasize, only the `CREATE FUNCTION` is supported by the procedural statements used by PostgreSQL version 10. The `CREATE PROCEDURE` statement isn't supported.

PL/pgSQL is the main database programming language used for migrating from SQL Server T-SQL code. PostgreSQL supports these additional programming languages, also available in Amazon Aurora PostgreSQL:

- PL/pgSQL
- PL/Tcl
- PL/Perl

Use the `show.rds.extensions` command to view all available Amazon Aurora extensions.

PostgreSQL Create Function Privileges

To create a function, make sure that a user has the `USAGE` privilege on the language. When you create a function, you can specify a language parameter as shown in the following examples.

Examples

The following example creates a new FUNC_ALG function.

```
CREATE OR REPLACE FUNCTION FUNC_ALG(P_NUM NUMERIC)
RETURNS NUMERIC
AS $$
BEGIN
  RETURN P_NUM * 2;
END; $$
LANGUAGE PLPGSQL;
```

The `CREATE OR REPLACE` statement creates a new function or replaces an existing function with these limitations:

- You can't change the function name or argument types.
- The statement doesn't allow changing the existing function return type.
- The user must own the function to replace it.
- The `P_NUM` INPUT parameter is implemented similar to SQL Server T-SQL INPUT parameter.
- The double dollar signs alleviate the need to use single-quoted string escape elements. With the double dollar sign, there is no need to use escape characters in the code when using single quotation marks. The double dollar sign appears after the keyword `AS` and after the function keyword `END`.
- Use the `LANGUAGE PLPGSQL` parameter to specify the language for the created function.

The following example creates a function with PostgreSQL PL/pgSQL.

```
CREATE OR REPLACE FUNCTION EMP_SAL_RAISE
(IN P_EMP_ID DOUBLE PRECISION, IN SAL_RAISE DOUBLE PRECISION)
RETURNS VOID
AS $$
DECLARE
V_EMP_CURRENT_SAL DOUBLE PRECISION;
BEGIN
SELECT SALARY INTO STRICT V_EMP_CURRENT_SAL
FROM EMPLOYEES WHERE EMPLOYEE_ID = P_EMP_ID;

UPDATE EMPLOYEES SET SALARY = V_EMP_CURRENT_SAL + SAL_RAISE WHERE EMPLOYEE_ID =
 P_EMP_ID;

RAISE DEBUG USING MESSAGE := CONCAT_WS('', 'NEW SALARY FOR EMPLOYEE ID: ', P_EMP_ID, '
IS ', (V_EMP_CURRENT_SAL + SAL_RAISE));
```

```
EXCEPTION
WHEN OTHERS THEN
RAISE USING ERRCODE := '20001', MESSAGE := CONCAT_WS('', 'AN ERROR WAS ENCOUNTERED -',
 SQLSTATE, ' -ERROR-', SQLERRM);
END; $$
LANGUAGE PLPGSQL;

select emp_sal_raise(200, 1000);
```

In the preceding example, you can replace the RAISE command with RETURN to inform the application that an error occurred.

The following example creates a function with PostgreSQL PL/pgSQL.

```
CREATE OR REPLACE FUNCTION EMP_PERIOD_OF_SERVICE_YEAR (IN P_EMP_ID DOUBLE PRECISION)
RETURNS DOUBLE PRECISION
AS $$
DECLARE
V_PERIOD_OF_SERVICE_YEARS DOUBLE PRECISION;
BEGIN
SELECT
EXTRACT (YEAR FROM NOW()) - EXTRACT (YEAR FROM (HIRE_DATE))
INTO STRICT V_PERIOD_OF_SERVICE_YEARS
FROM EMPLOYEES
WHERE EMPLOYEE_ID = P_EMP_ID;
RETURN V_PERIOD_OF_SERVICE_YEARS;
END; $$
LANGUAGE PLPGSQL;

SELECT EMPLOYEE_ID,FIRST_NAME, EMP_PERIOD_OF_SERVICE_YEAR(EMPLOYEE_ID) AS
PERIOD_OF_SERVICE_YEAR
FROM EMPLOYEES;
```

There is a new behavior in PostgreSQL version 10 for a set-returning function, used by LATERAL FROM clause.

PostgreSQL version 9.6 and lower

```
CREATE TABLE emps (id int, manager int);
INSERT INTO tab VALUES (23, 24), (52, 23), (21, 65);
SELECT x, generate_series(1,5) AS g FROM tab;
```

```
id  g
23  1
23  2
23  3
23  4
23  5
52  1
52  2
52  3
52  4
52  5
21  1
21  2
21  3
21  4
21  5
```

PostgreSQL version 10 and higher

```
SELECT id, g FROM emps, LATERAL generate_series(1,5) AS g;

id  g
23  1
23  2
23  3
23  4
23  5
52  1
52  2
52  3
52  4
52  5
21  1
21  2
21  3
21  4
21  5
```

In the preceding example, you can put the set-return function on the outside of the nested loop join because it has no actual lateral dependency on `emps` table.

Summary

The following table summarizes the differences between stored procedures in SQL Server and PostgreSQL.

Feature	SQL Server	Aurora PostgreSQL	Workaround
General CREATE syntax differences	`CREATE PROC|PROCEDURE <Procedure Name> @Parameter1 <Type>, ...n AS <Body>`	`CREATE [ OR REPLACE] FUNCTION <Function Name> (Parameter1 <Type>, ...n) AS $$ <body>`	Rewrite stored procedure creation scripts to use `FUNCTION` instead of `PROC` or `PROCEDURE` . Rewrite stored procedure creation scripts to omit the AS $$ pattern. Rewrite stored procedure parameters to not use the `@` symbol in parameter names. Add parentheses around the parameter declaration.
Security context	`{ EXEC | EXECUTE } AS { CALLER | SELF | OWNER | 'user_name' }`	`SECURITY INVOKER | SECURITY DEFINER`	For stored procedures that use an explicit user name, rewrite the code from `EXECUTE AS user` to `SECURITY DEFINER` and recreate the functions with this user.

Feature	SQL Server	Aurora PostgreSQL	Workaround
			For stored procedures that use the `CALLER` option, rewrite the code to include `SECURITY INVOKER`. For stored procedure s that use the `SELF` option, rewrite the code to `SECURITY DEFINER`.
Encryption	Use the `WITH ENCRYPTION` option.	Not supported in Aurora PostgreSQL.	
Parameter direction	`IN` and `OUT|OUTPUT`, by default `OUT` can be used as `IN` as well.	`IN`, `OUT`, `INOUT`, or `VARIADIC`	Although the functionality of these parameters is the same for SQL Server and PostgreSQL, rewrite the code for syntax compliance. Use `OUT` instead of `OUTPUT`. Use `INOUT` instead of `OUT` for bidirectional parameters.
Recompile	Use the `WITH RECOMPILE` option.	Not supported in Aurora PostgreSQL.	

Feature	SQL Server	Aurora PostgreSQL	Workaround
Table-valued parameters	Use declared table type user-defined parameters.	Use declared table type user-defined parameters.	
Additional restrictions	Use `BULK INSERT` to load data from text file.	Not supported in Aurora PostgreSQL.	

For more information, see CREATE FUNCTION, PL/pgSQL — SQL Procedural Language, Procedural Languages, and Query Language (SQL) Functions in the *PostgreSQL documentation*.

Error Handling

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Different paradigm and syntax will require rewrite of error handling code.

SQL Server Usage

SQL Server error handling capabilities have significantly improved throughout the years. However, previous features are retained for backward compatibility.

Before SQL Server 2008, only very basic error handling features were available. `RAISERROR` was the primary statement used for error handling.

Starting from SQL Server 2008, the extensive .NET-like error handling capabilities were added. They included `TRY…CATCH` blocks, `THROW` statements, the `FORMATMESSAGE` function, and a set of system functions that return metadata for the current error condition.

TRY...CATCH Blocks

TRY...CATCH blocks implement error handling similar to Microsoft Visual C# and Microsoft Visual C ++. TRY ... END TRY statement blocks can contain T-SQL statements.

If an error is raised by any of the statements within the TRY ... END TRY block, the run stops and is moved to the nearest set of statements that are bounded by a CATCH ... END CATCH block.

```
BEGIN TRY
<Set of SQL Statements>
END TRY
BEGIN CATCH
<Set of SQL Error Handling Statements>
END CATCH
```

THROW

The THROW statement raises an exception and transfers run of the TRY ... END TRY block of statements to the associated CATCH ... END CATCH block of statements.

Throw accepts either constant literals or variables for all parameters.

```
THROW [Error Number>, <Error Message>, < Error State>] [;]
```

Examples

The following example uses TRY...CATCH error blocks to handle key violations.

```
CREATE TABLE ErrorTest (Col1 INT NOT NULL PRIMARY KEY);
```

```
BEGIN TRY
  BEGIN TRANSACTION
    INSERT INTO ErrorTest(Col1) VALUES(1);
    INSERT INTO ErrorTest(Col1) VALUES(2);
    INSERT INTO ErrorTest(Col1) VALUES(1);
  COMMIT TRANSACTION;
END TRY
BEGIN CATCH
  THROW; -- Throw with no parameters = RETHROW
```

```
END CATCH;
```

```
(1 row affected)
(1 row affected)
(0 rows affected)
Msg 2627, Level 14, State 1, Line 7
Violation of PRIMARY KEY constraint 'PK__ErrorTes__A259EE54D8676973'.
Can't insert duplicate key in object 'dbo.ErrorTest'. The duplicate key value is (1).
```

Note

Contrary to what many SQL developers believe, the values 1 and 2 are indeed inserted into `ErrorTestTable` in the preceding example. This behavior is in accordance with ANSI specifications stating that a constraint violation should not roll back an entire transaction.

The following example uses `THROW` with variables.

```
BEGIN TRY
BEGIN TRANSACTION
INSERT INTO ErrorTest(Col1) VALUES(1);
INSERT INTO ErrorTest(Col1) VALUES(2);
INSERT INTO ErrorTest(Col1) VALUES(1);
COMMIT TRANSACTION;
END TRY
BEGIN CATCH
DECLARE @CustomMessage VARCHAR(1000),
  @CustomError INT,
  @CustomState INT;
SET @CustomMessage = 'My Custom Text ' + ERROR_MESSAGE();
SET @CustomError = 54321;
SET @CustomState = 1;
THROW @CustomError, @CustomMessage, @CustomState;
END CATCH;
```

```
(0 rows affected)
Msg 54321, Level 16, State 1, Line 19
My Custom Text Violation of PRIMARY KEY constraint 'PK__ErrorTes__A259EE545CBDBB9A'.
Can't insert duplicate key in object 'dbo.ErrorTest'. The duplicate key value is (1).
```

RAISERROR

The RAISERROR statement is used to explicitly raise an error message, similar to THROW. It causes an error state for the run session and forwards run to either the calling scope or, if the error occurred within a TRY … END TRY block, to the associated CATCH … END CATCH block. RAISERROR can reference a user-defined message stored in the sys.messages system table or can be used with dynamic message text.

The key differences between THROW and RAISERROR are:

- Message IDs passed to RAISERROR must exist in the sys.messages system table. The error number parameter passed to THROW doesn't.
- RAISERROR message text may contain printf formatting styles. The message text of THROW may not.
- RAISERROR uses the severity parameter for the error returned. For THROW, severity is always 16.

```
RAISERROR (<Message ID>|<Message Text>, <Message Severity>, <Message State>
[WITH option [<Option List>]])
```

The following example raises a custom error.

```
RAISERROR (N'This is a custom error message with severity 10 and state 1.', 10, 1)
```

FORMATMESSAGE

FORMATMESSAGE returns a sting message consisting of an existing error message in the sys.messages system table, or from a text string, using the optional parameter list replacements. The FORMATMESSAGE statement is similar to the RAISERROR statement.

```
FORMATMESSAGE (<Message Number> | <Message String>, <Parameter List>)
```

Error State Functions

SQL Server provides the following error state functions:

- ERROR_LINE
- ERROR_MESSAGE

- ERROR_NUMBER
- ERROR_PROCEDURE
- ERROR_SEVERITY
- ERROR_STATE
- @@ERROR

The following example uses error state functions within a CATCH block.

```
CREATE TABLE ErrorTest (Col1 INT NOT NULL PRIMARY KEY);
```

```
BEGIN TRY;
  BEGIN TRANSACTION;
    INSERT INTO ErrorTest(Col1) VALUES(1);
    INSERT INTO ErrorTest(Col1) VALUES(2);
    INSERT INTO ErrorTest(Col1) VALUES(1);
  COMMIT TRANSACTION;
END TRY
BEGIN CATCH
  SELECT ERROR_LINE(),
    ERROR_MESSAGE(),
    ERROR_NUMBER(),
    ERROR_PROCEDURE(),
    ERROR_SEVERITY(),
    ERROR_STATE(),
    @@Error;
THROW;
END CATCH;
```

```
6
Violation of PRIMARY KEY constraint 'PK__ErrorTes__A259EE543C8912D8'. Can't insert
duplicate key in object 'dbo.ErrorTest'. The duplicate key value is (1).
2627
NULL
14
1
2627
```

```
(1 row affected)
```

```
(1 row affected)
(0 rows affected)
(1 row affected)
Msg 2627, Level 14, State 1, Line 25
Violation of PRIMARY KEY constraint 'PK__ErrorTes__A259EE543C8912D8'. Can't insert
duplicate key in object 'dbo.ErrorTest'. The duplicate key value is (1).
```

For more information, see RAISERROR (Transact-SQL), TRY...CATCH (Transact-SQL), and THROW (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) doesn't provide native replacement for SQL Server error handling features and options, but it has many comparable options.

To trap the errors, use the `BEGIN.. EXCEPTION.. END`. By default, any error raised in a PL/pgSQL function block stops running and the surrounding transaction. You can trap and recover from errors using a `BEGIN` block with an `EXCEPTION` clause. The syntax is an extension to the normal syntax for a `BEGIN` block.

Syntax

```
[ <<label>> ]
[ DECLARE
  declarations ]
BEGIN
  statements
EXCEPTION
  WHEN condition [ OR condition ... ] THEN
    handler_statements
  [ WHEN condition [ OR condition ... ] THEN
    handler_statements
  ... ]
END;
```

For the preceding example, condition is related to the error or the code. For example:

- `WHEN interval_field_overflow THEN…`
- `WHEN SQLSTATE '22015' THEN…`

For all error codes, see PostgreSQL Error Codes in the *PostgreSQL documentation*.

Throw errors

You can use the PostgreSQL `RAISE` statement to throw errors. You can combine `RAISE` with several levels of severity including:

Severity	Usage
DEBUG1..DEBUG5	Provides successively more detailed information for use by developers.
INFO	Provides information implicitly requested by the user.
NOTICE	Provides information that might be helpful to users.
WARNING	Provides warnings of likely problems.
ERROR	Reports an error that caused the current command to abort.
LOG	Reports information of interest to administrators. For example, checkpoint activity.
FATAL	Reports an error that caused the current session to abort.
PANIC	Reports an error that caused all database sessions to abort.

Examples

The following example uses `RAISE DEBUG`, where `DEBUG` is the configurable severity level.

```
SET CLIENT_MIN_MESSAGES = 'debug';

DO $$
```

```
BEGIN
RAISE DEBUG USING MESSAGE := 'hello world';
END $$;

DEBUG: hello world
DO
```

The following example uses the `client_min_messages` parameter to control the level of messages sent to the client. The default is `NOTICE`. Use the `log_min_messages` parameter to control which message levels are written to the server log. The default is `WARNING`.

```
SET CLIENT_MIN_MESSAGES = 'debug';
```

The following example uses `EXCEPTION..WHEN…THEN` inside `BEGIN` and `END` block to handle dividing by zero violations.

```
CREATE TABLE ErrorTest (Col1 INT NOT NULL PRIMARY KEY);
```

```
INSERT INTO employee values ('John',10);
BEGIN
  SELECT 5/0;
EXCEPTION
  WHEN division_by_zero THEN
    RAISE NOTICE 'caught division_by_zero';
  return 0;
END;
```

Summary

The following table identifies similarities, differences, and key migration considerations.

SQL Server error handling feature	Aurora PostgreSQL equivalent

`TRY … END TRY` and `CATCH … END CATCH` blocks	`Inner`
	`BEGIN`
	`...`
	`EXCEPTION WHEN ... THEN`
	`END`

SQL Server error handling feature	Aurora PostgreSQL equivalent
THROW and RAISERROR	RAISE
FORMATMESSAGE	RAISE [level] 'format' or ASSERT
Error state functions	GET STACKED DIAGNOSTICS
Proprietary error messages in sys.messages system table	RAISE

For more information, see Error Handling, Errors and Messages, and When to Log in the *PostgreSQL documentation*.

SQL Server Flow Control and PostgreSQL Control Structures

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Flow Control	PostgreSQL doesn't support GOTO and WAITFOR TIME.

SQL Server Usage

Although SQL Server is a mostly declarative language, it does support flow control commands, which provide run time dynamic changes in script run paths.

Before SQL/PSM was introduced in SQL:1999, the ANSI standard didn't include flow control constructs. Therefore, there are significant syntax differences among RDBMS engines.

SQL Server provides the following flow control keywords.

- BEGIN… END — Define boundaries for a block of commands that are run together.
- RETURN — Exit a server code module (stored procedure, function, and so on) and return control to the calling scope. You can use RETURN <value> to return an INT value to the calling scope.

* `BREAK` — Exit `WHILE` loop run.
* `THROW` — Raise errors and potentially return control to the calling stack.
* `CONTINUE` — Restart a `WHILE` loop.
* `TRY… CATCH` — Error handling. For more information, see Error Handling.
* `GOTO label` — Moves the run point to the location of the specified label.
* `WAITFOR` — Delay.
* `IF… ELSE` — Conditional flow control.
* `WHILE <condition>` — Continue looping while <condition> returns TRUE.

> **Note**
>
> WHILE loops are commonly used with cursors and use the system variable @@FETCH_STATUS to determine when to exit. For more information, see Cursors.

Examples

The following example demonstrates a solution for running different processes based on the number of items in an order.

Create and populate an OrderItems table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200);
```

Declare a cursor for looping through all OrderItems and calculating the total quantity for each order.

```
DECLARE OrderItemCursor CURSOR FAST_FORWARD
FOR
SELECT OrderID,
  SUM(Quantity) AS NumItems
FROM OrderItems
GROUP BY OrderID
ORDER BY OrderID;

DECLARE @OrderID INT, @NumItems INT;

-- Instantiate the cursor and loop through all orders.
OPEN OrderItemCursor;

FETCH NEXT FROM OrderItemCursor
INTO @OrderID, @NumItems

WHILE @@Fetch_Status = 0
BEGIN;
  IF @NumItems > 100
    PRINT 'EXECUTING LogLargeOrder - '
    + CAST(@OrderID AS VARCHAR(5))
    + ' ' + CAST(@NumItems AS VARCHAR(5));
  ELSE
    PRINT 'EXECUTING LogSmallOrder - '
    + CAST(@OrderID AS VARCHAR(5))
    + ' ' + CAST(@NumItems AS VARCHAR(5));

FETCH NEXT FROM OrderItemCursor
INTO @OrderID, @NumItems;
END;

-- Close and deallocate the cursor.
CLOSE OrderItemCursor;
DEALLOCATE OrderItemCursor;
```

For the preceding example, the result looks as shown following.

```
EXECUTING LogSmallOrder - 1 100
EXECUTING LogSmallOrder - 2 100
```

```
EXECUTING LogLargeOrder - 3 200
```

For more information, see Control-of-Flow in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) provides the following flow control constructs:

- `BEGIN… END` — Define boundaries for a block of commands that are run together.
- `CASE` — Run a set of commands based on a predicate (not to be confused with CASE expressions).
- `IF… ELSE` — Perform conditional flow control.
- `ITERATE` — Restart a `LOOP` or `WHILE` statement.
- `LEAVE` — Exit a server code module such as stored procedure, function, and so on and return control to the calling scope.
- `LOOP` — Loop indefinitely.
- `REPEAT… UNTIL` — Loop until the predicate is true.
- `RETURN` — Terminate the run of the current scope and return to the calling scope.
- `WHILE` — Continue looping while the condition returns TRUE.

Examples

The following example demonstrates a solution for running different logic based on the number of items in an order. It provides the same functionality as the example for SQL Server flow control. However, unlike the SQL Server example ran as a batch script, Aurora PostgreSQL variables can only be used in stored routines such as procedures and functions.

Create and populate an OrderItems table.

```
CREATE TABLE OrderItems
(
  OrderID INT NOT NULL,
  Item VARCHAR(20) NOT NULL,
  Quantity SMALLINT NOT NULL,
  PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200);
```

Create a procedure to declare a cursor and loop through the order items.

```
CREATE OR REPLACE FUNCTION P()
  RETURNS numeric
  LANGUAGE plpgsql
AS $function$
DECLARE
  done int default false;
  var_OrderID int;
  var_NumItems int;
  OrderItemCursor CURSOR FOR SELECT OrderID, SUM(Quantity) AS NumItems
  FROM OrderItems
  GROUP BY OrderID
  ORDER BY OrderID;

  BEGIN
    OPEN OrderItemCursor;
    LOOP
      fetch from OrderItemCursor INTO var_OrderID, var_NumItems;
    EXIT WHEN NOT FOUND;
    IF var_NumItems > 100 THEN
      RAISE NOTICE 'EXECUTING LogLargeOrder - %s',var_OrderID;
      RAISE NOTICE 'Num Items: %s', var_NumItems;
    ELSE
      RAISE NOTICE 'EXECUTING LogSmallOrder - %s',var_OrderID;
      RAISE NOTICE 'Num Items: %s', var_NumItems;
    END IF;
    END LOOP;
done = TRUE;
CLOSE OrderItemCursor;
END; $function$
```

Summary

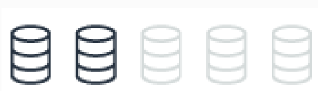
While there are some syntax differences between SQL Server and Aurora PostgreSQL flow control statements, most rewrites should be straightforward. The following table summarizes the differences and identifies how to modify T-SQL code to support similar functionality in Aurora PostgreSQL PL/pgSQL.

Command	SQL Server	Aurora PostgreSQL
`BEGIN…END`	Define command block boundaries.	Define command block boundaries.
`RETURN`	Exit the current scope and return to caller. Supported for both scripts and stored code such as procedures and functions.	Exit a stored function and return to caller.
`BREAK`	Exit `WHILE` loop run	`EXIT WHEN`
`THROW`	Raise errors and potentially return control to the calling stack.	Raise errors and potentially return control to the calling stack.
`TRY…CATCH`	Error handling.	Error handling. For more information, see Error Handling.
`GOTO`	Move run to a specified label	Consider rewriting the flow logic using either CASE statements or nested stored procedures. You can use nested stored procedures to circumvent this limitation by separating code sections and encapsulating them in sub-procedures. Use `IF <condition> EXEC <stored procedure>` instead of `GOTO`.

Command	SQL Server	Aurora PostgreSQL
`WAITFOR`	Delay	`pg_sleep`. For more information, see [Date/Time Functions and Operators] in the *PostgreSQL documentation*.
`IF… ELSE`	Conditional flow control.	Conditional flow control.
`WHILE`	Continue running while condition is true.	Continue running while condition is true.

For more information, see [Control Structures] in the *PostgreSQL documentation*.

Full-Text Search

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		[Full-Text Search]	Different paradigm and syntax require rewriting the application.

SQL Server Usage

SQL Server supports an optional framework for running full-text search queries against character-based data in SQL Server tables using an integrated, in-process full-text engine and a `fdhost.exe` filter daemon host process.

To run full-text queries, create a full-text catalog. This catalog in turn may contain one or more full-text indexes. A full-text index is comprised of one or more textual columns of a table.

Full-text queries perform smart linguistic searches against full-text indexes by identifying words and phrases based on specific language rules. The searches can be for simple words, complex phrases, or multiple forms of a word or a phrase. They can return ranking scores for matches or hits.

Full-Text Indexes

You can create a full-text index on one of more columns of a table or view for any of the following data types:

- `CHAR` — Fixed size ASCII string column data type.
- `VARCHAR` — Variable size ASCII string column data type.
- `NCHAR` — Fixed size UNICODE string column data type.
- `NVARCHAR` — Variable size UNICODE string column data type.
- `TEXT` — ASCII BLOB string column data type. This data type is deprecated.
- `NTEXT` — UNICODE BLOB string column data type. This data type is deprecated.
- `IMAGE` — Binary BLOB data type. This data type is deprecated.
- `XML` — XML structured BLOB data type.
- `VARBINARY(MAX)` — Binary BLOB data type.
- `FILESTREAM` — File-based storage data type.

For more information, see Data Types.

You can use the `CREATE FULLTEXT INDEX` statement to create full-text indexes. A full-text index may contain up to 1024 columns from a single table or view.

When you create full-text indexes on `BINARY` type columns, you can store documents such as Microsoft Word as a binary stream and parse them correctly by the full-text engine.

Full-Text Catalogs

Full-text indexes are contained within full-text catalog objects. A full-text catalog is a logical container for one or more full-text indexes, You can use a full-text catalog to collectively administer them as a group for tasks such as back-up, restore, refresh content, and so on.

You can use the `CREATE FULLTEXT CATALOG` statement to create full-text catalogs. A full-text catalog may contain zero or more full-text indexes and is limited in scope to a single database.

Full-Text Queries

After you create and populate a full-text catalog and index, you can run full-text queries against these indexes to query for:

- Simple term match for one or more words or phrases.
- Prefix term match for words that begin with a set of characters.
- Generational term match for inflectional forms of a word.
- Proximity term match for words or phrases that are close to another word or phrase.
- Thesaurus search for synonymous forms of a word.
- Weighted term match for finding words or phrases with weighted proximity values.

Full-text queries are integrated into T-SQL and use the following predicates and functions:

- `CONTAINS` predicate.
- `FREETEXT` predicate.
- `CONTAINSTABLE` table-valued function.
- `FREETEXTTABLE` table-valued function.

> **Note**
> Don't confuse full-text functionality with the `LIKE` predicate, which is used for pattern matching only.

Updating Full-Text Indexes

By default, full-text indexes are automatically updated when the underlying data is modified, similar to a normal B-tree or columnstore index. However, large changes to the underlying data may inflict a performance impact for the full-text indexes update because it is a resource intensive operation. In these cases, you can disable the automatic update of the catalog and update it manually, or on a schedule, to keep the catalog up to date with the underlying tables.

Note

You can monitor the status of the full-text catalog by using the `FULLTEXTCATALOGPROPERTY (<Full-text Catalog Name>, 'Populatestatus')` function.

Examples

The following example creates a product review table.

```
CREATE TABLE ProductReviews
(
  ReviewID INT NOT NULL
  IDENTITY(1,1),
  CONSTRAINT PK_ProductReviews PRIMARY KEY(ReviewID),
  ProductID INT NOT NULL
  /*REFERENCES Products(ProductID)*/,
  ReviewText VARCHAR(4000) NOT NULL,
  ReviewDate DATE NOT NULL,
  UserID INT NOT NULL
  /*REFERENCES Users(UserID)*/
);
```

```
INSERT INTO ProductReviews
( ProductID, ReviewText, ReviewDate, UserID)
VALUES
(1, 'This is a review for product 1, it is excellent and works as expected','20180701',
 2),
(1, 'This is a review for product 1, it isn't that great and failed after two
 days','20180702', 2),
(2, 'This is a review for product 3, it has exceeded my expectations. A+++','20180710',
 2);
```

The following example creates a full-text catalog for product reviews.

```
CREATE FULLTEXT CATALOG ProductFTCatalog;
```

The following example creates a full-text index for product reviews.

```
CREATE FULLTEXT INDEX
ON ProductReviews (ReviewText)
KEY INDEX PK_ProductReviews
ON ProductFTCatalog;
```

The following example queries the full-text index for reviews containing the word `excellent`.

```
SELECT *
FROM ProductReviews
WHERE CONTAINS(ReviewText, 'excellent');
```

```
ReviewID  ProductID  ReviewText
     ReviewDate  UserID
1         1          This is a review for product 1, it is excellent and works as
 expected  2018-07-01  2
```

For more information, see [Full-Text Search]() in the *SQL Server documentation*.

PostgreSQL Usage

You can use full-text indexes to speed up textual searches performed against textual data by using the full-text `@@` predicate.

You can create full-text indexes on almost any column data type. It depends on the operator class used when the index is created. You can query all classes from the `pg_opclass` table. Also, you can define the default values.

The default class uses index `tsvector` data types. The most common use is to create one column with text or other data type, and use triggers to convert it to a `tsvector`.

There are two index types for full-text searches: `GIN` and `GiST`.

`GIN` is slower when building the index because it is complete and doesn't have false positive results, but it's faster when querying.

You can improve the `GIN` performance on creation by increasing the `maintenance_work_mem` parameter.

When you create `GIN` indexes, you can combine them with these parameters:

- `fastupdate` puts updates on the index on a waiting list so they will occur in `VACUUM` or related scenarios. The default value is `ON`.
- `gin_pending_list_limit`: the maximum size of a waiting list in KB. The default value is 4MB.

You can't use `GIN` as composite index (multi columns) unless you add the `btree_gin` extension (which is supported in Amazon Aurora).

```
CREATE EXTENSION btree_gin;
CREATE INDEX reviews_idx ON reviews USING GIN (title, body);
```

Full-Text Search Functions

Boolean search

You can use `to_tsquery()`, which accepts a list of words is checked against the normalized vector created with `to_tsvector()`. To do this, use the `@@` operator to check if `tsquery` matches `tsvector`. For example, the following statement returns `t` because the column contains the word **boy**. This search also returns `t` for **boys** but not for **boyser**.

```
SELECT to_tsvector('The quick young boy jumped over the fence')
@@ to_tsquery('boy');
```

Operators search

The following example shows how to use the `AND (&)`, `OR (|)`, and `NOT (!)` operators.

```
SELECT to_tsvector('The quick young boy jumped over the fence')
@@ to_tsquery('young & (boy | guy) & !girl');
```

Phase search

When using `to_tsquery`, you can also search for a similar term if you replace boy with boys and add the langauge to be used.

```
SELECT to_tsvector('The quick young boy jumped over the fence')
@@ to_tsquery('english', 'young & (boys | guy) & !girl');
```

Search words within a specific distance. In the following example, `-` is equal to 1. These examples return true.

```
SELECT to_tsvector('The quick young boy jumped over the fence') @@
  to_tsquery('young <-> boy'),
  to_tsvector('The quick young boy jumped over the fence') @@
  to_tsquery('quick <3> jumped');
```

Migration Considerations

Migrating full-text indexes from SQL Server to Aurora PostgreSQL requires a full rewrite of the code that addresses creating, managing, and querying of full-text searches.

Although the Aurora PostgreSQL full-text engine is significantly less comprehensive than SQL Server, it is also much simpler to create and manage, and it is sufficiently powerful for most common, basic full-text requirements.

You can create a text search dictionary. For more information, see CREATE TEXT SEARCH DICTIONARY.

For more complex full-text workloads, use Amazon CloudSearch, a managed service that makes it simple and cost-effective to set up, manage, and scale an enterprise grade search solution. Amazon CloudSearch supports 34 languages and advanced search features such as highlighting, autocomplete, and geospatial search.

Currently, there is no direct tooling integration with Aurora PostgreSQL. Therefore, create a custom application to synchronize the data between Amazon RDS instances and the CloudSearch service.

For more information, see Amazon CloudSearch.

Examples

```
CREATE TABLE ProductReviews
(
  ReviewID SERIAL PRIMARY KEY,
  ProductID INT NOT NULL
  ReviewText TEXT NOT NULL,
  ReviewDate DATE NOT NULL,
  UserID INT NOT NULL
);
```

```
INSERT INTO ProductReviews
```

```
(ProductID, ReviewText, ReviewDate, UserID)
VALUES
(1, 'This is a review for product 1, it is excellent and works as expected',
 '20180701', 2),
(1, 'This is a review for product 1, it isn't that great and failed after two days',
 '20180702', 2),
(2, 'This is a review for product 3, it has exceeded my expectations. A+++',
 '20180710', 2);
```

The following example creates a full-text search index.

```
CREATE INDEX gin_idx ON ProductReviews USING gin (ReviewText gin_trgm_ops);
```

You can use `gin_trgm_ops` to index a `TEXT` data type.

The following example queries the full-text index for reviews containing the word excellent.

```
SELECT * FROM ProductReviews where ReviewText @@ to_tsquery('excellent');
```

For more information, see [Full Text Search](#) and [Additional Features](#) in the *PostgreSQL documentation*.

SQL Server Graph

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	No native support. Rewriting the application is required.

SQL Server Usage

SQL Server offers graph database capabilities to model many-to-many relationships. The graph relationships are integrated into Transact-SQL and receive the benefits of using SQL Server as the foundational database management system.

A graph database is a collection of nodes or vertices and edges or relationships. A node represents an entity. For example, a person or an organization. An edge represents a relationship between the two nodes that it connects. For example, this can be likes or friends. Both nodes and edges may have properties associated with them. Here are some features that make a graph database unique:

- Edges or relationships are first class entities in a Graph Database and can have attributes or properties associated with them.
- A single edge can flexibly connect multiple nodes in a Graph Database.
- You can express pattern matching and multi-hop navigation queries easily.
- You can express transitive closure and polymorphic queries easily.

A relational database can achieve anything a graph database can. However, a graph database makes it easier to express certain kinds of queries. Also, with specific optimizations, certain queries may perform better. Your decision to choose either a relational or graph database is based on following factors:

- Your application has hierarchical data. You can use the `HierarchyID` data type to implement hierarchies, but it has some limitations. For example, it doesn't allow you to store multiple parents for a node.
- Your application has complex many-to-many relationships. As application evolves, new relationships are added.
- You need to analyze interconnected data and relationships.

SQL Server 2017 adds new graph database capabilities for modeling graph many-to-many relationships. They include the new `CREATE TABLE` syntax for creating node and edge tables, and the keyword `MATCH` for queries. For more information, see [Graph processing with SQL Server and Azure SQL Database](#).

The following example creates SQL Server graph tables.

```
CREATE TABLE Person (ID INTEGER PRIMARY KEY, Name VARCHAR(100), Age INT) AS NODE;
CREATE TABLE friends (StartDate date) AS EDGE;
```

A new `MATCH` clause is introduced to support pattern matching and multi-hop navigation through the graph. The `MATCH` function uses ASCII-art style syntax for pattern matching. The following example uses the `MATCH` function.

```
-- Find friends of John
SELECT Person2.Name
FROM Person Person1, Friends, Person Person2
WHERE MATCH(Person1-(Friends)->Person2)
AND Person1.Name = 'John';
```

SQL Server 2019 adds ability to define cascaded delete actions on an edge constraint in a graph database. Edge constraints enable users to add constraints to their edge tables, thereby enforcing specific semantics and also maintaining data integrity. For more information, see [Edge constraints](#) in the *SQL Server documentation*.

In SQL Server 2019, graph tables now have support for table and index partitioning. For more information, see [Partitioned Tables and Indexes](#) in the *SQL Server documentation*.

PostgreSQL Usage

Currently, PostgreSQL doesn't provide native Graph Database feature, but it is possible to implement some of them using recursive CTE queries or serializing graphs to regular relations.

JSON and XML

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		[XML](#)	Syntax and option differences, similar functionality. PostgreSQL doesn't have a `FOR XML` clause.

SQL Server Usage

JavaScript Object Notation (JSON) and eXtensible Markup Language (XML) are the two most common types of semi-structured data documents used by a variety of data interfaces and NoSQL databases. Most REST web service APIs support JSON as their native data transfer format. XML is an older, more mature framework that is still widely used. It provides many extensions such as XQuery, name spaces, schemas, and more.

The following example is a JSON document:

```
[{
  "name": "Robert",
  "age": "28"
}, {
  "name": "James",
  "age": "71"
  "lastname": "Drapers"
}]
```

The following example is the XML counterpart of the preceding example.

```
<?xml version="1.0" encoding="UTF-16" ?>
<root>
  <Person>
    <name>Robert</name>
    <age>28</age>
  </Person>
  <Person>
    <name>James</name>
    <age>71</age>
    <lastname>Drapers</lastname>
  </Person>
</root>
```

SQL Server provides native support for both JSON and XML in the database using the familiar and convenient T-SQL interface.

XML Data

SQL Server provides extensive native support for working with XML data including XML Data Types, XML Columns, XML Indexes, and XQuery.

XML Data Types and Columns

In SQL Server, you can use the following data types to store XML data:

- The Native XML Data Type uses a BLOB structure but preserves the XML Infoset, which consists of the containment hierarchy, document order, and element/attribute values. An XML typed document may differ from the original text; white space is removed and the order of objects may change. XML Data stored as a native XML data type has the additional benefit of schema validation.
- You can use an Annotated Schema (AXSD) to distribute XML documents to one or more tables. Hierarchical structure is maintained, but element order isn't.
- You can use CLOB or BLOB such as `VARCHAR(MAX)` and `VARBINARY(MAX)` to store the original XML document.

XML Indexes

In SQL Server, you can create `PRIMARY` and `SECONDARY` XML indexes on columns with a native XML data type. You can create secondary indexes for `PATH`, `VALUE`, or `PROPERTY`, which are helpful for various types of workload queries.

XQuery

SQL Server supports a subset of the W3C XQUERY language specification. You can run queries directly against XML data and use them as expressions or sets in standard T-SQL statements.

The following example uses the XQuery language specification.

```
DECLARE @XMLVar XML = '<Root><Data>My XML Data</Data></Root>';
SELECT @XMLVar.query('/Root/Data');
```

```
Result: <Data>My XML Data</Data>
```

JSON Data

SQL Server doesn't support a dedicated JSON data type. However, you can store JSON documents in an `NVARCHAR` column. For more information about BLOBS, see Data Types.

SQL Server provides a set of JSON functions. You can use these functions for the following tasks:

- Retrieve and modify values in JSON documents.
- Convert JSON objects to a set (table) format.

- Use standard T-SQL queries with converted JSON objects.
- Convert tabular results of T-SQL queries to JSON format.

The functions are:

- `ISJSON` — Tests if a string contains a valid JSON string. Use in WHERE clause to avoid errors.
- `JSON_VALUE` — Retrieves a scalar value from a JSON document.
- `JSON_QUERY` — Retrieves a whole object or array from a JSON document.
- `JSON_MODIFY` — Modifies values in a JSON document.
- `OPENJSON` — Converts a JSON document to a `SET` that you can use in the `FROM` clause of a T-SQL query.

You can use the `FOR JSON` clause of `SELECT` queries to convert a tabular set to a JSON document.

Examples

The following example creates a table with a native typed XML column.

```
CREATE TABLE MyTable
(
  XMLIdentifier INT NOT NULL PRIMARY KEY,
  XMLDocument XML NULL
);
```

The following example queries a JSON document.

```
DECLARE @JSONVar NVARCHAR(MAX);
SET @JSONVar = '{"Data":{"Person":[{"Name":"John"},{"Name":"Jane"},
{"Name":"Maria"}]}}';
SELECT JSON_QUERY(@JSONVar, '$.Data');
```

For more information, see [JSON data in SQL Server]() and [XML Data (SQL Server)]() in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL provides native JSON Document support using the JSON data types `JSON` and `JSONB`.

JSON stores an exact copy of the input text that processing functions must re-parse on each run. It also preserves semantically-insignificant white space between tokens and the order of keys within JSON objects.

JSONB stores data in a decomposed binary format causing slightly slower input performance due to added conversion to binary overhead. But it is significantly faster to process, since no re-parsing is needed on reads.

- Doesn't preserve white space.
- Doesn't preserve the order of object keys.
- Doesn't keep duplicate object keys. If duplicate keys are specified in the input, only the last value is retained.

Most applications store JSON data as JSONB unless there are specialized needs. For more information, see JSON Types in the *PostgreSQL documentation*.

To comply with the full JSON specification, database encoding must be set to UTF8. If the database code page isn't set to UTF8, then non-UTF8 characters are allowed and the database encoding will be non-compliant with the full JSON specification.

In PostgreSQL version 10 and higher, JSON and JSONB are compatible with full-text search.

Examples

Querying JSON data in PostgreSQL uses different syntax than SQL Server

The following example returns the JSON document stored in the emp_data column associated with emp_id=1.

```
SELECT emp_data FROM employees WHERE emp_id = 1;
```

The following example returns all JSON documents stored in the emp_data column having a key named address.

```
SELECT emp_data FROM employees WHERE emp_data ? ' address';
```

The following example returns all JSON items that have an address key or a hobbies key.

```
SELECT * FROM employees WHERE emp_data ?| array['address', 'hobbies'];
```

The following example returns all JSON items that have both an address key and a hobbies key.

```
SELECT * FROM employees WHERE emp_data ?& array['a', 'b'];
```

The following example returns the value of home key in the phone numbers array.

```
SELECT emp_data ->'phone numbers'->>'home' FROM employees;
```

The following example returns all JSON documents where the address key is equal to a specified value and return all JSON documents where address key contains a specific string (using like).

```
SELECT * FROM employees WHERE emp_data->>'address' = '1234 First Street, Capital City';
SELECT * FROM employees WHERE emp_data->>'address' like '%Capital City%';
```

The following example removes keys from JSON. You can remove more than one key in PostgreSQL 10 only.

```
select '{"id":132, "fname":"John", "salary":999999, "bank_account":1234}'::jsonb -
 '{salary,bank_account}'::text[];
```

For more information, see [JSON Functions and Operators](#) in the *PostgreSQL documentation*.

Indexing and Constraints with JSONB Columns

You can use the CREATE UNIQUE INDEX statement to enforce constraints on values inside JSON documents.

The following example creates a unique index that forces values of the address key to be unique.

```
CREATE UNIQUE INDEX employee_address_uq ON employees( (emp_data->>'address') ) ;
```

This index allows the first SQL insert statement to work and causes the second to fail.

```
INSERT INTO employees VALUES
(2, 'Second Employee','{ "address": "1234 Second Street, Capital City"}');
INSERT INTO employees VALUES
```

```
(3, 'Third Employee', '{ "address": "1234 Second Street, Capital City"}');
ERROR: duplicate key value violates unique constraint "employee_address_uq" SQL state:
23505 Detail: Key ((emp_data ->> 'address'::text))=(1234 Second Street, Capital City)
already exists.
```

For JSON data, PostgreSQL supports B-tree, hash, and Generalized Inverted Indexes (GIN). A GIN index is a special inverted index structure that is useful when an index must map many values to a row (such as indexing JSON documents).

When you use GIN indexes, you can efficiently and quickly query data using only the following JSON operators: `@>, ?, ?&, ?|`.

Without indexes, PostgreSQL is forced to perform a full table scan when filtering data. This condition applies to JSON data and will most likely have a negative impact on performance since Postgres has to step into each JSON document.

The following example creates an index on the address key of emp_data.

```
CREATE idx1_employees ON employees ((emp_data->>'address'));
```

The following example creates a GIN index on a specific key or the entire emp_data column.

```
CREATE INDEX idx2_employees ON cards USING gin ((emp_data->'tags'));
CREATE INDEX idx3_employees ON employees USING gin (emp_data);
```

XML Examples

PostgreSQL provides an XML data type for table columns. The primary advantage of using XML columns, rather than placing XML data in text columns, is that the XML data is type checked when inserted. Additionally, there are support functions to perform type-safe operations.

XML can store well-formed documents as defined by the XML standard or content fragments that defined by the production XMLDecl. Content fragments can have more than one top-level element or character node.

You can use `IS DOCUMENT` to evaluate whether a particular XML value is a full document or only a content fragment.

The following example demonstrates how to create XML data and insert it into a table.

Insert a document, and then insert a content fragment. You can insert both types of XML data into the same column. If the XML is incorrect (such as a missing tag), the insert fails with the relevant error. The query retrieves only document records.

```
CREATE TABLE test (a xml);

insert into test values (XMLPARSE (DOCUMENT '<?xml version="1.0"?
><Series><title>Simpsons</title><chapter>...</chapter></Series>'));

insert into test values (XMLPARSE (CONTENT 'note<tag>value</tag><tag>value</tag>'));

select * from test where a IS DOCUMENT;
```

Converting XML data to rows was a feature added in PostgreSQL 10. This can be very helpful reading XML data using a table equivalent.

```
CREATE TABLE xmldata_sample AS SELECT
xml $$
<ROWS>
  <ROW id="1">
    <EMP_ID>532</EMP_ID>
    <EMP_NAME>John</EMP_NAME>
  </ROW>
  <ROW id="5">
    <EMP_ID>234</EMP_ID>
    <EMP_NAME>Carl</EMP_NAME>
    <EMP_DEP>6</EMP_DEP>
    <SALARY unit="dollars">10000</SALARY>
  </ROW>
  <ROW id="6">
    <EMP_ID>123</EMP_ID>
    <EMP_DEP>8</EMP_DEP>
    <SALARY unit="dollars">5000</SALARY>
  </ROW>
</ROWS>
$$ AS data;

SELECT xmltable.*
  FROM xmldata_sample,
    XMLTABLE('//ROWS/ROW'
      PASSING data
      COLUMNS id int PATH '@id',
```

```
        ordinality FOR ORDINALITY,
        "EMP_NAME" text,
        "EMP_ID" text PATH 'EMP_ID',
        SALARY_USD float PATH 'SALARY[@unit = "dollars"]',
        MANAGER_NAME text PATH 'MANAGER_NAME' DEFAULT 'not specified');

id  ordinality  EMP_NAME  EMP_ID  salary_usd  manager_name
1   1           John      532                 not specified
5   2           Carl      234     10000       not specified
6   3                     123     5000        not specified
```

Summary

The following table identifies similarities, differences, and key migration considerations.

Feature	SQL Server	Aurora PostgreSQL
XML and JSON native data types.	XML with schema collections.	JSON.
JSON functions.	`IS_JSON`, `JSON_VALUE`, `JSON_QUERY`, `JSON_MODIFY`, `OPEN_JSON`, `FOR JSON`.	A set of more than 20 dedicated JSON functions. For more information, see JSON Functions and Operators in the *PostgreSQL documentation*.
XML functions	`XQUERY` and `XPATH`, `OPEN_XML`, `FOR XML`.	Many XML functions. For more information, see XML Functions in the *PostgreSQL documentation*. PostgreSQL doesn't have a `FOR XML` clause. You can use `string_agg` instead.
XML and JSON indexes.	Primary and Secondary `PATH`, `VALUE` and `PROPERTY` indexes.	Supported.

For more information, see XML Type, XML Functions, JSON Types, and JSON Functions and Operators in the *PostgreSQL documentation*.

MERGE

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		MERGE	Rewrite to use `INSERT… ON CONFLICT`.

SQL Server Usage

`MERGE` is a complex , hybrid DML/DQL statement for performing `INSERT`, `UPDATE`, or `DELETE` operations on a target table based on the results of a logical join of the target table and a source data set.

`MERGE` can also return row sets similar to SELECT using the OUTPUT clause, which gives the calling scope access to the actual data modifications of the `MERGE` statement.

The `MERGE` statement is most efficient for non-trivial conditional DML. For example, inserting data if a row key value doesn't exist and updating the existing row if the key value already exists.

You can easily manage additional logic such as deleting rows from the target that don't appear in the source. For simple, straightforward updates of data in one table based on data in another, it is typically more efficient to use simple `INSERT`, `DELETE`, and `UPDATE` statements. You can replace all `MERGE` functionality with `INSERT`, `DELETE`, and `UPDATE` statements, but not necessarily less efficiently.

The SQL Server `MERGE` statement provides a wide range of functionality and flexibility and is compatible with ANSI standard SQL:2008. SQL Server has many extensions to `MERGE` that provide efficient T-SQL solutions for synchronizing data.

Syntax

```
MERGE [INTO] <Target Table> [AS] <Table Alias>]
USING <Source Table>
```

```
ON <Merge Predicate>
[WHEN MATCHED [AND <Predicate>]
THEN UPDATE SET <Column Assignments...> | DELETE]
[WHEN NOT MATCHED [BY TARGET] [AND <Predicate>]
THEN INSERT [(<Column List>)]
VALUES (<Values List>) | DEFAULT VALUES]
[WHEN NOT MATCHED BY SOURCE [AND <Predicate>]
THEN UPDATE SET <Column Assignments...> | DELETE]
OUTPUT [<Output Clause>]
```

Examples

The following example performs a simple one-way synchronization of two tables.

```
CREATE TABLE SourceTable
(
  Col1 INT NOT NULL PRIMARY KEY,
  Col2 VARCHAR(20) NOT NULL
);
```

```
CREATE TABLE TargetTable
(
  Col1 INT NOT NULL PRIMARY KEY,
  Col2 VARCHAR(20) NOT NULL
);
```

```
INSERT INTO SourceTable (Col1, Col2)
VALUES
(2, 'Source2'),
(3, 'Source3'),
(4, 'Source4');
```

```
INSERT INTO TargetTable (Col1, Col2)
VALUES
(1, 'Target1'),
(2, 'Target2'),
(3, 'Target3');
```

```
MERGE INTO TargetTable AS TGT
USING SourceTable AS SRC ON TGT.Col1 = SRC.Col1
```

```
WHEN MATCHED
  THEN UPDATE SET TGT.Col2 = SRC.Col2
WHEN NOT MATCHED
  THEN INSERT (Col1, Col2)
  VALUES (SRC.Col1, SRC.Col2);
```

```
SELECT * FROM TargetTable;
```

For the preceding examples, the result looks as shown following.

```
Col1  Col2
1     Target1
2     Source2
3     Source3
4     Source4
```

Perform a conditional two-way synchronization using `NULL` for no change and `DELETE` from the target when the data isn't found in the source.

```
TRUNCATE TABLE SourceTable;
INSERT INTO SourceTable (Col1, Col2) VALUES (3, NULL), (4, 'NewSource4'),
 (5,'Source5');
```

```
MERGE INTO TargetTable AS TGT
USING SourceTable AS SRC ON TGT.Col1 = SRC.Col1
WHEN MATCHED AND SRC.Col2 IS NOT NULL
  THEN UPDATE SET TGT.Col2 = SRC.Col2
WHEN NOT MATCHED
  THEN INSERT (Col1, Col2)
    VALUES (SRC.Col1, SRC.Col2)
WHEN NOT MATCHED BY SOURCE
  THEN DELETE;
```

```
SELECT *
FROM TargetTable;
```

For the preceding examples, the result looks as shown following.

```
Col1  Col2
3     Source3
```

```
4       NewSource4
5       Source5
```

For more information, see [MERGE (Transact-SQL)](#) in the *SQL Server documentation*.

PostgreSQL Usage

Currently, PostgreSQL version 10 doesn't support the use of the `MERGE` command. As an alternative, consider using the `INSERT… ON CONFLICT` clause, which can handle cases where insert clauses might cause a conflict, and then redirect the operation as an update.

Examples

The following example uses the `ON ONFLICT` clause.

```
CREATE TABLE EMP_BONUS (
EMPLOYEE_ID NUMERIC,
BONUS_YEAR VARCHAR(4),
SALARY NUMERIC,
BONUS NUMERIC,
PRIMARY KEY (EMPLOYEE_ID, BONUS_YEAR));

INSERT INTO EMP_BONUS (EMPLOYEE_ID, BONUS_YEAR, SALARY)
  SELECT EMPLOYEE_ID, EXTRACT(YEAR FROM NOW()), SALARY
  FROM EMPLOYEES
  WHERE SALARY < 10000
  ON CONFLICT (EMPLOYEE_ID, BONUS_YEAR)
  DO UPDATE SET BONUS = EMP_BONUS.SALARY * 0.5;
  SELECT * FROM EMP_BONUS;

employee_id  bonus_year  salary   bonus
103          2017        9000.00  4500.000
104          2017        6000.00  3000.000
105          2017        4800.00  2400.000
106          2017        4800.00  2400.000
107          2017        4200.00  2100.000
109          2017        9000.00  4500.000
110          2017        8200.00  4100.000
111          2017        7700.00  3850.000
112          2017        7800.00  3900.000
113          2017        6900.00  3450.000
115          2017        3100.00  1550.000
116          2017        2900.00  1450.000
```

```
117          2017         2800.00  1400.000
118          2017         2600.00  1300.000
```

Running the same operation multiple times using the `ON CONFLICT` clause doesn't generate an error because the existing records are redirected to the update clause.

For more information, see INSERT and Unsupported Features in the *PostgreSQL documentation*.

PIVOT and UNPIVOT

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		PIVOT and UNPIVOT	Straightforward rewrite to use traditional SQL syntax.

SQL Server Usage

`PIVOT` and `UNPIVOT` are relational operations used to transform a set by rotating rows into columns and columns into rows.

PIVOT

The `PIVOT` operator consists of several clauses and implied expressions.

The *anchor column* isn't pivoted and results in a single row for each unique value, similar to `GROUP BY`.

The pivoted columns are derived from the `PIVOT` clause and are the row values transformed into columns. The values for these columns are derived from the source column defined in the `PIVOT` clause.

PIVOT Syntax

```
SELECT <Anchor column>,
  [Pivoted Column 1] AS <Alias>,
  [Pivoted column 2] AS <Alias>
```

```
  ...n
FROM
  (<SELECT Statement of Set to be Pivoted>)
  AS <Set Alias>
PIVOT
(
  <Aggregate Function>(<Aggregated Column>)
FOR
[<Column With the Values for the Pivoted Columns Names>]
  IN ( [Pivoted Column 1], [Pivoted column 2] ...)
) AS <Pivot Table Alias>;
```

PIVOT Examples

The following example creates and populates the Orders table.

```
CREATE TABLE Orders
(
  OrderID INT NOT NULL
  IDENTITY(1,1) PRIMARY KEY,
  OrderDate DATE NOT NULL,
  Customer VARCHAR(20) NOT NULL
);
```

```
INSERT INTO Orders (OrderDate, Customer)
VALUES
('20180101', 'John'),
('20180201', 'Mitch'),
('20180102', 'John'),
('20180104', 'Kevin'),
('20180104', 'Larry'),
('20180104', 'Kevin'),
('20180104', 'Kevin');
```

The following example creates a simple PIVOT for the number of orders for each day. Days of month from 5 to 31 are omitted for example simplicity.

```
SELECT 'Number of Orders for Day' AS DayOfMonth,
  [1], [2], [3], [4] /*...[31]*/
FROM (
  SELECT OrderID,
    DAY(OrderDate) AS OrderDay
```

```
  FROM Orders
  ) AS SourceSet
PIVOT
(
  COUNT(OrderID)
  FOR OrderDay IN ([1], [2], [3], [4] /*...[31]*/)
) AS PivotSet;
```

For the preceding example, the result looks as shown following.

```
DayOfMonth                1  2  3  4  /*...[31]*/
Number of Orders for Day  2  1  0  4
```

The result set is now oriented in rows against columns. The first column is the description of the columns to follow.

PIVOT for number of orders for each day, for each customer.

```
SELECT Customer,
  [1], [2], [3], [4] /*...[31]*/
FROM (
  SELECT OrderID,
    Customer,
    DAY(OrderDate) AS OrderDay
  FROM Orders
  ) AS SourceSet
PIVOT
(
  COUNT(OrderID)
  FOR OrderDay IN ([1], [2], [3], [4] /*...[31]*/)
) AS PivotSet;
```

```
Customer  1  2  3  4
John      1  1  0  0
Kevin     0  0  0  3
Larry     0  0  0  1
Mitch     1  0  0  0
```

UNPIVOT

`UNPIVOT` is similar to `PIVOT` in reverse, but spreads existing column values into rows.

The source set is similar to the result of the PIVOT with values pertaining to particular entities listed in columns. Because the result set has more rows than the source, aggregations aren't required.

It is less commonly used than PIVOT because most data in relational databases have attributes in columns; not the other way around.

UNPIVOT Examples

The following example creates and populates the pivot-like EmployeeSales table. This is most likely a view or a set from an external source.

```
CREATE TABLE EmployeeSales
(
  SaleDate DATE NOT NULL PRIMARY KEY,
  John INT,
  Kevin INT,
  Mary INT
);
```

```
INSERT INTO EmployeeSales
VALUES
('20180101', 150, 0, 300),
('20180102', 0, 0, 0),
('20180103', 250, 50, 0),
('20180104', 500, 400, 100);
```

The following example unpivots employee sales for each date into individual rows for each employee.

```
SELECT SaleDate,
  Employee,
  SaleAmount
FROM
(
  SELECT SaleDate, John, Kevin, Mary
  FROM EmployeeSales
) AS SourceSet
UNPIVOT (
  SaleAmount
  FOR Employee IN (John, Kevin, Mary)
```

```
)AS UnpivotSet;
```

For the preceding example, the result looks as shown following.

```
SaleDate    Employee  SaleAmount
2018-01-01  John      150
2018-01-01  Kevin     0
2018-01-01  Mary      300
2018-01-02  John      0
2018-01-02  Kevin     0
2018-01-02  Mary      0
2018-01-03  John      250
2018-01-03  Kevin     50
2018-01-03  Mary      0
2018-01-04  John      500
2018-01-04  Kevin     400
2018-01-04  Mary      100
```

For more information, see [FROM - Using PIVOT and UNPIVOT] in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) doesn't support the `PIVOT` and `UNPIVOT` relational operators.

You can rewrite the functionality of these operators to use standard SQL syntax, as shown in the following examples.

PIVOT Examples

The following example creates and populates the Orders table.

```
CREATE TABLE Orders
(
  OrderID SERIAL PRIMARY KEY,
  OrderDate DATE NOT NULL,
  Customer VARCHAR(20) NOT NULL
);
```

```
INSERT INTO Orders (OrderDate, Customer)
VALUES
```

```
('20180101', 'John'),
('20180201', 'Mitch'),
('20180102', 'John'),
('20180104', 'Kevin'),
('20180104', 'Larry'),
('20180104', 'Kevin'),
('20180104', 'Kevin');
```

The following example creates a simple PIVOT for the number of orders for each day. Days of month from 5 to 31 are omitted for example simplicity.

```
SELECT 'Number of Orders for Day' AS DayOfMonth,
COUNT(CASE WHEN date_part('day', OrderDate) = 1 THEN 'OrderDate' ELSE NULL END) AS "1",
COUNT(CASE WHEN date_part('day', OrderDate) = 2 THEN 'OrderDate' ELSE NULL END) AS "2",
COUNT(CASE WHEN date_part('day', OrderDate) = 3 THEN 'OrderDate' ELSE NULL END) AS "3",
COUNT(CASE WHEN date_part('day', OrderDate) = 4 THEN 'OrderDate' ELSE NULL END) AS
 "4" /*...[31]*/
FROM Orders AS O;
```

For the preceding example, the result looks as shown following.

```
DayOfMonth                1  2  3  4  /*...[31]*/
Number of Orders for Day  2  1  0  4
```

PIVOT for number of orders for each day, for each customer.

```
SELECT Customer,
COUNT(CASE WHEN date_part('day', OrderDate) = 1 THEN 'OrderDate' ELSE NULL END) AS "1",
COUNT(CASE WHEN date_part('day', OrderDate) = 2 THEN 'OrderDate' ELSE NULL END) AS "2",
COUNT(CASE WHEN date_part('day', OrderDate) = 3 THEN 'OrderDate' ELSE NULL END) AS "3",
COUNT(CASE WHEN date_part('day', OrderDate) = 4 THEN 'OrderDate' ELSE NULL END) AS
 "4" /*...[31]*/
FROM Orders AS O
GROUP BY Customer;
```

For the preceding example, the result looks as shown following.

```
Customer  1  2  3  4
John      1  1  0  0
Kevin     0  0  0  3
Larry     0  0  0  1
```

```
Mitch     1  0  0  0
```

UNPIVOT Examples

The following example creates and populates the pivot-like `EmployeeSales` table. In real life this will most likely be a view, or a set from an external source.

```
CREATE TABLE EmployeeSales
(
  SaleDate DATE NOT NULL PRIMARY KEY,
  John INT,
  Kevin INT,
  Mary INT
);
```

```
INSERT INTO EmployeeSales
VALUES
('20180101', 150, 0, 300),
('20180102', 0, 0, 0),
('20180103', 250, 50, 0),
('20180104', 500, 400, 100);
```

The following example unpivots employee sales for each date into individual rows for each employee.

```
SELECT SaleDate, Employee, SaleAmount
FROM (
  SELECT SaleDate,
    Employee,
    CASE
      WHEN Employee = 'John' THEN 'John'
      WHEN Employee = 'Kevin' THEN 'Kevin'
      WHEN Employee = 'Mary' THEN 'Mary'
    END AS SaleAmount
  FROM EmployeeSales as emp
  CROSS JOIN
  (
    SELECT 'John' AS Employee
    UNION ALL
    SELECT 'Kevin'
    UNION ALL
```

```
    SELECT 'Mary'
  ) AS Employees
) AS UnpivotedSet;
```

For the preceding example, the result looks as shown following.

```
SaleDate    Employee  SaleAmount
2018-01-01  John      150
2018-01-01  Kevin     0
2018-01-01  Mary      300
2018-01-02  John      0
2018-01-02  Kevin     0
2018-01-02  Mary      0
2018-01-03  John      250
2018-01-03  Kevin     50
2018-01-03  Mary      0
2018-01-04  John      500
2018-01-04  Kevin     400
2018-01-04  Mary      100
```

Triggers

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		[Triggers](#)	Syntax and option differences, similar functionality.

SQL Server Usage

Triggers are special types of stored procedures that run automatically in response to events. They are most commonly used for Data Manipulation Language (DML).

SQL Server supports `AFTER`, `FOR`, and `INSTEAD OF` triggers, which you can create on tables and views (`AFTER` and `FOR` are synonymous). SQL Server also provides an event trigger framework at the server and database levels that includes Data Definition Language (DDL), Data Control Language (DCL), and general system events such as login.

Note

SQL Server doesn't support `FOR EACH ROW` triggers in which the trigger code is run once for each row of modified data.

Trigger Run

`AFTER` triggers runs after DML statements complete run. `INSTEAD OF` triggers run code in place of the original DML statement. You can create `AFTER` triggers on tables only. You can create `INSTEAD OF` triggers on tables and views.

You can create only one `INSTEAD OF` trigger for any given object and event. When multiple `AFTER` triggers exist for the same event and object, you can partially set the trigger order by using the `sp_settriggerorder` system stored procedure. You can use it to set the first and last triggers to be run, but not the order of others.

Trigger Scope

SQL Server supports statement level triggers only. The trigger code runs once for each statement. The data modified by the DML statement is available to the trigger scope and is saved in two virtual tables: `INSERTED` and `DELETED`. These tables contain the entire set of changes performed by the DML statement that caused trigger run.

SQL Server triggers always run within the transaction of the statement that triggered the run. If the trigger code issues an explicit `ROLLBACK`, or causes an exception that mandates a rollback, the DML statement is also rolled back. For `INSTEAD OF` triggers, the DML statement doesn't run and doesn't require a rollback.

Examples

Use a DML trigger to audit invoice deletions

The following examples demonstrate how to use a trigger to log rows deleted from a table.

Create and populate the `Invoices` table.

```
CREATE TABLE Invoices
(
  InvoiceID INT NOT NULL PRIMARY KEY,
```

```
  Customer VARCHAR(20) NOT NULL,
  TotalAmount DECIMAL(9,2) NOT NULL
);

INSERT INTO Invoices (InvoiceID,Customer,TotalAmount)
VALUES
(1, 'John', 1400.23),
(2, 'Jeff', 245.00),
(3, 'James', 677.22);
```

Create the `InvoiceAuditLog` table.

```
CREATE TABLE InvoiceAuditLog
(
  InvoiceID INT NOT NULL PRIMARY KEY,
  Customer VARCHAR(20) NOT NULL,
  TotalAmount DECIMAL(9,2) NOT NULL,
  DeleteDate DATETIME NOT NULL DEFAULT (GETDATE()),
  DeletedBy VARCHAR(128) NOT NULL DEFAULT (CURRENT_USER)
);
```

Create an `AFTER DELETE` trigger to log deletions from the `Invoices` table to the audit log.

```
CREATE TRIGGER LogInvoiceDeletes
ON Invoices
AFTER DELETE
AS
BEGIN
INSERT INTO InvoiceAuditLog (InvoiceID, Customer, TotalAmount)
SELECT InvoiceID,
  Customer,
  TotalAmount
FROM Deleted
END;
```

Delete an invoice.

```
DELETE FROM Invoices
WHERE InvoiceID = 3;
```

Query the content of both tables.

```
SELECT *
FROM Invoices AS I
  FULL OUTER JOIN
  InvoiceAuditLog AS IAG
  ON I.InvoiceID = IAG.InvoiceID;
```

For the preceding example, the result looks as shown following.

```
InvoiceID  Customer  TotalAmount  InvoiceID  Customer  TotalAmount  DeleteDate
 DeletedBy
1          John      1400.23      NULL       NULL      NULL         NULL
 NULL
2          Jeff      245.00       NULL       NULL      NULL         NULL
 NULL
NULL       NULL      NULL         3          James     677.22       20180224 13:02
 Domain/JohnCortney
```

Create a DDL trigger

Create a trigger to protect all tables in the database from accidental deletion.

```
CREATE TRIGGER PreventTableDrop
ON DATABASE FOR DROP_TABLE
AS
BEGIN
  RAISERROR ('Tables can't be dropped in this database', 16, 1)
  ROLLBACK TRANSACTION
END;
```

Test the trigger by attempting to drop a table.

```
DROP TABLE [Invoices];
  GO
```

The system displays the following message explaining that the Invoices table can’t be dropped:

```
Msg 50000, Level 16, State 1, Procedure PreventTableDrop, Line 5 [Batch Start Line 56]
Tables Can't be dropped in this database.
Msg 3609, Level 16, State 2, Line 57
The transaction ended in the trigger. The batch has been aborted.
```

For more information, see DML Triggers and DDL Triggers in the *SQL Server documentation*.

PostgreSQL Usage

Triggers provide much of the same functionality as SQL Server:

- DML triggers run based on table related events, such as DML.
- Event triggers run after certain database events, such as running DDL commands.

Unlike SQL Server triggers, PostgreSQL triggers must call a function. They don't support anonymous blocks of PL/pgSQL code as part of the trigger body. The user-supplied function is declared with no arguments and has a return type of trigger.

PostgreSQL DML Triggers

PostgreSQL triggers can be fired BEFORE or AFTER a DML operation.

- They run before the operation is attempted on a row.
  - Before constraints are checked and the INSERT, UPDATE, or DELETE is attempted.
  - If the trigger runs before or instead of the event, the trigger can skip the operation for the current row or change the row being inserted (for INSERT and UPDATE operations only).
- Triggers can run after the operation was completed, after constraints are checked, and the `INSERT`, `UPDATE`, or `DELETE` command completed. If the trigger runs after the event, all changes, including the effects of other triggers, are visible to the trigger.

PostgreSQL triggers can run `INSTEAD OF` a DML command when created on views.

PostgreSQL triggers can run `FOR EACH ROW` affected by the DML statement or `FOR EACH STATEMENT` running only once as part of a DML statement.

When fired	Database event	Row-Level trigger (FOR EACH ROW)	Statement-level trigger (FOR EACH STATEMENT)
BEFORE	INSERT, UPDATE, DELETE	Tables and foreign tables	Tables, views, and foreign tables

When fired	Database event	Row-Level trigger (FOR EACH ROW)	Statement-level trigger (FOR EACH STATEMENT)
BEFORE	TRUNCATE	—	Tables
AFTER	INSERT, UPDATE, DELETE	Tables and foreign tables	Tables, views, and foreign tables
AFTER	TRUNCATE	—	Tables
INSTEAD OF	INSERT, UPDATE, DELETE	Views	—
INSTEAD OF	TRUNCATE	—	—

PostgreSQL Event Triggers

An event trigger runs when a specific event associated with the trigger occurs in the database. Supported events include `ddl_command_start`, `ddl_command_end`, `table_rewrite`, and `sql_drop`.

- `ddl_command_start` occurs before the run of a `CREATE`, `ALTER`, `DROP`, `SECURITY LABEL`, `COMMENT`, `GRANT`, `REVOKE`, or `SELECT INTO` command.
- `ddl_command_end` occurs after the command completed and before the transaction commits.
- `sql_drop` runs only for the `DROP` DDL command, before the `ddl_command_end` trigger runs.

For a full list of supported PostgreSQL event trigger types, see Event Trigger Firing Matrix in the *PostgreSQL documentation*.

PostgreSQL CREATE TRIGGER Synopsis

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ]}
  ON table_name
  [ FROM referenced_table_name ]
  [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
  [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
  [ FOR [ EACH ] { ROW | STATEMENT } ]
```

```
  [ WHEN ( condition ) ]
  EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:

  INSERT
  UPDATE [ OF column_name [, ... ] ]
  DELETE
  TRUNCATE
```

Note

REFERENCING is a new option since PostgreSQL 10. You can use it with AFTER trigger to interact with the overall view of the OLD or the NEW TABLE changed rows.

Examples

Create a trigger

Create a trigger function that stores the run logic (this is the same as a SQL Server DML trigger).

```
CREATE OR REPLACE FUNCTION PROJECTS_SET_NULL()
  RETURNS TRIGGER
  AS $$
  BEGIN
IF TG_OP = 'UPDATE' AND OLD.PROJECTNO != NEW.PROJECTNO OR
  TG_OP = 'DELETE' THEN
UPDATE EMP
  SET PROJECTNO = NULL
  WHERE EMP.PROJECTNO = OLD.PROJECTNO;
  END IF;
  IF TG_OP = 'UPDATE' THEN RETURN NULL;
    ELSIF TG_OP = 'DELETE' THEN RETURN NULL;
  END IF;
END; $$
LANGUAGE PLPGSQL;

CREATE FUNCTION
```

Create the trigger.

```
CREATE TRIGGER TRG_PROJECTS_SET_NULL
AFTER UPDATE OF PROJECTNO OR DELETE
ON PROJECTS
FOR EACH ROW
EXECUTE PROCEDURE PROJECTS_SET_NULL();

CREATE TRIGGER
```

Test the trigger by deleting a row from the `PROJECTS` table.

```
DELETE FROM PROJECTS WHERE PROJECTNO=123;
SELECT PROJECTNO FROM EMP WHERE PROJECTNO=123;

projectno
(0 rows)
```

Create a trigger

Create an event trigger function. This is the same as a SQL Server DDL System/Schema level trigger, such as a trigger that prevents running a DDL DROP on objects in the HR schema.

Note that trigger functions are created with no arguments and must have a return type of `TRIGGER` or `EVENT_TRIGGER`.

```
CREATE OR REPLACE FUNCTION ABORT_DROP_COMMAND()
  RETURNS EVENT_TRIGGER
  AS $$
BEGIN
  RAISE EXCEPTION 'The % Command is Disabled', tg_tag;
END; $$
LANGUAGE PLPGSQL;

CREATE FUNCTION
```

Create the event trigger, which runs before the start of a DDL `DROP` command.

```
CREATE EVENT TRIGGER trg_abort_drop_command
  ON DDL_COMMAND_START
  WHEN TAG IN ('DROP TABLE', 'DROP VIEW', 'DROP FUNCTION', 'DROP
    SEQUENCE', 'DROP MATERIALIZED VIEW', 'DROP TYPE')
```

```
  EXECUTE PROCEDURE abort_drop_command();
```

Test the trigger by attempting to drop the EMPLOYEES table.

```
DROP TABLE EMPLOYEES;

ERROR: The DROP TABLE Command is Disabled
CONTEXT: PL/pgSQL function abort_drop_command() line 3 at RAISE
```

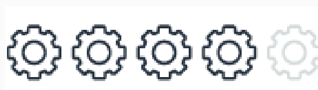
Summary

Feature	SQL Server	Aurora PostgreSQL
DML Triggers Scope	Statement level only	FOR EACH ROW and FOR EACH STATMENT
Access to change set	INSERTED and DELETED virtual multi-row tables	OLD and NEW virtual one-row tables or the whole view of changed rows
System event triggers	DDL, DCL, and other event types	Event triggers
Trigger run phase	AFTER and INSTEAD OF	AFTER, BEFORE, and INSTEAD OF
Multi-trigger run order	Can only set first and last using sp_settriggerorder	Call function within a function
Drop a trigger	DROP TRIGGER <trigger name>;	DROP TRIGGER <trigger name>;
Modify trigger code	Use the ALTER TRIGGER statement	Modify function code
Enable or disable a trigger	Use the ALTER TRIGGER <trigger name> ENABLE; and ALTER TRIGGER	ALTER TABLE

Feature	SQL Server	Aurora PostgreSQL
	<trigger name> DISABLE;	
Triggers on views	INSTEAD OF triggers only	INSTEAD OF triggers only

For more information, see Trigger Functions in the *PostgreSQL documentation*.

SQL Server TOP and FETCH and PostgreSQL LIMIT and OFFSET

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		TOP and FETCH	PostgreSQL doesn't support TOP.

SQL Server Usage

SQL Server supports two options for limiting and paging result sets returned to the client. TOP is a legacy, proprietary T-SQL keyword that is still supported due to its wide usage. The ANSI compliant syntax of FETCH and OFFSET were introduced in SQL Server 2012 and are recommended for paginating results sets.

TOP

The TOP (n) operator is used in the SELECT list and limits the number of rows returned to the client based on the ORDER BY clause.

> **Note**
>
> When TOP is used with no ORDER BY clause, the query is non-deterministic and may return any rows up to the number specified by the TOP operator.

You can use TOP (n) with two modifier options:

- `TOP (n) PERCENT` is used to designate a percentage of the rows to be returned instead of a fixed maximal row number `limit (n)`. When you use `PERCENT`, `n` can be any value from 1-100.
- `TOP (n) WITH TIES` is used to allow overriding the n maximal number or percentage of rows specified in case there are additional rows with the same ordering values as the last row.

If you use `TOP (n)` without `WITH TIES` and there are additional rows that have the same ordering value as the last row in the group of n rows, the query is also non-deterministic because the last row may be any of the rows that share the same ordering value.

Syntax

```
ORDER BY <Ordering Expression> [ ASC | DESC ] [ ,...n ]
OFFSET <Offset Expression> { ROW | ROWS }
[FETCH { FIRST | NEXT } <Page Size Expression> { ROW | ROWS } ONLY ]
```

Examples

The following example creates the OrderItems table.

```
CREATE TABLE OrderItems
(
  OrderID INT NOT NULL,
  Item VARCHAR(20) NOT NULL,
  Quantity SMALLINT NOT NULL,
  PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Locking Nut', 300);
```

The following example retrieves the 3 most ordered items by quantity.

```
-- Using TOP
SELECT TOP (3) *
FROM OrderItems
```

```
ORDER BY Quantity DESC;

-- USING FETCH
SELECT *
FROM OrderItems
ORDER BY Quantity DESC
OFFSET 0 ROWS FETCH NEXT 3 ROWS ONLY;
```

For the preceding example, the result looks as shown following.

```
OrderID  Item             Quantity
3        M6 Locking Nut   300
3        M8 Washer        200
2        M8 Nut           100
```

The following example includes rows with ties.

```
SELECT TOP (3) WITH TIES *
FROM OrderItems
ORDER BY Quantity DESC;
```

For the preceding example, the result looks as shown following.

```
OrderID  Item             Quantity
3        M6 Locking Nut   300
3        M8 Washer        200
2        M8 Nut           100
1        M8 Bolt          100
```

The following example retrieves half the rows based on quantity.

```
SELECT TOP (50) PERCENT *
FROM OrderItems
ORDER BY Quantity DESC;
```

For the preceding example, the result looks as shown following.

```
OrderID  Item             Quantity
3        M6 Locking Nut   300
3        M8 Washer        200
```

For more information, see SELECT - ORDER BY Clause (Transact-SQL) and TOP (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) supports the non-ANSI compliant but popular with other engines `LIMIT… OFFSET` operator for paging results sets.

The `LIMIT` clause limits the number of rows returned and doesn't require an `ORDER BY` clause, although that would make the query non-deterministic.

The `OFFSET` clause is zero-based, similar to SQL Server and used for pagination. `OFFSET 0` is the same as omitting the `OFFSET` clause, as is `OFFSET` with a NULL argument.

Syntax

```
SELECT select_list
  FROM table_expression
  [ ORDER BY ... ]
  [ LIMIT { number | ALL } ] [ OFFSET number ]
```

Migration Considerations

You can use the `LIMIT… OFFSET` syntax to replace the functionality of `TOP(n)` and `FETCH… OFFSET` in SQL Server. It is automatically converted by the AWS Schema Conversion Tool (AWS SCT) except for the `WITH TIES` and `PERCENT` modifiers.

To replace the `PERCENT` option, first calculate how many rows the query returns and then calculate the fixed number of rows to be returned based on that number.

Note

Because this technique involves added complexity and accessing the table twice, consider changing the logic to use a fixed number instead of percentage.

To replace the `WITH TIES` option, rewrite the logic to add another query that checks for the existence of additional rows that have the same ordering value as the last row returned from the `LIMIT` clause.

Note

Because this technique introduces significant added complexity and three accesses to the source table, consider changing the logic to introduce a tie-breaker into the ORDER BY clause.

Examples

The following example creates the OrderItems table.

```
CREATE TABLE OrderItems
(
  OrderID INT NOT NULL,
  Item VARCHAR(20) NOT NULL,
  Quantity SMALLINT NOT NULL,
  PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Locking Nut', 300);
```

The following example retrieves the three most ordered items by quantity.

```
SELECT *
FROM OrderItems
ORDER BY Quantity DESC
LIMIT 3 OFFSET 0;
```

For the preceding example, the result looks as shown following.

```
OrderID  Item            Quantity
3        M6 Locking Nut  300
3        M8 Washer       200
1        M8 Bolt         100
```

The following example includes rows with ties.

```
SELECT *
FROM
(
  SELECT *
  FROM OrderItems
  ORDER BY Quantity DESC
  LIMIT 3 OFFSET 0
) AS X
UNION
SELECT *
FROM OrderItems
WHERE Quantity = (
  SELECT Quantity
  FROM OrderItems
  ORDER BY Quantity DESC
  LIMIT 1 OFFSET 2
)
ORDER BY Quantity DESC
```

For the preceding example, the result looks as shown following.

```
OrderID  Item            Quantity
3        M6 Locking Nut  300
3        M8 Washer       200
2        M8 Nut          100
1        M8 Bolt         100
```

The following example retrieves half the rows based on quantity.

```
CREATE or replace FUNCTION getOrdersPct(int) RETURNS SETOF OrderItems AS $$
SELECT * FROM OrderItems
ORDER BY Quantity desc LIMIT (SELECT COUNT(*)*$1/100 FROM OrderItems) OFFSET 0;
$$ LANGUAGE SQL;
```

```
SELECT * from getOrdersPct(50);
or
SELECT getOrdersPct(50);

OrderID  Item            Quantity
```

```
3         M6 Locking Nut  300
3         M8 Washer       200
```

Summary

SQL Server	Aurora PostgreSQL	Comments
`TOP (n)`	`LIMIT n`	
`TOP (n) WITH TIES`	Not supported	See examples for workaround
`TOP (n) PERCENT`	Not supported	See examples for workaround
`OFFSET… FETCH`	`LIMIT… OFFSET`	

For more information, see LIMIT and OFFSET in the *PostgreSQL documentation*.

User-Defined Functions

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Syntax and option differences.

SQL Server Usage

User-Defined Functions (UDF) are code objects that accept input parameters and return either a scalar value or a set consisting of rows and columns. You can use T-SQL or Common Language Runtime (CLR) code to implement SQL Server UDFs.

> **Note**
>
> This section doesn't cover CLR code objects.

Function invocations can't have any lasting impact on the database. They must be contained and can only modify objects and data local to their scope (for example, data in local variables). Functions aren't allowed to modify data or the structure of a database.

Functions may be deterministic or non-deterministic. Deterministic functions always return the same result when you run them with the same data. Non-deterministic functions may return different results each time they run. For example, a function that returns the current date or time.

SQL Server supports three types of T-SQL UDFs: Scalar Functions, Table-Valued Functions, and Multi-Statement Table-Valued Functions.

SQL Server 2019 adds scalar user-defined functions (UDF) inlining. Inlining transforms functions into relational expressions and embeds them in the calling SQL query. This transformation improves the performance of workloads that take advantage of scalar UDFs. Scalar UDF inlining facilitates cost-based optimization of operations inside UDFs. The results are efficient, set-oriented, and parallel instead of inefficient, iterative, serial run plans. For more information, see Scalar UDF Inlining in the *SQL Server documentation*.

Scalar User-Defined Functions

Scalar UDFs accept zero or more parameters and return a scalar value. You can use scalar UDFs in T-SQL expressions.

Syntax

```
CREATE FUNCTION <Function Name> ([{<Parameter Name> [AS] <Data Type> [= <Default
Value>] [READONLY]} [,...n]])
RETURNS <Return Data Type>
[AS]
BEGIN
<Function Body Code>
RETURN <Scalar Expression>
END[;]
```

Examples

The following example creates a scalar function to change the first character of a string to upper case.

```
CREATE FUNCTION dbo.UpperCaseFirstChar (@String VARCHAR(20))
```

```
RETURNS VARCHAR(20)
AS
BEGIN
RETURN UPPER(LEFT(@String, 1)) + LOWER(SUBSTRING(@String, 2, 19))
END;
```

```
SELECT dbo.UpperCaseFirstChar ('mIxEdCasE');

Mixedcase
```

User-Defined Table-Valued Functions

Inline table-valued UDFs are similar to views or a Common Table Expressions (CTE) with the added benefit of parameters. You can use inline table-valued UDFs in FROM clauses as subqueries. Also, you can join inline table-valued UDFs to other source table rows using the APPLY and OUTER APPLY operators. In-line table-valued UDFs have many associated internal optimizer optimizations due to their simple, view-like characteristics.

Syntax

```
CREATE FUNCTION <Function Name> ([{<Parameter Name> [AS] <Data Type> [= <Default
Value>] [READONLY]} [,...n]])
RETURNS TABLE
[AS]
RETURN (<SELECT Query>)[;]
```

Examples

The following example creates a table-valued function to aggregate employee orders.

```
CREATE TABLE Orders
(
  OrderID INT NOT NULL PRIMARY KEY,
  EmployeeID INT NOT NULL,
  OrderDate DATETIME NOT NULL
);
```

```
INSERT INTO Orders (OrderID, EmployeeID, OrderDate)
VALUES
```

```
(1, 1, '20180101 13:00:05'),
(2, 1, '20180201 11:33:12'),
(3, 2, '20180112 10:22:35');
```

```
CREATE FUNCTION dbo.EmployeeMonthlyOrders
(@EmployeeID INT)
RETURNS TABLE AS
RETURN
(
  SELECT EmployeeID,
    YEAR(OrderDate) AS OrderYear,
    MONTH(OrderDate) AS OrderMonth,
    COUNT(*) AS NumOrders
  FROM Orders AS O
  WHERE EmployeeID = @EmployeeID
  GROUP BY EmployeeID,
    YEAR(OrderDate),
    MONTH(OrderDate)
);
```

```
SELECT *
FROM dbo.EmployeeMonthlyOrders (1)

EmployeeID  OrderYear  OrderMonth  NumOrders
1           2018       1           1
1           2018       2           1
```

Multi-Statement User-Defined Table-Valued Functions

Multi-statement table-valued UDFs, such as In-line UDFs, are also similar to views or CTEs with the added benefit of parameters. You can use multi-statement table-valued UDFs in `FROM` clauses as sub queries. Also, you can join multi-statement table-valued UDFs to other source table rows using the `APPLY` and `OUTER APPLY` operators.

The difference between multi-statement UDFs and the inline UDFs is that multi-statement UDFs aren't restricted to a single `SELECT` statement. They can consist of multiple statements including logic implemented with flow control, complex data processing, security checks, and so on.

The downside of using multi-statement UDFs is that there are far less optimizations possible and performance may suffer.

Syntax

```
CREATE FUNCTION <Function Name> ([{<Parameter Name> [AS] <Data Type> [= <Default
Value>] [READONLY]} [,...n]])
RETURNS <@Return Variable> TABLE <Table Definition>
[AS]
BEGIN
<Function Body Code>
RETURN
END[;]
```

For more information, see CREATE FUNCTION (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

For more information, see Stored Procedures.

Syntax

```
CREATE [ OR REPLACE ] FUNCTION
name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...]] )
[ RETURNS rettype
| RETURNS TABLE ( column_name column_type [, ...] ) ]
{ LANGUAGE lang_name
| TRANSFORM { FOR TYPE type_name } [, ... ]
| WINDOW
| IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
| CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
| [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
| PARALLEL { UNSAFE | RESTRICTED | SAFE }
| COST execution_cost
| ROWS result_rows
| SET configuration_parameter { TO value | = value | FROM CURRENT }
| AS 'definition'
| AS 'obj_file', 'link_symbol'
} ...
[ WITH ( attribute [, ...] ) ]
```

User-Defined Types

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Syntax and option differences.

SQL Server Usage

SQL Server user-defined types provide a mechanism for encapsulating custom data types and for adding NULL constraints.

SQL Server also supports table-valued user-defined types, which you can use to pass a set of values to a stored procedure.

User-defined types can also be associated to CLR code assemblies. Beginning with SQL Server 2014, memory optimized types support memory optimized tables and code.

> **Note**
>
> If your code uses custom rules bound to data types, Microsoft recommends discontinuing the use of this deprecated feature.

All user-defined types are based on an existing system data types. They allow developers to reuse the definition, making the code and schema more readable.

Syntax

The simplified syntax for the `CREATE TYPE` statement is shown following.

```
CREATE TYPE <type name> {
FROM <base type> [ NULL | NOT NULL ] | AS TABLE (<Table Definition>)}
```

User-Defined Types Examples

The following example creates a `ZipCode` scalar user-defined type.

```
CREATE TYPE ZipCode
FROM CHAR(5)
NOT NULL
```

The following example uses this `ZipCode` type in a table.

```
CREATE TABLE UserLocations
(UserID INT NOT NULL PRIMARY KEY, ZipCode ZipCode);

INSERT INTO [UserLocations] ([UserID],[ZipCode]) VALUES (1, '94324');
INSERT INTO [UserLocations] ([UserID],[ZipCode]) VALUES (2, NULL);
```

The code in the preceding example displays the following error message indicating that NULL values for `ZipCode` aren't allowed.

```
Msg 515, Level 16, State 2, Line 78
Can't insert the value NULL into column 'ZipCode', table 'tempdb.dbo.UserLocations';
column does not allow nulls. INSERT fails.
The statement has been terminated.
```

Table-Valued Types Examples

The following example demonstrates how to create and use a table-valued types to pass a set of values to a stored procedure.

Create the `OrderItems` table.

```
CREATE TABLE OrderItems
(
  OrderID INT NOT NULL,
  Item VARCHAR(20) NOT NULL,
  Quantity SMALLINT NOT NULL,
  PRIMARY KEY(OrderID, Item)
);
```

Create a table-valued type for the `OrderItems` table.

```
CREATE TYPE OrderItems
AS TABLE
```

```
(
  OrderID INT NOT NULL,
  Item VARCHAR(20) NOT NULL,
  Quantity SMALLINT NOT NULL,
  PRIMARY KEY(OrderID, Item)
);
```

Create the InsertOrderItems procedure. Note that the entire set of rows from the table-valued parameter is handled with one statement.

```
CREATE PROCEDURE InsertOrderItems
@OrderItems AS OrderItems READONLY
AS
BEGIN
  INSERT INTO OrderItems(OrderID, Item, Quantity)
  SELECT OrderID,
    Item,
    Quantity
  FROM @OrderItems;
END
```

Instantiate the OrderItems type, insert the values, and pass it to a stored procedure.

```
DECLARE @OrderItems AS OrderItems;

INSERT INTO @OrderItems ([OrderID], [Item], [Quantity])
VALUES
(1, 'M8 Bolt', 100),
(1, 'M8 Nut', 100),
(1, M8 Washer, 200);

EXECUTE [InsertOrderItems] @OrderItems = @OrderItems;

(3 rows affected)
```

Select all rows from the OrderItems table.

```
SELECT * FROM OrderItems;

OrderID  Item       Quantity
1        M8 Bolt    100
```

```
1         M8 Nut     100
1         M8 Washer  200
```

For more information, see [CREATE TYPE (Transact-SQL)](#) in the *SQL Server documentation*.

PostgreSQL Usage

Similar to SQL Server, PostgreSQL enables the creation of user-defined types using the CREATE TYPE statement.

A user-defined type is owned by the user who creates it. If a schema name is specified, the type is created under that schema.

PostgreSQL supports the creation of several different user-defined types: * Composite types store a single named attribute attached to a data type or multiple attributes as an attribute collection. In PostgreSQL, you can also use the CREATE TYPE statement standalone with an association to a table. * Enumerated types (enum) store a static ordered set of values. For example, product categories.

+

```
CREATE TYPE PRODUCT_CATEGORT AS ENUM
  ('Hardware', 'Software', 'Document');
```

- Range Types store a range of values, for example, a range of timestamps used to represent the ranges of time of when a course is scheduled.

  ```
  CREATE TYPE float8_range AS RANGE
    (subtype = float8, subtype_diff = float8mi);
  ```

  For more information, see [Range Types](#) in the *PostgreSQL documentation*.

- Base types are the system core types (abstract types) and are implemented in a low-level language such as C.
- Array types support definition of columns as multidimensional arrays. You can create an array column with a built-in type or a user-defined base type, enum type, or composite.

  ```
  CREATE TABLE COURSE_SCHEDULE (
    COURSE_ID NUMERIC PRIMARY KEY,
  ```

```
  COURSE_NAME VARCHAR(60),
  COURSE_SCHEDULES text[]);
```

For more information, see Arrays in the *PostgreSQL documentation*.

Syntax

```
CREATE TYPE name AS RANGE (
  SUBTYPE = subtype
  [ , SUBTYPE_OPCLASS = subtype_operator_class ]
  [ , COLLATION = collation ]
  [ , CANONICAL = canonical_function ]
  [ , SUBTYPE_DIFF = subtype_diff_function ]
)
CREATE TYPE name (
  INPUT = input_function,
  OUTPUT = output_function
  [ , RECEIVE = receive_function ]
  [ , SEND = send_function ]
  [ , TYPMOD_IN = type_modifier_input_function ]
  [ , TYPMOD_OUT = type_modifier_output_function ]
  [ , ANALYZE = analyze_function ]
  [ , INTERNALLENGTH = { internallength | VARIABLE } ]
  [ , PASSEDBYVALUE ]
  [ , ALIGNMENT = alignment ]
  [ , STORAGE = storage ]
  [ , LIKE = like_type ]
  [ , CATEGORY = category ]
  [ , PREFERRED = preferred ]
  [ , DEFAULT = default ]
  [ , ELEMENT = element ]
  [ , DELIMITER = delimiter ]
  [ , COLLATABLE = collatable ]
)
```

Examples

The following example creates a user-defined type for storing an employee phone numbers.

```
CREATE TYPE EMP_PHONE_NUM AS (
  PHONE_NUM VARCHAR(11));
```

```
CREATE TABLE EMPLOYEES (
  EMP_ID NUMERIC PRIMARY KEY,
  EMP_PHONE EMP_PHONE_NUM NOT NULL);

INSERT INTO EMPLOYEES VALUES(1, ROW('111-222-333'));

SELECT a.EMP_ID, (a.EMP_PHONE).PHONE_NUM FROM EMPLOYEES a;

emp_id  phone_num
1       111-222-333
(1 row)
```

The following example creates a PostgreSQL Object Type as a collection of Attributes for the employees table.

```
CREATE OR REPLACE TYPE EMP_ADDRESS AS OBJECT (
  STATE VARCHAR(2),
  CITY VARCHAR(20),
  STREET VARCHAR(20),
  ZIP_CODE NUMERIC);

CREATE TABLE EMPLOYEES (
  EMP_ID NUMERIC PRIMARY KEY,
  EMP_NAME VARCHAR(10) NOT NULL,
  EMP_ADDRESS EMP_ADDRESS NOT NULL);

INSERT INTO EMPLOYEES
  VALUES(1, 'John Smith',
  ('AL', 'Gulf Shores', '3033 Joyce Street', '36542'));

SELECT a.EMP_NAME,
  (a.EMP_ADDRESS).STATE,
  (a.EMP_ADDRESS).CITY,
  (a.EMP_ADDRESS).STREET,
  (a.EMP_ADDRESS).ZIP_CODE
FROM EMPLOYEES a;

emp_name    state  city         street             zip_code
John Smith  AL     Gulf Shores  3033 Joyce Street  36542
```

For more information, see CREATE TYPE and Composite Types in the *PostgreSQL documentation*.

Sequences and Identity

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Less options with `SERIAL`. Reseeding needs to be rewritten.

SQL Server Usage

Automatic enumeration functions and columns are common with relational database management systems and are often used for generating surrogate keys.

SQL Server provides several features that support automatic generation of monotonously increasing value generators.

- `IDENTITY` property of a table column.
- `SEQUENCE` objects framework.
- Numeric functions such as `IDENTITY` and `NEWSEQUENTIALID`.

Identity

The `IDENTITY` property is probably the most widely used means of generating surrogate primary keys in SQL Server applications. Each table may have a single numeric column assigned as an `IDENTITY`, using the `CREATE TABLE` or `ALTER TABLE` DDL statements. You can explicitly specify a starting value and increment.

> **Note**
>
> The identity property doesn't enforce uniqueness of column values, indexing, or any other property. Additional constraints such as primary or unique keys, explicit index specifications, or other properties must be specified in addition to the `IDENTITY` property.

The IDENTITY value is generated as part of the transaction that inserts table rows. Applications can obtain IDENTITY values using the @@IDENTITY, SCOPE_IDENTITY, and IDENT_CURRENT functions.

You can manage IDENTITY columns using the DBCC CHECKIDENT command, which provides functionality for reseeding and altering properties.

Syntax

```
IDENTITY [(<Seed Value>, <Increment Value>)]
```

Examples

The following example creates a table with an IDENTITY column.

```
CREATE TABLE MyTABLE
(
  Col1 INT NOT NULL
  PRIMARY KEY NONCLUSTERED IDENTITY(1,1),
  Col2 VARCHAR(20) NOT NULL
);
```

The following example inserts a row and retrieve the generated IDENTITY value.

```
DECLARE @LastIdent INT;
INSERT INTO MyTable(Col2)
VALUES('SomeString');
SET @LastIdent = SCOPE_IDENTITY()
```

The following example creates a table with a non-key IDENTITY column and an increment of 10.

```
CREATE TABLE MyTABLE
(
  Col1 VARCHAR(20) NOT NULL
  PRIMARY KEY,
  Col2 INT NOT NULL
  IDENTITY(1,10),
);
```

The following example creates a table with a compound primary key including an IDENTITY column.

```
CREATE TABLE MyTABLE
(
  Col1 VARCHAR(20) NOT NULL,
  Col2 INT NOT NULL
  IDENTITY(1,10),
  PRIMARY KEY (Col1, Col2)
);
```

SEQUENCE

Sequences are objects that are independent of a particular table or column and are defined using the `CREATE SEQUENCE` DDL statement. You can manage sequences using the `ALTER SEQUENCE` statement. Multiple tables and multiple columns from the same table may use the values from one or more `SEQUENCE` objects.

You can retrieve a value from a `SEQUENCE` object using the `NEXT VALUE FOR` function. For example, a `SEQUENCE` value can be used as a default value for a surrogate key column.

`SEQUENCE` objects provide several advantages over `IDENTITY` columns:

- You can use `SEQUENCE` objects to obtain a value before the actual `INSERT` takes place.
- You can share value series among columns and tables.
- Easier management, restart, and modification of sequence properties.
- Allows assignment of value ranges using `sp_sequence_get_range` and not just per-row values.

Syntax

```
CREATE SEQUENCE <Sequence Name> [AS <Integer Data Type> ]
START WITH <Seed Value>
INCREMENT BY <Increment Value>;
```

```
ALTER SEQUENCE <Sequence Name>
RESTART [WITH <Reseed Value>]
INCREMENT BY <New Increment Value>;
```

Examples

The following example creates sequence and uses it for a primary key default.

```
CREATE SEQUENCE MySequence AS INT START WITH 1 INCREMENT BY 1;
CREATE TABLE MyTable
(
  Col1 INT NOT NULL
  PRIMARY KEY NONCLUSTERED DEFAULT (NEXT VALUE FOR MySequence),
  Col2 VARCHAR(20) NULL
);
```

```
INSERT MyTable (Col1, Col2) VALUES (DEFAULT, 'cde'), (DEFAULT, 'xyz');
```

```
SELECT * FROM MyTable;
```

```
Col1  Col2
1     cde
2     xyz
```

Identity

SQL Server provides two sequential generation functions: `IDENTITY` and `NEWSEQUENTIALID`.

> **Note**
>
> The IDENTITY function should not be confused with the IDENTITY property of a column.

You can use the IDENTITY function only in a `SELECT … INTO` statement to insert `IDENTITY` column values into a new table.

The `NEWSEQUNTIALID` function generates a hexadecimal GUID, which is an integer. While the `NEWID` function generates a random GUID, the `NEWSEQUENTIALID` function guarantees that every GUID created is greater (in numeric value) than any other GUID previously generated by the same function on the same server since the operating system restart.

You can use `NEWSEQUENTIALID` only with `DEFAULT` constraints associated with columns having a `UNIQUEIDENTIFIER` data type.

Syntax

```
IDENTITY (<Data Type> [, <Seed Value>, <Increment Value>]) [AS <Alias>]
```

```
NEWSEQUENTIALID()
```

Examples

The following example uses the `IDENTITY` function as surrogate key for a new table based on an existing table.

```
CREATE TABLE MySourceTable
(
  Col1 INT NOT NULL PRIMARY KEY,
  Col2 VARCHAR(10) NOT NULL,
  Col3 VARCHAR(10) NOT NULL
);
```

```
INSERT INTO MySourceTable
VALUES
(12, 'String12', 'String12'),
(25, 'String25', 'String25'),
(95, 'String95', 'String95');
```

```
SELECT IDENTITY(INT, 100, 1) AS SurrogateKey,
  Col1,
  Col2,
  Col3
INTO MyNewTable
FROM MySourceTable
ORDER BY Col1 DESC;
```

```
SELECT *
FROM MyNewTable;
```

```
SurrogateKey  Col1  Col2      Col3
100           95    String95  String95
101           25    String25  String25
102           12    String12  String12
```

The following example uses `NEWSEQUENTIALID` as a surrogate key for a new table.

```
CREATE TABLE MyTable
```

```
(
  Col1 UNIQUEIDENTIFIER NOT NULL
  PRIMARY KEY NONCLUSTERED DEFAULT NEWSEQUENTIALID()
);
```

```
INSERT INTO MyTable
DEFAULT VALUES;
```

```
SELECT *
FROM MyTable;
```

```
Col1

9CC01320-C5AA-E811-8440-305B3A017068
```

For more information, see Sequence Numbers and CREATE TABLE (Transact-SQL) IDENTITY (Property) in the *SQL Server documentation*.

PostgreSQL Usage

The PostgreSQL `CREATE SEQUENCE` command is mostly compatible with the SQL Server `CREATE SEQUENCE` command. Sequences in PostgreSQL serve the same purpose as in SQL Server; they generate numeric identifiers automatically. A sequence object is owned by the user that created it.

Sequence Parameters

- `TEMPORARY` or `TEMP` — PostgreSQL can create a temporary sequence within a session. Once the session ends, the sequence is automatically dropped.
- `IF NOT EXISTS` — Creates a sequence. If a sequence with an identical name already exists, it is replaced.
- `INCREMENT BY` — An optional parameter with a default value of 1. Positive values generate sequence values in ascending order. Negative values generate sequence values in descending sequence.
- `START WITH` — An optional parameter having a default of 1. It uses the MINVALUE for ascending sequences and the MAXVALUE for descending sequences.
- `MAXVALUE` | `NO MAXVALUE` — Defaults are between 263 for ascending sequences and -1 for descending sequences.

- `MINVALUE` | `NO MINVALUE` — Defaults are between 1 for ascending sequences and -263 for descending sequences.
- `CYCLE` | `NO CYCLE` — If the sequence value reaches `MAXVALUE` or `MINVALUE`, the `CYCLE` parameter instructs the sequence to return to the initial value (`MINVALUE` or `MAXVALUE`). The default is `NO CYCLE`.
- `CACHE` — In PostgreSQL, the `NOCACHE` isn't supported. By default, when the `CACHE` parameter isn't specified, no sequence values are pre-cached into memory (equivalent to the SQL Server `NOCACHE` parameter). The minimum value is 1.
- `OWNED BY` | `OWNBY NON` — Specifies that the sequence object is to be associated with a specific column in a table. When dropping this type of sequence, an error is returned due to the sequence/table association.
- `AS data_type` — This option is available in PostgreSQL version 10 and higher. To easily determine the minimum and maximum values and also improve storage management, you can select the data type for the sequence. The available data types are smallint, integer, and bigint. The default data type is bigint.

Syntax

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name
[ INCREMENT [ BY ] increment ]
[ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
[ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
[ OWNED BY { table_name.column_name | NONE } ]
```

Most SQL Server `CREATE SEQUENCE` parameters are compatible with PostgreSQL.

Examples

The following example creates a sequence.

```
CREATE SEQUENCE SEQ_1 START WITH 100
  INCREMENT BY 1 MAXVALUE 99999999999 CACHE 20 NO CYCLE;
```

The following example drops a sequence.

```
DROP SEQUENCE SEQ_1;
```

View sequences created in the current schema and sequence specifications.

```
SELECT * FROM INFORMATION_SCHEMA.SEQUENCES;
OR
\ds
```

The following example uses a PostgreSQL sequence as part of a `CREATE TABLE` and an `INSERT` statement.

```
CREATE TABLE SEQ_TST
(COL1 NUMERIC DEFAULT NEXTVAL('SEQ_1') PRIMARY KEY, COL2 VARCHAR(30));
INSERT INTO SEQ_TST (COL2) VALUES('A');
SELECT * FROM SEQ_TST;

col1  col2
100   A
```

Use the OWNED BY parameter to associate the sequence with a table.

```
CREATE SEQUENCE SEQ_1 START WITH 100 INCREMENT BY 1 OWNED BY SEQ_TST.COL1;
```

Query the current value of a sequence.

```
SELECT CURRVAL('SEQ_1);
```

Manually increment a sequence value according to the `INCREMENT BY` value.

```
SELECT NEXTVAL('SEQ_1');
OR
SELECT SETVAL('SEQ_1', 200);
```

Alter an existing sequence.

```
ALTER SEQUENCE SEQ_1 MAXVALUE 1000000;
```

IDENTITY Usage

Starting from PostgreSQL 10, there is a new option called identity columns which is similar to the `SERIAL` data type but more SQL standard compliant. The identity columns are slightly more compatible compared to SQL Server identity columns.

To create a table with identity columns, use the following statement:

```
CREATE TABLE emps (
  emp_id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  emp_name VARCHAR(35) NOT NULL);

INSERT INTO emps (emp_name) VALUES ('Robert');
INSERT INTO emps (emp_id, emp_name) VALUES (DEFAULT, 'Brian');

SELECT * FROM emps;

col1  col2
1     Robert
2     Brian
```

In PostgreSQL, for `SERIAL` and `IDENTITY`, you can insert any value, so long as it won't violate the primary key constraint. If the value violates the primary key constraint and you use the identity column sequence value again, the following error might be raised:

```
SQL Error [23505]: ERROR: duplicate key value violates unique constraint
 "emps_iden_pkey"
Detail: Key (emp_id)=(2) already exists.
```

SERIAL Usage

In PostgreSQL, you can create a sequence similar to the `IDENTITY` property supported by identity columns. When you create a new table, the sequence is created through the `SERIAL` pseudo-type. Other types from the same family are `SMALLSERIAL` and `BIGSERIAL`.

By assigning a `SERIAL` type to a column during table creation, PostgreSQL creates a sequence using the default configuration and adds a NOT NULL constraint to the column. The newly created sequence behaves like a regular sequence (incremented by 1) and no composite `SERIAL` option.

The following example uses `SERIAL` sequence.

```
CREATE TABLE SERIAL_SEQ_TST(COL1 SERIAL PRIMARY KEY, COL2 VARCHAR(10));

INSERT INTO SERIAL_SEQ_TST(COL2) VALUES('A');
SELECT * FROM SERIAL_SEQ_TST;

col1  col2
```

```
1     A

\ds

Schema  Name                     Type      Owner
public  serial_seq_tst_col1_seq  sequence  pg_tst_db
```

The following example uses the PostgreSQL SERIAL pseudo-type with a sequence that is created implicitly.

```
CREATE TABLE SERIAL_SEQ_TST(COL1 SERIAL PRIMARY KEY, COL2 VARCHAR(10));

\ds

Schema  Name                     Type      Owner
public  serial_seq_tst_col1_seq  sequence  pg_tst_db

ALTER SEQUENCE SERIAL_SEQ_TST_COL1_SEQ RESTART WITH 100 INCREMENT BY 10;
INSERT INTO SERIAL_SEQ_TST(COL2) VALUES('A');
INSERT INTO SERIAL_SEQ_TST(COL1, COL2) VALUES(DEFAULT, 'B');
SELECT * FROM SERIAL_SEQ_TST;

col1  col2
100   A
110   B
```

Use the ALTER SEQUENCE command to change the default sequence configuration in a SERIAL column.

Create a table with a SERIAL column that uses increments of 10:

```
CREATE TABLE SERIAL_SEQ_TST(COL1 SERIAL PRIMARY KEY, COL2 VARCHAR(10));

ALTER SEQUENCE serial_seq_tst_col1_seq INCREMENT BY 10;
```

Note

The auto generated sequence's name should be created with the following format: TABLENAME_COLUMNNAME_seq.

Create a table with a compound primary key including a SERIAL column:

```
CREATE TABLE SERIAL_SEQ_TST
(COL1 SERIAL, COL2 VARCHAR(10), PRIMARY key (COL1,COL2));
```

Summary

The following table identifies similarities, differences, and key migration considerations.

Feature	SQL Server	Aurora PostgreSQL
Independent SEQUENCE object	CREATE SEQUENCE	CREATE SEQUENCE
Automatic enumerator column property	IDENTITY	SERIAL or IDENTITY
Reseed sequence value	DBCC CHECKIDENT	1. Find sequence name: pg_get_serial_sequence('[table_name]', '[serial_field_name]') 2. SELECT SETVALSELECT pg_get_serial_sequence('table_name', 'person_id', 1, false);
Column restrictions	Numeric	Numeric
Controlling seed and interval values	CREATE/ALTER SEQUENCE	CREATE/ALTER SEQUENCE
Sequence setting initialization	Maintained through service restarts	ALTER SEQUENCE

Feature	SQL Server	Aurora PostgreSQL
Explicit values to column	Not allowed by default, `SET IDENTITY_INSERT ON` required	Allowed

For more information, see CREATE SEQUENCE, Sequence Manipulation Functions, Numeric Types, and CREATE TABLE in the *PostgreSQL documentation*.

Configuration

Topics

- Upgrades
- Session Options
- Database Options
- Server Options in SQL Server and Parameter Groups in Amazon Aurora

Upgrades

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
N/A	N/A	N/A	N/A

SQL Server Usage

As a database administrator, from time to time a database upgrade is required. It can be either for security fix, bugs fixes, compliance, or new database features.

You can plan the database upgrade to minimize the database downtime and risk. You can perform an upgrade in-place or migrate to a new installation.

Upgrade in-place

With this approach, we are retaining the current hardware and OS version by adding the new SQL Server binaries on the same server and then upgrade the SQL Server instance.

Before upgrading the database engine, review the SQL Server release notes for the intended target release version for any limitations and known issues to help you plan the upgrade.

In general, these will be the steps to perform the upgrade:

Prerequisite steps

- Back up all SQL Server database files, so that you can restore them if required.

- Run the appropriate Database Console Commands (DBCC CHECKDB) on databases to be upgraded to ensure that they are in a consistent state.
- Ensure to allocate enough disk space for SQL Server components, in addition to user databases.
- Disable all startup stored procedures as stored procedures processed at startup time might block the upgrade process.
- Stop all applications, including all services that have SQL Server dependencies.

Steps for upgrade

- Install new software.
  - Fix issues raised.
  - Set if you prefer to have automatic updates or not.
  - Select products install to upgrade, this is the new binaries installation.
  - Monitor the progress of downloading, extracting, and installing the Setup files.
- Specify the instance of SQL Server to upgrade.
  - On the Select Features page, the features to upgrade will be preselected. The prerequisites for the selected features are displayed on the right-hand pane. SQL Server Setup will install the prerequisite that aren’t already installed during the installation step described later in this procedure.
- Review upgrade plan before the actual upgrade.
- Monitor installation progress.

Post upgrade tasks

- Review summary log file for the installation and other important notes.
- Register your servers.

Migrate to a new installation

This approach maintains the current environment while building a new SQL Server environment. This is usually done when migrating on a new hardware and with a new version of the operating system. In this approach migrate the system objects so that they are same as the existing environment, then migrate the user database either using backup and restore.

For more information, see [Upgrade Database Engine]() in the *SQL Server documentation*.

PostgreSQL Usage

After migrating your databases to Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL), you will still need to upgrade your database instance from time to time, for the same reasons you have done in the past, new features, bugs and security fixes.

In a managed service like Amazon Relational Database Service, the upgrade process is much easier and simpler compared to the on-premises Oracle process.

To determine the current Aurora PostgreSQL version being used, use the following AWS CLI command:

```
aws rds describe-db-engine-versions --engine aurora-postgresql --query '*[].
[EngineVersion]' --output text --region your-AWS-Region
```

This can also be queried from the database, using the following queries:

```
SELECT AURORA_VERSION();

aurora_version
4.0.0

SHOW SERVER_VERSION;

server_version
12.4
```

For all Aurora and PostgreSQL versions mapping, see [Amazon Aurora PostgreSQL releases and engine versions]() in the *User Guide for Aurora*.

AWS doesn't apply major version upgrades on Amazon Aurora automatically. Major version upgrades contains new features and functionality which often involves system table and other code changes. These changes may not be backward-compatible with previous versions of the database so application testing are highly recommended.

Applying automatic minor upgrades can be set by configuring the Amazon Relational Database Service (Amazon RDS) instance to allow it.

You can use the following AWS CLI command on Linux to determine the current automatic upgrade minor versions.

```
aws rds describe-db-engine-versions --engine aurora-postgresql | grep -A 1 AutoUpgrade|
 grep -A 2 true |grep PostgreSQL | sort --unique | sed -e 's/"Description": "//g'
```

If no results are returned, there is no automatic minor version upgrade available and scheduled.

When enabled, the instance will be automatically upgraded during the scheduled maintenance window.

For major upgrades, this is the recommended process:

- Have a version-compatible parameter group ready. If you are using a custom DB instance or DB cluster parameter group, you have two options:
  1. Specify the default DB instance, DB cluster parameter group, or both for the new DB engine version.
  2. Create your own custom parameter group for the new DB engine version.

> **Note**
>
> If you associate a new DB instance or DB cluster parameter group as a part of the upgrade request, make sure to reboot the database after the upgrade completes to apply the parameters. If a DB instance needs to be rebooted to apply the parameter group changes, the instance's parameter group status shows pending-reboot. You can view an instance's parameter group status in the console or by using a CLI command such as `describe-db-instances` or `describe-db-clusters`.

- Check for unsupported usage:
  1. Commit or roll back all open prepared transactions before attempting an upgrade. You can use the following query to verify that there are no open prepared transactions on your instance.

     ```
     SELECT count(*) FROM pg_catalog.pg_prepared_xacts;
     ```

  2. Remove all uses of the reg* data types before attempting an upgrade. Except for `regtype` and `regclass`, you can't upgrade the reg* data types. The `pg_upgrade` utility can't persist

this data type, which is used by Amazon Aurora to do the upgrade. To verify that there are no uses of unsupported reg* data types, use the following query for each database.

```
SELECT count(*) FROM pg_catalog.pg_class c, pg_catalog.pg_namespace n, pg_
catalog.pg_attribute aWHERE c.oid = a.attrelid
AND NOT a.attisdropped
AND a.atttypid IN ('pg_catalog.regproc'::pg_catalog.regtype,
'pg_catalog.regprocedure'::pg_catalog.regtype,
'pg_catalog.regoper'::pg_catalog.regtype,
'pg_catalog.regoperator'::pg_catalog.regtype,
'pg_catalog.regconfig'::pg_catalog.regtype,
'pg_catalog.regdictionary'::pg_catalog.regtype)
AND c.relnamespace = n.oid
AND n.nspname NOT IN ('pg_catalog', 'information_schema');
```

- Perform a backup. The upgrade process creates a DB cluster snapshot of your DB cluster during upgrading.
- Upgrade certain extensions to the latest available version before performing the major version upgrade. The extensions to update include the following:
  1. pgRouting
  2. postGIS
- Run the following command for each extension that you are using.

```
ALTER EXTENSION PostgreSQL-extension UPDATE TO 'new-version'
```

If you are upgrading versions older than PostgreSQL 12, there are a few more steps. For more information, see Upgrading the PostgreSQL DB engine for Aurora PostgreSQL in the *User Guide for Aurora*.

You can perform the actual upgrade through the console or AWS CLI.

Console

1. Sign in to the AWS Management Console and choose **RDS**.
2. In the navigation pane, choose **Databases**, and then choose the DB cluster that you want to upgrade.
3. Choose **Modify**. The **Modify DB cluster** page appears.

4. For **DB engine version**, choose the new version.
5. Choose **Continue** and check the summary of modifications.
6. To apply the changes immediately, choose **Apply immediately**. Choosing this option can cause an outage in some cases. For more information, see Modifying an Amazon Aurora DB cluster in the *User Guide for Aurora*.
7. On the confirmation page, review your changes. If they are correct, choose **Modify Cluster** to save your changes. Or choose **Back** to edit your changes or **Cancel** to cancel your changes.

AWS CLI

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
--db-cluster-identifier mydbcluster \
--engine-version new_version \
--allow-major-version-upgrade \
--no-apply-immediately
```

For Microsoft Windows:

```
aws rds modify-db-cluster ^
--db-cluster-identifier mydbcluster ^
--engine-version new_version ^
--allow-major-version-upgrade ^
--no-apply-immediately
```

Summary

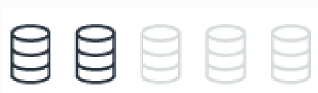
Phase	SQL Server Step	Aurora PostgreSQL
Prerequisite	Perform an instance backup.	Run Amazon RDS instance backup.
Prerequisite	DBCC for consistent verificat ion.	N/A
Prerequisite	Validate disk size and free space.	N/A

<table>
<tr><th>Phase</th><th>SQL Server Step</th><th>Aurora PostgreSQL</th></tr>
<tr><td>Prerequisite</td><td>Disable all startup stored procedures (if applicable).</td><td>N/A</td></tr>
<tr><td>Prerequisite</td><td>Stop application and connection.</td><td>N/A</td></tr>
<tr><td>Prerequisite</td><td>Install new software and fix prerequisites errors raised.</td><td>1. Remove all uses of the reg* data types.
2. Upgrade certain extension s.
3. Commit or roll back all open prepared transacti ons.
<pre>SELECT count(*) FROM
 pg_catalog.pg_prep
ared_xacts;</pre></td></tr>
<tr><td>Prerequisite</td><td>Select instances to upgrade.</td><td>Select the right Amazon RDS instance.</td></tr>
<tr><td>Prerequisite</td><td>Review pre-upgrade summary.</td><td>N/A</td></tr>
<tr><td>Runtime</td><td>Monitor upgrade progress.</td><td>You can review from the console.</td></tr>
<tr><td>Post-upgrade</td><td>Results.</td><td>You can review from the console.</td></tr>
<tr><td>Post-upgrade</td><td>Register server.</td><td>N/A</td></tr>
<tr><td>Post-upgrade</td><td>Test applications against the new upgraded database.</td><td>Test applications against the new upgraded database.</td></tr>
</table>

Phase	SQL Server Step	Aurora PostgreSQL
Production deployment	Re-run all steps in a productio n environment.	Re-run all steps in a productio n environment.

For more information, see Upgrading the PostgreSQL DB engine for Aurora PostgreSQL in the *User Guide for Aurora*.

Session Options

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	SET options are significantly different, except for transaction isolation control.

SQL Server Usage

Session options in SQL Server is a collection of run-time settings that control certain aspects of how the server handles data for individual sessions. A session is the period between a login event and a disconnect event or an `exec sp_reset_connection` command for connection pooling.

Each session may have multiple run scopes, which are all the statements before the GO keyword used in SQL Server Management Studio scripts, or any set of commands sent as a single run batch by a client application. Each run scope may contain additional sub-scopes. For example, scripts calling stored procedures or functions.

You can set the global session options, which all run scopes use by default, using the SET T-SQL command. Server code modules such as stored procedures and functions may have their own run context settings, which are saved along with the code to guarantee the validity of results.

Developers can explicitly use SET commands to change the default settings for any session or for an run scope within the session. Typically, client applications send explicit SET commands upon connection initiation.

You can view the metadata for current sessions using the `sp_who_system` stored procedure and the `sysprocesses` system table.

> **Note**
>
> To change the default setting for SQL Server Management Studio, choose **Tools**, **Options**, **Query Execution**, **SQL Server**, **Advanced**.

Syntax

Syntax for the SET command:

```
SET
Category Setting
Date and time    DATEFIRST | DATEFORMAT
Locking          DEADLOCK_PRIORITY | SET LOCK_TIMEOUT
Miscellaneous    CONCAT_NULL_YIELDS_NULL | CURSOR_CLOSE_ON_COMMIT | FIPS_FLAGGER | SET
 IDENTITY_INSERT | LANGUAGE | OFFSETS | QUOTED_IDENTIFIER
Query Execution  ARITHABORT | ARITHIGNORE | FMTONLY | NOCOUNT | NOEXEC |
 NUMERIC_ROUNDABORT | PARSEONLY | QUERY_GOVERNOR_COST_LIMIT | ROWCOUNT | TEXTSIZE
ANSI             ANSI_DEFAULTS | ANSI_NULL_DFLT_OFF | ANSI_NULL_DFLT_ON | ANSI_NULLS |
 ANSI_PADDING | ANSI_WARNINGS
Execution Stats  FORCEPLAN | SHOWPLAN_ALL | SHOWPLAN_TEXT | SHOWPLAN_XML | STATISTICS
 IO | STATISTICS XML | STATISTICS PROFILE | STATISTICS TIME
Transactions     IMPLICIT_TRANSACTIONS | REMOTE_PROC_TRANSACTIONS | TRANSACTION
 ISOLATION LEVEL | XACT_ABORT
```

For more information, see SET Statements (Transact-SQL) in the *SQL Server documentation*.

SET ROWCOUNT for DML Deprecated Setting

The SET ROWCOUNT for DML statements has been deprecated as of SQL Server 2008. For more information, see Deprecated Database Engine Features in SQL Server 2008 R2 in the *SQL Server documentation*.

For SSQL Server version 2008 R2 and lower, you could limit the number of rows affected by `INSERT`, `UPDATE`, and `DELETE` operations using `SET ROWCOUNT`. For example, it is a common practice in SQL Server to batch large `DELETE` or `UPDATE` operations to avoid transaction logging issues.

The following example loops and deletes rows where ForDelete is set to 1, but only 5000 rows at a time in separate transactions (assuming the loop isn't within an explicit transaction).

```
SET ROWCOUNT 5000;
WHILE @@ROWCOUNT > 0
BEGIN
  DELETE FROM MyTable
  WHERE ForDelete = 1;
END
```

Starting from SQL Server 2012, SET ROWCOUNT is ignored for INSERT, UPDATE, and DELETE statements. You can achieve the same functionality using TOP. You can convert TOP to the Aurora PostgreSQL LIMIT.

For example, you can rewrite the preceding code as:

```
WHILE @@ROWCOUNT > 0
BEGIN
DELETE TOP (5000)
  FROM MyTable
  WHERE ForDelete = 1;
END
```

AWS Schema Conversion Tool can convert this syntax automatically.

Examples

Use SET within a stored procedure.

```
CREATE PROCEDURE <ProcedureName>
AS
BEGIN
  <Some non-critical transaction code>
  SET TRANSACTION_ISOLATION_LEVEL SERIALIZABLE;
  SET XACT_ABORT ON;
  <Some critical transaction code>
END
```

Explicit SET commands affect their run scope and sub scopes. After the scope terminates and the procedure code exits, the calling scope resumes its original settings used before the calling the stored procedure.

For more information, see [SET Statements (Transact-SQL)]() in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) supports hundreds of Server System Variables to control server behavior and the global and session levels.

PostgreSQL provides session-modifiable parameters that are configured using the `SET SESSION` command. Configuration of parameters using `SET SESSION` will only be applicable in the current session. To view the list of parameters that can be set with `SET SESSION`, you can query `pg_settings`:

```
SELECT * FROM pg_settings where context = 'user';
```

Examples of commonly used session parameters:

- `client_encoding` configures the connected client character set.
- `force_parallel_mode` forces use of parallel query for the session.
- `lock_timeout` sets the maximum allowed duration of time to wait for a database lock to release.
- `search_path` sets the schema search order for object names that aren't schema-qualified.
- `transaction_isolation` sets the current Transaction Isolation Level for the session.

You can view Aurora PostgreSQL variables using the PostgreSQL command line utility, Amazon Aurora database cluster parameters, Amazon Aurora database instance parameters, or SQL Server interface system variables.

Converting from SQL Server 2008 SET ROWCOUNT for DML operations

The use of `SET ROWCOUNT` for DML operations is deprecated as of SQL Server 2008 R2. Code that uses the `SET ROWCOUNT` syntax can't be converted automatically.

You can either rewrite the code to use `TOP` before running AWS SCT, or manually change it afterward.

Consider the example that is used to batch `DELETE` operations in SQL Server using `TOP`:

```
WHILE @@ROWCOUNT > 0
BEGIN
```

```
  DELETE TOP (5000)
  FROM MyTable
  WHERE ForDelete = 1;
END
```

You can rewrite the preceding example to use the Aurora PostgreSQL `LIMIT` clause:

```
WHILE row_count() > 0 LOOP
  DELETE FROM num_test
  WHERE ctid IN (
  SELECT ctid
  FROM num_test
  LIMIT 10)
END LOOP;
```

Examples

Change the time zone of the connected session.

```
SET SESSION DateStyle to POSTGRES, DMY;
SET

SELECT NOW();

now
Sat 09 Sep 11:03:43.597202 2017 UTC
(1 row)

SET SESSION DateStyle to ISO, MDY;
SET

SELECT NOW();

now
2017-09-09 11:04:01.3859+00
(1 row)
```

Summary

The following table summarizes commonly used SQL Server session options and their corresponding Aurora PostgreSQL system variables.

Category	SQL Server	Aurora PostgreSQL
Date and time	DATEFIRST	Use DOW in queries
Date and time	DATEFORMAT	DateStyle
Locking	LOCK_TIMEOUT	lock_timeout
Transactions	IMPLICIT_TRANSACTIONS	SET TRANSACTION
Transactions	TRANSACTION ISOLATION LEVEL	BEGIN TRANSACTION ISOLATION LEVEL
Query run	IDENTITY_INSERT	See Sequences and Identity.
Query run	LANGUAGE	lc_monetary , lc_numeric , or lc_time
Query run	QUOTED_IDENTIFIER	N/A
Query run	NOCOUNT	N/A and not needed
Run stats	SHOWPLAN_ALL , TEXT, XML, STATISTICS IO , PROFILE, and TIME	See Run Plans.
Miscellaneous	CONCAT_NULL_YIELDS_NULL	N/A
Miscellaneous	ROWCOUNT	Use LIMIT within SELECT.

For more information, see SET in the *PostgreSQL documentation*.

Database Options

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Difference.

SQL Server Usage

SQL Server provides database level options that you can set using the `ALTER DATABASE … SET` command. You can use these settings to:

- Set default session options. For more information, see Session Options.
- Enable or disable database features such as `SNAPSHOT_ISOLATION`, `CHANGE_TRANCKING`, and `ENABLE_BROKER`.
- Configure high availability and disaster recovery options such as always on availability groups.
- Configure security access control such as restricting access to a single user, setting the database offline, or setting the database to read-only.

Syntax

Syntax for setting database options:

```
ALTER DATABASE { <database name> } SET { <option> [ ,...n ] };
```

Examples

Set a database to read-only and use ARITHABORT by default.

```
ALTER DATABASE Demo SET READ_ONLY, ARITHABORT ON;
```

Set a database to use automatic statistic creation.

```
ALTER DATABASE Demo SET AUTO_CREATE_STATISTICS ON;
```

Set a database offline immediately.

```
ALTER DATABASE DEMO SET OFFLINE WITH ROLLBACK IMMEDIATE;
```

For more information, see ALTER DATABASE SET options (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) supports `CREATE SCHEMA` and `CREATE DATABASE` statements.

As with SQL Server, Aurora PostgreSQL does have the concept of an instance hosting multiple databases, which in turn contain multiple schemas. Objects in Aurora PostgreSQL are referenced as a three-part name: <database>.<schema>.<object>.

Database options are related to the cluster-level parameters which are managed by the AWS Cluster Parameter Groups. You can find some SQL Server equivalent parameters at the instance level in the AWS Database Parameter Group.

Datable options are being compared to AWS Database Parameter Group and Server Options are being compared to AWS Cluster Parameter Group. For more information, see Server Options.

Server Options in SQL Server and Parameter Groups in Amazon Aurora

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Use Cluster and Database/Cluster Parameter.

SQL Server Usage

SQL Server provides server-level settings that affect all databases and all sessions. You can modify these settings using the `sp_configure` system stored procedure.

You can use server options to perform the following configuration tasks:

- Define hardware utilization such as memory management, affinity mask, priority boost, network packet size, and soft Non-Uniform Memory Access (NUMA).
- Alter run time global values such as recovery interval, remote login timeout, optimization for ad-hoc workloads, and cost threshold for parallelism.
- Enable and disable global features such as C2 Audit, OLE, procedures, CLR procedures, and allow trigger recursion.
- Configure global security settings such as server authentication mode, remote access, shell access with `xp_cmdshell`, CLR access level, and database chaining.
- Set default values for sessions such as user options, default language, backup compression, and fill factor.

Some settings require an explicit `RECONFIGURE` command to apply the changes to the server. High risk settings require `RECONFIGURE WITH OVERRIDE` for the changes to be applied. Some advanced options are hidden by default. To view and modify these settings, set `show advanced options` to 1 and run `sp_configure`.

> **Note**
>
> Server audits are managed with the T-SQL commands `CREATE` and `ALTER SERVER AUDIT`.

Syntax

```
EXECUTE sp_configure <option>, <value>;
```

Examples

Limit server memory usage to 4 GB.

```
EXECUTE sp_configure 'show advanced options', 1;
```

```
RECONFIGURE;
```

```
sp_configure 'max server memory', 4096;
```

```
RECONFIGURE;
```

Allow command shell access from T-SQL.

```
EXEC sp_configure 'show advanced options', 1;
```

```
RECONFIGURE;
```

```
EXEC sp_configure 'xp_cmdshell', 1;
```

```
RECONFIGURE;
```

View the current values.

```
EXECUTE sp_configure
```

For more information, see Server Configuration Options (SQL Server) in the *SQL Server documentation*.

PostgreSQL Usage

When running PostgreSQL databases as Amazon Aurora Clusters, Parameter Groups are used to change to cluster-level and database-level parameters.

Most of the PostgreSQL parameters are configurable in an Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) cluster, but some are disabled and can't be modified. Because Amazon Aurora clusters restrict access to the underlying operating system, modification to PostgreSQL parameters must be made using Parameter Groups.

Amazon Aurora is a cluster of database instances and, as a direct result, some of the PostgreSQL parameters apply to the entire cluster while other parameters apply only to a particular database instance.

Aurora PostgreSQL parameter class	Controlled by

Cluster-level parameters
 	
Single cluster parameter group for each Amazon Aurora Cluster.
	Managed by cluster parameter groups. For example,
	
	• The PostgreSQL `wal_buffers` parameter is controlled by a cluster parameter group.
	  • The PostgreSQL `autovacuum` parameter is controlled by a cluster parameter group.
	  • The `client_encoding` parameter is controlled by a cluster parameter group.

Database instance-level parameters You can associate every instance in an Amazon Aurora cluster with a unique database parameter group.	Managed by database parameter groups. For example, • The PostgreSQL `shared_buffers` memory cache configuration parameter is controlled by a database parameter group with an optimized default value based on the configured database class: `{DBInstanceClassMemory/10922}`. • The PostgreSQL `max_connections` parameter, which controls the maximum number of client connections allowed to the PostgreSQL instance, is controlled by a database parameter group. The default value is optimized by AWS based on the configured database class: `LEAST({DBInstanceClassMemory/9531392},5000)`. • The `authentication_timeout` parameter, which controls the maximum time to complete client authentication (in seconds), is controlled by a database parameter group.

Aurora PostgreSQL parameter class	Controlled by
	• The `superuser_reserved_connections` parameter, which determines the number of reserved connection slots for PostgreSQL superusers, is configured by a database parameter group. • The PostgreSQL `effective_cache_size`, which informs the query optimizer how much cache is present in the kernel and helps control how expensive large index scans will be, is controlled by a database level parameter group. The default value is optimized by AWS based on database class (RAM): `{DBInstanceClassMemory/10922}`.

New parameters in PostgreSQL 10:

1. `enable_gathermerge` enables the gather merge run plan.
2. `max_parallel_workers` stands for the maximum number of parallel workers process.
3. `max_sync_workers_per_subscription` stands for the maximum number of synchronous workers for subscription.
4. `wal_consistency_checking` checks consistency of WAL on the standby instance (can't be set in Aurora PostgreSQL).
5. `max_logical_replication_workers` stands for the maximum number of logical replication worker process.
6. `max_pred_locks_per_relation` stands for the maximum number of records that you can predicate-lock before locking the entire relation.
7. `max_pred_locks_per_page` stands for the maximum number of records that you can predicate-lock before locking the entire page.
8. `min_parallel_table_scan_size` stands for the minimum table size to consider parallel table scan.

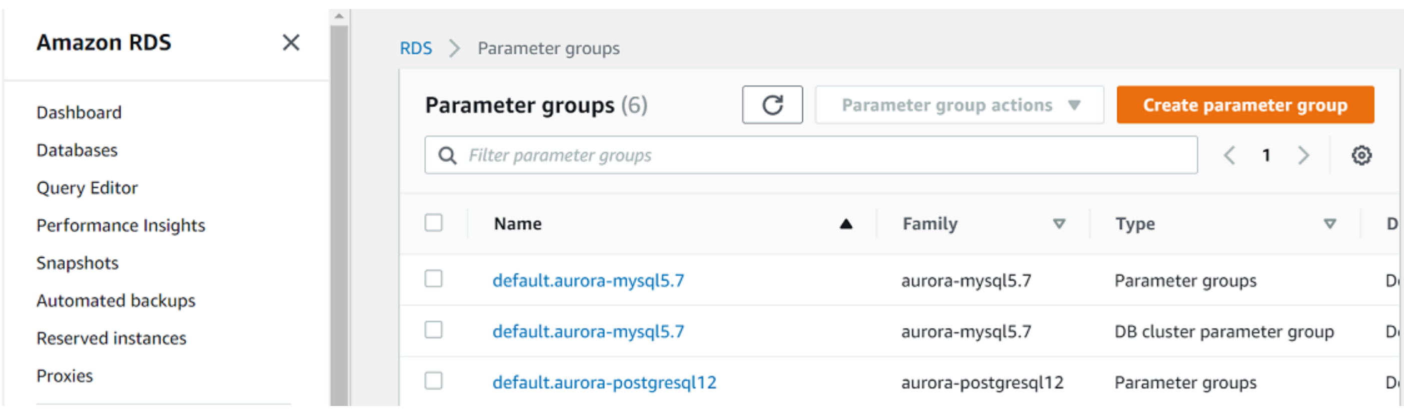
9. `min_parallel_index_scan_size` stands for the minimum table size to consider parallel index scan.

Examples

To create and configure a new parameter group

1. Sign in to the AWS Management Console and choose **RDS**.
2. Choose **Parameter groups**.

> **Note**
>
> You can’t edit the default parameter group. Create a custom parameter group to apply changes to your Amazon Aurora cluster and its database instances.



3. Select the DB family from the Parameter group family drop-down list.
4. For **Type**, select the DB parameter group.
5. Choose **Create**.

To modify an existing parameter group

1. Sign in to the AWS Management Console and choose **RDS**.
2. Choose **Parameter groups**.
3. Choose the name of the parameter to edit.
4. Choose **Edit parameters**.

5. Change parameter values and choose **Save changes**.

For more information, see Working with parameter groups in the *Amazon RDS User Guide*.

High Availability and Disaster Recovery

Topics

- [Backup and Restore]
- [High Availability Essentials]

Backup and Restore

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	Backup	Storage level backup managed by Amazon RDS.

SQL Server Usage

The term *backup* refers to both the process of copying data and to the resulting set of data created by the processes that copy data for safekeeping and disaster recovery. Backup processes copy SQL Server data and transaction logs to media such as tapes, network shares, cloud storage, or local files. You can copy these backups back to the database using a *restore* process.

SQL Server uses files, or filegroups, to create backups for an individual database or subset of a database. Table backups aren't supported.

When a database uses the FULL recovery model, transaction logs also need to be backed up. Transaction logs allow backing up only database changes since the last full backup and provide a mechanism for point-in-time restore operations.

Recovery model is a database-level setting that controls transaction log management. The three available recovery models are SIMPLE, FULL, and BULK LOGGED. For more information, see Recovery Models (SQL Server) in the *SQL Server documentation*.

The SQL Server `RESTORE` process copies data and log pages from a previously created backup back to the database. It then triggers a recovery process that rolls forward all committed transactions not yet flushed to the data pages when the backup took place. It also rolls back all uncommitted transactions written to the data files.

SQL Server supports the following types of backups:

- **Copy-only backups** are independent of the standard chain of SQL Server backups. They are typically used as one-off backups for special use cases and don't interrupt normal backup operations.
- **Data backups** copy data files and the transaction log section of the activity during the backup. A data backup may contain the whole database (database backup) or part of the database. The parts can be a partial backup, a file, or a filegroup.
- **A database backup** is a data backup representing the entire database at the point in time when the backup process finished.
- **A differential backup** is a data backup containing only the data structures (extents) modified since the last full backup. A differential backup is dependent on the previous full backup and can't be used alone.
- **A full backup** is a data backup containing a Database Backup and the transaction log records of the activity during the backup process.
- **Transaction log backups** don't contain data pages. They contain the log pages for all transaction activity since the last full backup or the previous transaction log backup.
- **File backups** consist of one or more files or filegroups.

SQL Server also supports media families and media sets that you can use to mirror and stripe backup devices. For more information, see Media Sets, Media Families, and Backup Sets (SQL Server) in the *SQL Server documentation*.

SQL Server 2008 Enterprise edition and later versions, support backup compression. Backup compression provides the benefit of a smaller backup file footprint, less I/O consumption, and less network traffic at the expense of increased CPU utilization for running the compression algorithm. For more information, see Backup Compression (SQL Server) in the *SQL Server documentation*.

A database backed up in the `SIMPLE` recovery mode can only be restored from a full or differential backup. For `FULL` and `BULK LOGGED` recovery models, you can restore transaction log backups to minimize potential data loss.

Restoring a database involves maintaining a correct sequence of individual backup restores. For example, a typical restore operation may include the following steps:

1. Restore the most recent full backup.
2. Restore the most recent differential backup.
3. Restore a set of uninterrupted transaction log backups, in order.
4. Recover the database.

For large databases, a full restore, or a complete database restore, from a full database backup isn't always a practical solution. SQL Server supports data file restore that restores and recovers a set of files and a single Data Page Restore, except for databases using the `SIMPLE` recovery model.

Syntax

SQL Server uses the following backup syntax.

```
Backing Up a Whole Database
BACKUP DATABASE <Database Name> [ <Files / Filegroups> ] [ READ_WRITE_FILEGROUPS ]
  TO <Backup Devices>
  [ <MIRROR TO Clause> ]
  [ WITH [DIFFERENTIAL ]
  [ <Option List> ][;]
```

```
BACKUP LOG <Database Name>
  TO <Backup Devices>
  [ <MIRROR TO clause> ]
  [ WITH <Option List> ][;]
```

```
<Option List> =
COPY_ONLY | {COMPRESSION | NO_COMPRESSION } | DESCRIPTION = <Description>
| NAME = <Backup Set Name> | CREDENTIAL | ENCRYPTION | FILE_SNAPSHOT | { EXPIREDATE =
<Expiration Date> | RETAINDAYS = <Retention> }
{ NOINIT | INIT } | { NOSKIP | SKIP } | { NOFORMAT | FORMAT } |
{ NO_CHECKSUM | CHECKSUM } | { STOP_ON_ERROR | CONTINUE_AFTER_ERROR }
{ NORECOVERY | STANDBY = <Undo File for Log Shipping> } | NO_TRUNCATE
ENCRYPTION ( ALGORITHM = <Algorithm> | SERVER CERTIFICATE = <Certificate> | SERVER
ASYMMETRIC KEY = <Key> );
```

SQL Server uses the following restore syntax.

```
RESTORE DATABASE <Database Name> [ <Files / Filegroups> ] | PAGE = <Page ID>
```

```
FROM <Backup Devices>
[ WITH [ RECOVERY | NORECOVERY | STANDBY = <Undo File for Log Shipping> } ]
[, <Option List>]
[;]
```

```
RESTORE LOG <Database Name> [ <Files / Filegroups> ] | PAGE = <Page ID>
[ FROM <Backup Devices>
[ WITH [ RECOVERY | NORECOVERY | STANDBY = <Undo File for Log Shipping> } ]
[, <Option List>]
[;]
```

```
<Option List> =
MOVE <File to Location>
| REPLACE | RESTART | RESTRICTED_USER | CREDENTIAL
| FILE = <File Number> | PASSWORD = <Password>
| { CHECKSUM | NO_CHECKSUM } | { STOP_ON_ERROR | CONTINUE_AFTER_ERROR }
| KEEP_REPLICATION | KEEP_CDC
| { STOPAT = <Stop Time>
| STOPATMARK = <Log Sequence Number>
| STOPBEFOREMARK = <Log Sequence Number>
```

Examples

Perform a full compressed database backup.

```
BACKUP DATABASE MyDatabase TO DISK='C:\Backups\MyDatabase\FullBackup.bak'
WITH COMPRESSION;
```

Perform a log backup.

```
BACKUP DATABASE MyDatabase TO DISK='C:\Backups\MyDatabase\LogBackup.bak'
WITH COMPRESSION;
```

Perform a partial differential backup.

```
BACKUP DATABASE MyDatabase
  FILEGROUP = 'FileGroup1',
  FILEGROUP = 'FileGroup2'
  TO DISK='C:\Backups\MyDatabase\DB1.bak'
```

```
  WITH DIFFERENTIAL;
```

Restore a database to a point in time.

```
RESTORE DATABASE MyDatabase
  FROM DISK='C:\Backups\MyDatabase\FullBackup.bak'
  WITH NORECOVERY;

RESTORE LOG AdventureWorks2012
  FROM DISK='C:\Backups\MyDatabase\LogBackup.bak'
  WITH NORECOVERY, STOPAT = '20180401 10:35:00';

RESTORE DATABASE AdventureWorks2012 WITH RECOVERY;
```

For more information, see Backup Overview (SQL Server) and Restore and Recovery Overview (SQL Server) in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) continuously backs up all cluster volumes and retains restore data for the duration of the backup retention period. The backups are incremental and you can use them to restore the cluster to any point in time within the backup retention period. You can specify a backup retention period from one to 35 days when creating or modifying a database cluster. Backups incur no performance impact and don't cause service interruptions.

Additionally, you can manually trigger data snapshots in a cluster volume that you can save beyond the retention period. You can use Snapshots to create new database clusters.

> **Note**
>
> Manual snapshots incur storage charges for Amazon Relational Database Service (Amazon RDS).

Restoring Data

You can recover databases from Amazon Aurora automatically retained data or from a manually saved snapshot. Using the automatically retained data significantly reduces the need to take frequent snapshots and maintain Recovery Point Objective (RPO) policies.

The Amazon Relational Database Service (Amazon RDS) console displays the available time frame for restoring database instances in the Latest Restorable Time and Earliest Restorable Time fields. The Latest Restorable Time is typically within the last five minutes. The Earliest Restorable Time is the end of the backup retention period.

> **Note**
>
> The Latest Restorable Time and Earliest Restorable Time fields display when a database cluster restore has been completed. Both display NULL until the restore process completes.

Database Cloning

Database cloning is a fast and cost-effective way to create copies of a database. You can create multiple clones from a single DB cluster. You can also create additional clones from existing clones. When first created, a cloned database requires only minimal additional storage space.

Database cloning uses a copy-on-write protocol. Data is copied only when it changes either on the source or cloned database.

Data cloning is useful for avoiding impacts on production databases. For example:

- Testing schema or parameter group modifications.
- Isolating intensive workloads. For example, exporting large amounts of data and running high resource-consuming queries.
- Development and testing with a copy of a production database.

Copying and sharing snapshots

You can copy and share database snapshots within the same AWS Region, across AWS Regions, and across AWS accounts. Snapshot sharing allows an authorized AWS account to access and copy snapshots. Authorized users can restore a snapshot from its current location without first copying it.

Copying an automated snapshot to another AWS account requires two steps:

- Create a manual snapshot from the automated snapshot.
- Copy the manual snapshot to another account.

Backup Storage

In all Amazon RDS regions, backup storage is the collection of both automated and manual snapshots for all database instances and clusters. The size of this storage is the sum of all individual instance snapshots.

When an Aurora PostgreSQL database instance is deleted, all automated backups of that database instance are also deleted. However, Amazon RDS provides the option to create a final snapshot before deleting a database instance. This final snapshot is retained as a manual snapshot. Manual snapshots aren’t automatically deleted.

The Backup Retention Period

Retention periods for Aurora PostgreSQL DB cluster backups are configured when creating a cluster. If not explicitly set, the default retention is one day when using the Amazon RDS API or the AWS CLI. The retention period is seven days if using the AWS Console. You can modify the backup retention period at any time with values of one to 35 days.

Disabling automated backups

You can’t disable automated backups on Aurora PostgreSQL. The backup retention period for Aurora PostgreSQL is managed by the database cluster.

Migration Considerations

Migrating from a self-managed backup policy to a Platform as a Service (PaaS) environment such as Aurora PostgreSQL is a complete paradigm shift. You no longer need to worry about transaction logs, file groups, disks running out of space, and purging old backups.

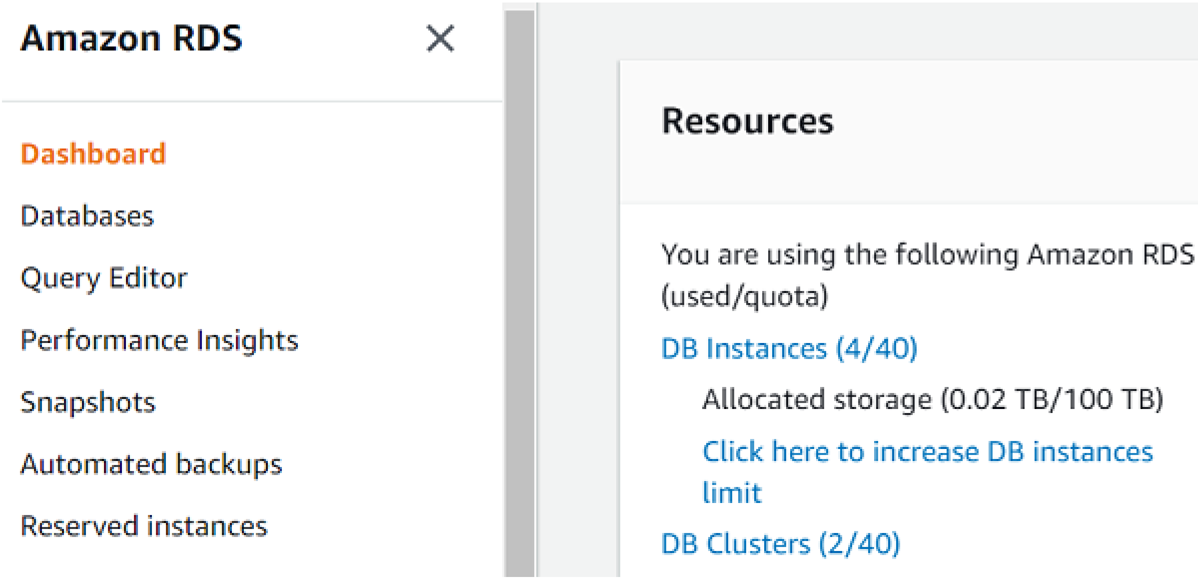
Amazon RDS provides guaranteed continuous backup with point-in-time restore up to 35 days.

Managing a SQL Server backup policy with similar RTO and RPO is a challenging task. With Aurora PostgreSQL, all you need to set is the retention period and take some manual snapshots for special use cases.

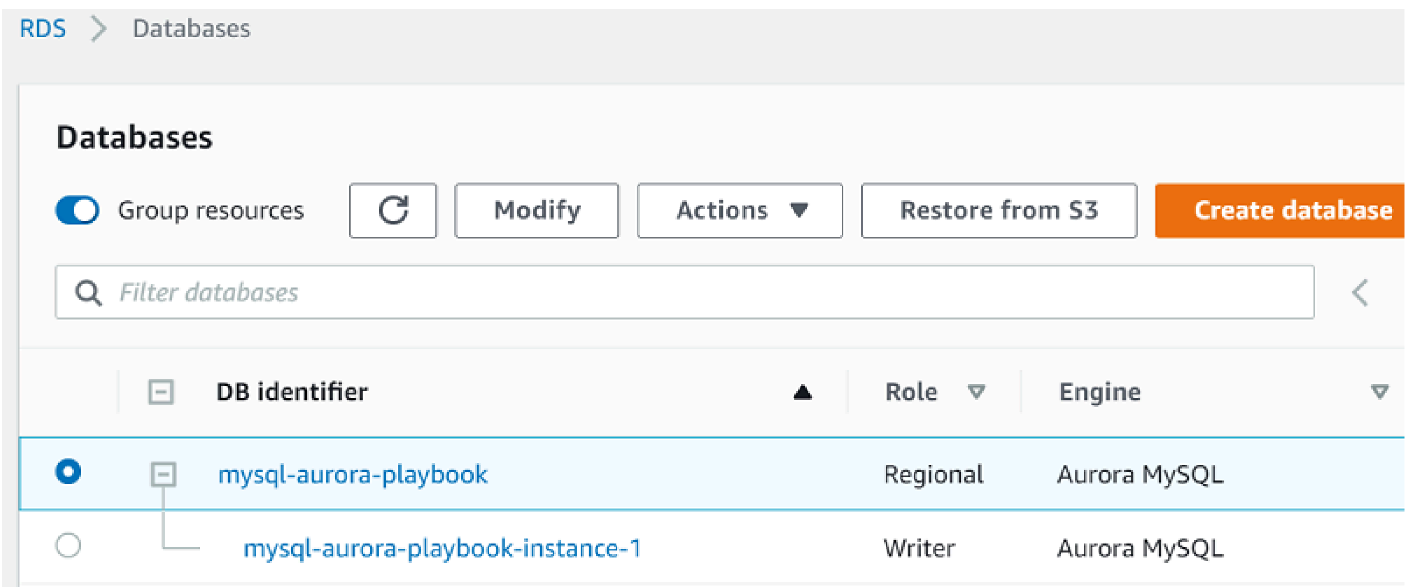
Examples

The following walkthrough describes how to change Aurora PostgreSQL DB cluster retention settings from one day to seven days using the Amazon RDS console.

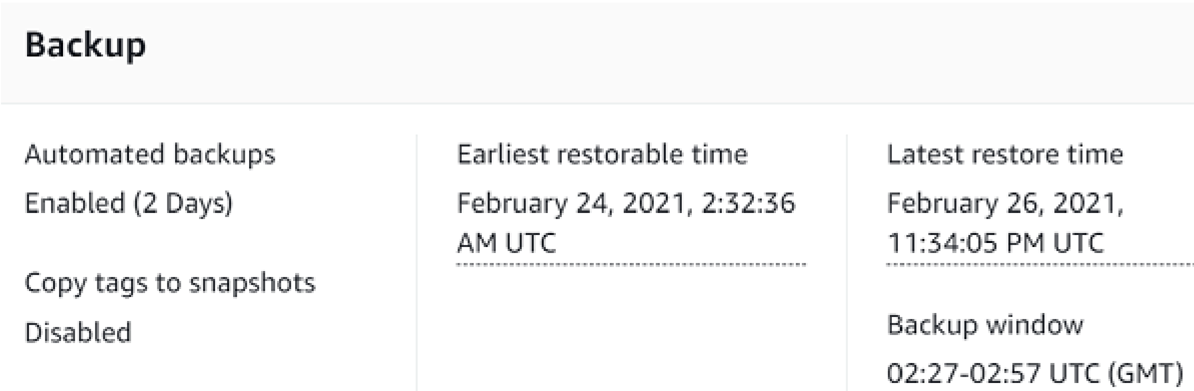
1. Log in to the Amazon RDS Console and on dashboard choose **Databases**.



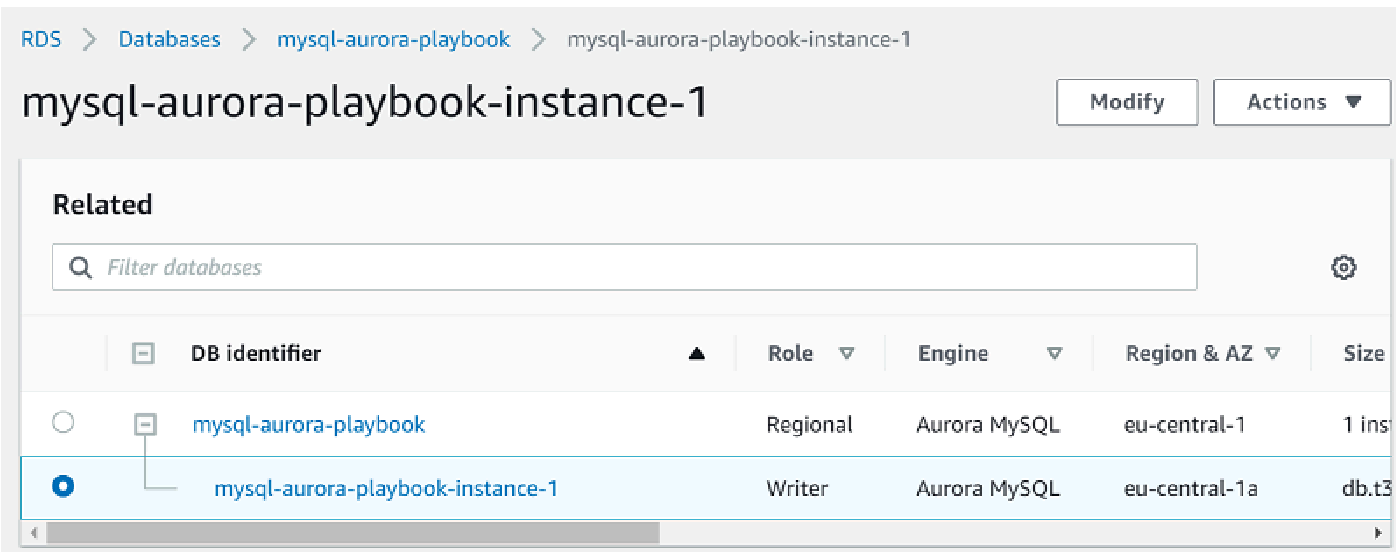
2. Choose the relevant DB identifier.



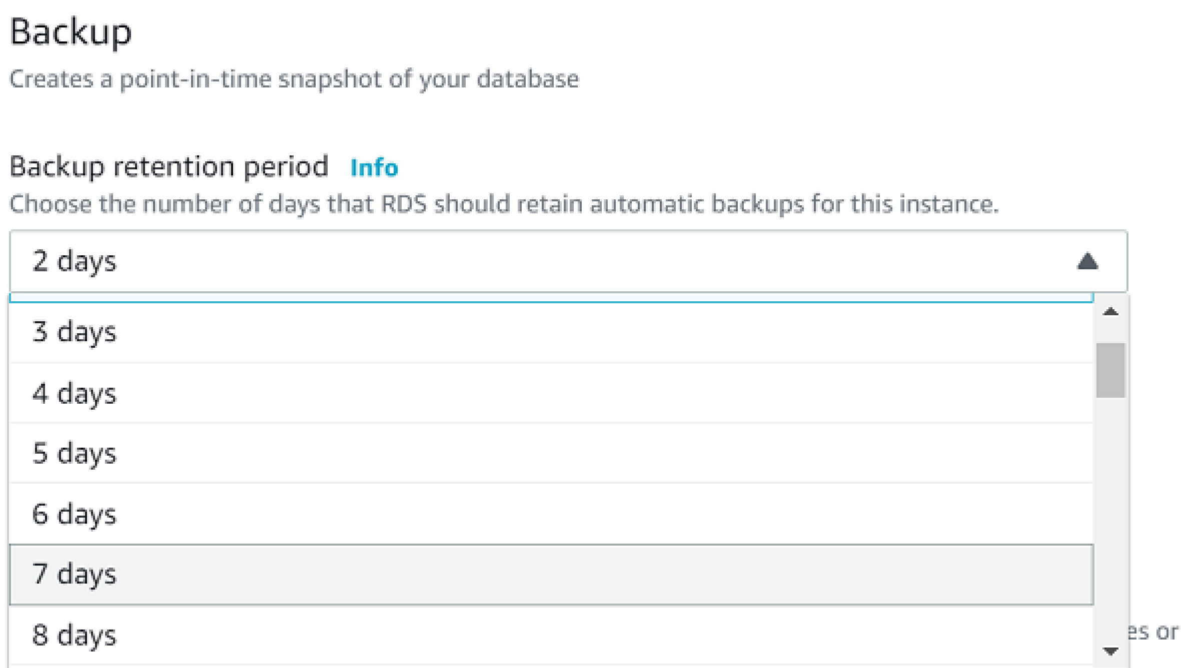
3. Verify the current automatic backup settings.



4. In this cluster, select database instance with the writer role.



5. On the top right, choose **Modify**.
6. For **Backup retention period**, choose *7 Days.



7. Choose **Continue** and review the summary.
8. For **When to apply modifications**, choose **Apply during the next scheduled maintenance window** to apply your changes during the next scheduled maintenance window. Or, choose **Apply immediately** to apply your changes immediately.
9. Choose **Modify DB instance**.

For more information, see Maintenance Plans.

Summary

Feature	SQL Server	Aurora PostgreSQL	Comments
Recovery Model	`SIMPLE`, `BULK LOGGED`, `FULL`	N/A	The functionality of Aurora PostgreSQL backups is equivalent to the FULL recovery model.
Backup database	`BACKUP DATABASE`	`aws rds create-db-clusters`	

Feature	SQL Server	Aurora PostgreSQL	Comments
		`napshot --db-cluster-snapshot-identifier Snapshot_name --db-cluster-identifier Cluster_Name`	
Partial backup	`BACKUP DATABASE … FILE= … | FILEGROUP = …`	N/A	Can use export utilities. For more information, see SQL Server Export and Import with Text Files and PostgreSQL pg_dump and pg_restore.
Log backup	`BACKUP LOG`	N/A	Backup is at the storage level.
Differential backups	`BACKUP DATABASE … WITH DIFFERENTIAL`	N/A	You can do manually using export tools.

Feature	SQL Server	Aurora PostgreSQL	Comments
Database snapshots	`BACKUP DATABASE … WITH COPY_ONLY`	Amazon RDS console or API.	The terminology is inconsistent between SQL Server and Aurora PostgreSQL. A database snapshot in SQL Server is similar to database cloning in Aurora PostgreSQL. Aurora PostgreSQL database snapshots are similar to a `COPY_ONLY` backup in SQL Server.

Feature	SQL Server	Aurora PostgreSQL	Comments
Database clones	`CREATE DATABASE… AS SNAPSHOT OF…`	Create new cluster from a cluster snapshot: `aws rds restore-db-cluster-from-snapshot --db-clusteridentifier NewCluster --snapshot-identifier SnapshotToRestore --engine aurora-postgresql .` Add a new instance to the new or restored cluster: `aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --dbcluster-identifier clustername-restore --db-instanceidentifier newinstancenodeA --db-instance-class db.r4.large .`	The terminology is inconsistent between SQL Server and Aurora PostgreSQL. A database snapshot in SQL Server is similar to database cloning in Aurora PostgreSQL. Aurora PostgreSQL database snapshots are similar to a `COPY_ONLY` backup in SQL Server.

Feature	SQL Server	Aurora PostgreSQL	Comments
Point in time restore	`RESTORE DATABASE | LOG … WITH STOPAT…`	Create new cluster from a cluster snapshot by given custom time to restore: `aws rds restore-db-clusterto-point-in-time --db-clusteridentifier clusternamerestore --source-db-clusteridentifier clustername --restore-to-time 2017-09-19T23:45:00.000Z .` Add a new instance to the new or restored cluster: `aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --dbcluster-identifier clustername-restore --db-`	

Feature	SQL Server	Aurora PostgreSQL	Comments
		`instanceidentifier newinstancenodeA --db-instance-class db.r4.large .`	
Partial restore	`RESTORE DATABASE… FILE= … | FILEGROUP = …`	N/A	You can restore the cluster to a new cluster and copy the needed data to the primary cluster.

For more information, see [Managing an Amazon Aurora DB cluster]() in the *User Guide for Aurora*.

High Availability Essentials

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Multi replica, scale out solution using Amazon Aurora clusters and Availability Zones.

SQL Server Usage

SQL Server provides several solutions to support high availability and disaster recovery requirements including Always On Failover Cluster Instances (FCI), Always On Availability Groups, Database Mirroring, and Log Shipping. The following sections describe each solution.

SQL Server 2017 also adds new Availability Groups functionality which includes read-scale support without a cluster, Minimum Replica Commit Availability Groups setting, and Windows-Linux cross-OS migrations and testing.

SQL Server 2019 introduces support for creating Database Snapshots of databases that include memory-optimized filegroups. A database snapshot is a read-only, static view of a SQL Server database. The database snapshot is transactional consistent with the source database as of the moment of the snapshot's creation. Among other things, some benefits of the database snapshots with regard to high availability are:

- You can use snapshots for reporting purposes.
- Maintaining historical data for report generation.
- Using a mirror database that you are maintaining for availability purposes to offload reporting.

For more information about snapshots, see Database Snapshots in the *SQL Server documentation*.

SQL Server 2019 introduces secondary to primary connection redirection for Always On Availability Groups. It allows client application connections to be directed to the primary replica regardless of the target server specified in the connections string. The connection string can target a secondary replica. Using the right configuration of the availability group replica and the settings in the connection string, you can automatically redirect the connection to the primary replica.

For more information, see Secondary to primary replica read/write connection redirection (Always On Availability Groups) in the *SQL Server documentation*.

Always On Failover Cluster Instances

Always On Failover Cluster Instances (FCI) use the Windows Server Failover Clustering (WSFC) operating system framework to deliver redundancy at the server instance level.

An FCI is an instance of SQL Server installed across two or more WSFC nodes. For client applications, the FCI is transparent and appears to be a normal instance of SQL Server running on a single server. The FCI provides failover protection by moving the services from one WSFC node Windows server to another WSFC node windows server in the event the current active node becomes unavailable or degraded.

FCIs target scenarios where a server fails due to a hardware malfunction or a software hang up. Without FCI, a significant hardware or software failure would render the service unavailable until the malfunction is corrected. With FCI, you can configure another server as a standby to replace the original server if it stops servicing requests.

For each service or cluster resource, there is only one node that actively services client requests (known as owning a resource group). A monitoring agent constantly monitors the resource owners and can transfer ownership to another node in the event of a failure or planned maintenance such as installing service packs or security patches. This process is completely transparent to the client application, which may continue to submit requests as normal when the failover or ownership transfer process completes.

FCI can significantly minimize downtime due to hardware or software general failures. The main benefits of FCI are:

- Full instance level protection.
- Automatic failover of resources from one node to another.
- Supports a wide range of storage solutions. WSFC cluster disks can be iSCSI, Fiber Channel, SMB file shares, and others.
- Supports multi-subnet.
- No need client application configuration after a failover.
- Configurable failover policies.
- Automatic health detection and monitoring.

For more information, see Always On Failover Cluster Instances (SQL Server) in the *SQL Server documentation*.

Always On Availability Groups

Always On Availability Groups is the most recent high availability and disaster recovery solution for SQL Server. It was introduced in SQL Server 2012 and supports high availability for one or more user databases. Because you can configure and manage it at the database level rather than the entire server, it provides much more control and functionality. As with FCI, Always On Availability Groups relies on the framework services of Windows Server Failover Cluster (WSFC) nodes.

Always On Availability Groups utilize real-time log record delivery and apply mechanism to maintain near-real-time readable copies of one or more databases. These copies can also be used as redundant copies for resource usage distribution between servers (a scale-out read solution).

The main characteristics of Always On Availability Groups are:

- Supports up to nine availability replicas: One primary replica and up to eight secondary readable replicas.
- Supports both asynchronous-commit and synchronous-commit availability modes.

- Supports automatic failover, manual failover, and a forced failover. Only the latter can result in data loss.
- Secondary replicas allow both read-only access and offloading of backups.
- Availability Group Listener may be configured for each availability group. It acts as a virtual server address where applications can submit queries. The listener may route requests to a read-only replica or to the primary replica for read-write operations. This configuration also facilitates fast failover as client applications don't need to be reconfigured post failover.
- Flexible failover policies.
- The automatic page repair feature protects against page corruption.
- Log transport framework uses encrypted and compressed channels.
- Rich tooling and APIs including Transact-SQL DDL statements, management studio wizards, Always On Dashboard Monitor, and PowerShell scripting.

For more information, see Always On availability groups: a high-availability and disaster-recovery solution in the *SQL Server documentation*.

Database Mirroring

Microsoft recommends avoiding Database Mirroring for new development. This feature is deprecated and will be removed in a future release. It is recommended to use Always On Availability Groups instead.

Database mirroring is a legacy solution to increase database availability by supporting near instantaneous failover. It is similar in concept to Always On Availability Groups, but can only be configured for one database at a time and with only one standby replica.

For more information, see Database Mirroring (SQL Server) in the *SQL Server documentation*.

Log Shipping

Log shipping is one of the oldest and well tested high availability solutions. It is configured at the database level similar to Always On Availability Groups and Database Mirroring. You can use log shipping to maintain one or more secondary databases for a single primary database.

The log shipping process involves three steps:

1. Backing up the transaction log of the primary database instance.

2. Copying the transaction log backup file to a secondary server.
3. Restoring the transaction log backup to apply changes to the secondary database.

You can configure log shipping to create multiple secondary database replicas by repeating steps 2 and 3 for each secondary server. Unlike FCI and Always On Availability Groups, log shipping solutions don't provide automatic failover.

In the event the primary database becomes unavailable or unusable for any reason, an administrator must configure the secondary database to serve as the primary and potentially reconfigure all client applications to connect to the new database.

Note

You can use secondary databases used for read-only access, but require special handling. For more information, see Configure Log Shipping (SQL Server) in the *SQL Server documentation*.

The main characteristics of log shipping solutions are:

- Provides redundancy for a single primary database and one or more secondary databases. Log shipping is considered less of a high availability solution due to the lack of automatic failover.
- Supports limited read-only access to secondary databases.
- Administrators have control over the timing and delays of the primary server log backup and secondary server restoration.
- Longer delays can be useful if data is accidentally modified or deleted in the primary database.

For more information, see About Log Shipping (SQL Server) in the *SQL Server documentation*.

Examples

Configure an Always On Availability Group.

```
CREATE DATABASE DB1;

ALTER DATABASE DB1 SET RECOVERY FULL;

BACKUP DATABASE DB1 TO DISK = N'\\MyBackupShare\DB1\DB1.bak' WITH FORMAT;
```

```
CREATE ENDPOINT DBHA STATE=STARTED
AS TCP (LISTENER_PORT=7022) FOR DATABASE_MIRRORING (ROLE=ALL);

CREATE AVAILABILITY GROUP AG_DB1
  FOR
    DATABASE DB1
  REPLICA ON
    'SecondarySQL' WITH
      (
        ENDPOINT_URL = 'TCP://SecondarySQL.MyDomain.com:7022',
        AVAILABILITY_MODE = ASYNCHRONOUS_COMMIT,
        FAILOVER_MODE = MANUAL
      );

-- On SecondarySQL
ALTER AVAILABILITY GROUP AG_DB1 JOIN;

RESTORE DATABASE DB1 FROM DISK = N'\\MyBackupShare\DB1\DB1.bak'
WITH NORECOVERY;

-- On Primary
BACKUP LOG DB1
TO DISK = N'\\MyBackupShare\DB1\DB1_Tran.bak'
  WITH NOFORMAT

-- On SecondarySQL
RESTORE LOG DB1
  FROM DISK = N'\\MyBackupShare\DB1\DB1_Tran.bak'
  WITH NORECOVERY

ALTER DATABASE MyDb1 SET HADR AVAILABILITY GROUP = MyAG;
```

For more information, see Business continuity and database recovery - SQL Server in the *SQL Server documentation*.

PostgreSQL Usage

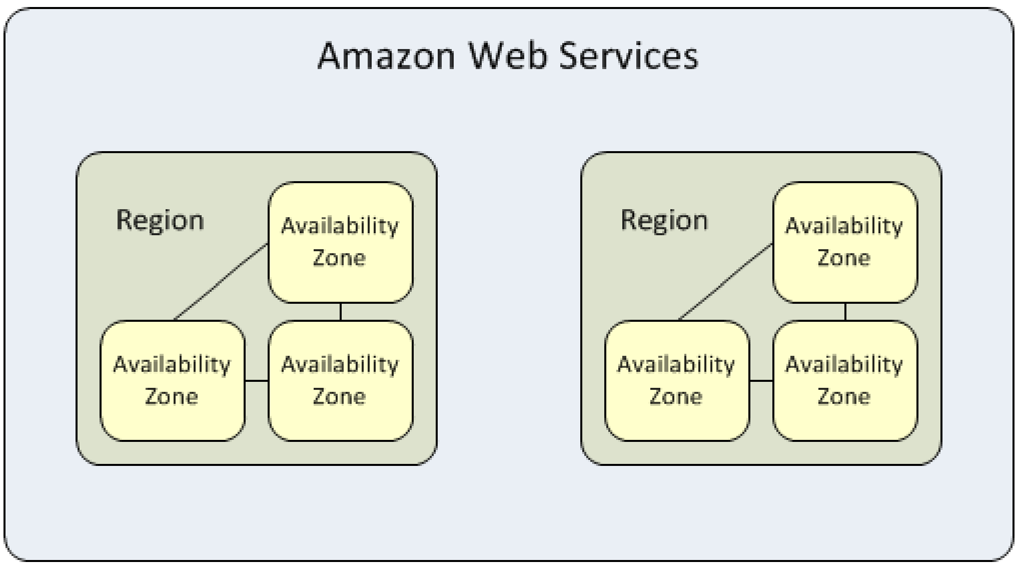
Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) is a fully managed Platform as a Service (PaaS) providing high availability capabilities. Amazon RDS provides database and instance administration functionality for provisioning, patching, backup, recovery, failure detection, and repair.

New Aurora PostgreSQL database instances are always created as part of a cluster. If you don't specify replicas at creation time, a single-node cluster is created. You can add database instances to clusters later.

Regions and Availability Zones

Amazon Relational Database Service (Amazon RDS) is hosted in multiple global locations. Each location is composed of Regions and Availability Zones. Each Region is a separate geographic area having multiple, isolated Availability Zones. Amazon RDS supports placement of resources such as database instances and data storage in multiple locations. By default, resources aren't replicated across regions.

Each Region is completely independent and each Availability Zone is isolated from all others. However, the main benefit of Availability Zones within a Region is that they are connected through low-latency, high bandwidth local network links.



Resources may have different scopes. A resource may be global, associated with a specific region (region level), or associated with a specific Availability Zone within a region. For more information, see [Resource locations](#) in the *User Guide for Linux Instances*.

When you create a database instance, you can specify an availability zone or use the default **No preference** option. In this case, Amazon chooses the availability zone for you.

You can distribute Aurora PostgreSQL instances across multiple availability zones. You can design applications designed to take advantage of failover such that in the event of an instance in one availability zone failing, another instance in different availability zone will take over and handle requests.

You can use elastic IP addresses to abstract the failure of an instance by remapping the virtual IP address to one of the available database instances in another Availability Zone. For more information, see Elastic IP addresses in the *User Guide for Linux Instances*.

An Availability Zone is represented by a region code followed by a letter identifier. For example, `us-east-1a`.

Note

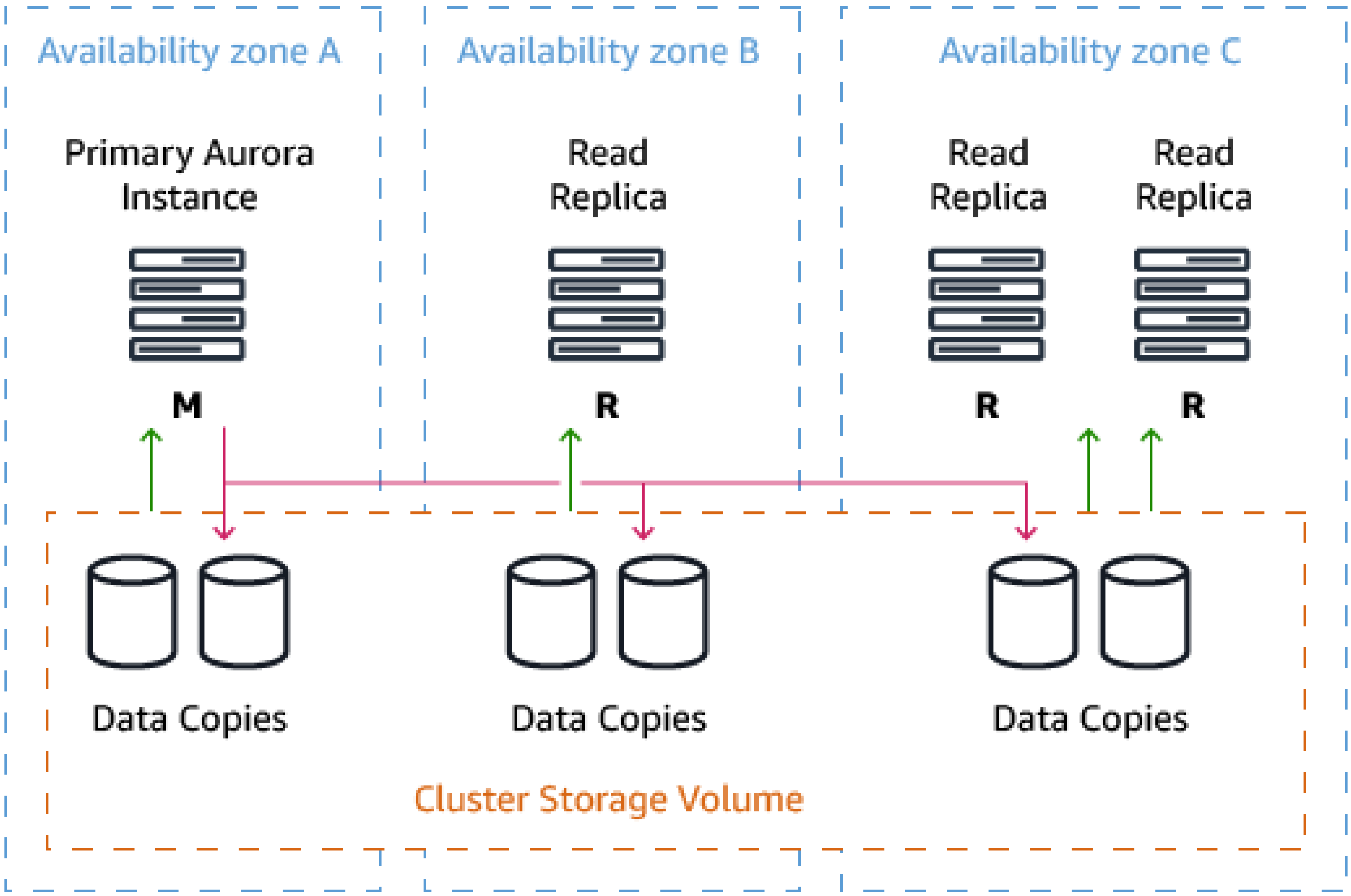
To guarantee even resource distribution across Availability Zones for a region, Amazon RDS independently maps Availability Zones to identifiers for each account. For example, the Availability Zone us-east-1a for one account might not be in the same location as us-east-1a for another account. Users can't coordinate Availability Zones between accounts.

Aurora PostgreSQL DB Cluster

A DB cluster consists of one or more DB instances and a cluster volume that manages the data for those instances. A cluster volume is a virtual database storage volume that may span multiple Availability Zones with each holding a copy of the database cluster data.

An Amazon Aurora database cluster is made up of one of more of the following types of instances:

- A Primary instance that supports both read and write workloads. This instance is used for all DML transactions. Every Amazon Aurora DB cluster has one, and only, one primary instance.
- An Amazon Aurora Replica that supports read-only workloads. Every Aurora PostgreSQL database cluster may contain from zero to 15 Amazon Aurora Replicas in addition to the primary instance for a total maximum of 16 instances. Amazon Aurora Replicas enable scale-out of read operations by offloading reporting or other read-only processes to multiple replicas. Place Amazon Aurora replicas in multiple availability Zones to increase availability of the databases.



Endpoints

Endpoints are used to connect to Aurora PostgreSQL databases. An endpoint is a Universal Resource Locator (URL) comprised of a host address and port number.

- A Cluster Endpoint is an endpoint for an Amazon Aurora database cluster that connects to the current primary instance for that database cluster regardless of the availability zone in which the primary resides. Every Aurora PostgreSQL DB cluster has one cluster endpoint and one primary instance. The cluster endpoint should be used for transparent failover for either read or write workloads.

> **Note**
>
> Use the cluster endpoint for all write operations including all DML and DDL statements.

If the primary instance of a DB cluster fails for any reason, Amazon Aurora automatically fails over server requests to a new primary instance. An example of a typical Aurora PostgreSQL DB Cluster endpoint is: `mydbcluster.cluster-123456789012.us-east-1.rds.amazonaws.com:3306`.

- A Reader Endpoint is an endpoint that is used to connect to one of the Aurora read-only replicas in the database cluster. Each Aurora PostgreSQL database cluster has one reader endpoint. If there are more than one Aurora Replicas in the cluster, the reader endpoint redirects the connection to one of the available replicas. Use the Reader Endpoint to support load balancing for read-only connections. If the DB cluster contains no replicas, the reader endpoint redirects the connection to the primary instance. If an Aurora Replica is created later, the Reader Endpoint starts directing connections to the new Aurora Replica with minimal interruption in service. An example of a typical Aurora PostgreSQL DB Reader Endpoint is: `mydbcluster.cluster-ro-123456789012.us-east-1.rds.amazonaws.com:3306`.
- An Instance Endpoint is a specific endpoint for every database instance in an Aurora DB cluster. Every Aurora PostgreSQL DB instance regardless of its role has its own unique instance endpoint. Use the Instance Endpoints only when the application handles failover and read workload scale-out on its own. For example, you can have certain clients connect to one replica and others to another. An example of a typical Aurora PostgreSQL DB Reader Endpoint is: `pgsdbinstance.123456789012.us-east-1.rds.amazonaws.com:3306`.

Some general considerations for using endpoints:

- Consider using the cluster endpoint instead of individual instance endpoints because it supports high-availability scenarios. In the event that the primary instance fails, Aurora PostgreSQL automatically fails over to a new primary instance. You can accomplish this configuration by either promoting an existing Aurora Replica to be the new primary or by creating a new primary instance.
- If you use the cluster endpoint instead of the instance endpoint, the connection is automatically redirected to the new primary.
- If you choose to use the instance endpoint, you must use the Amazon RDS console or the API to discover which database instances in the database cluster are available and their current roles. Then, connect using that instance endpoint.

- Be aware that the reader endpoint load balances connections to Aurora Replicas in an Aurora database cluster, but it doesn't load balance specific queries or workloads. If your application requires custom rules for distributing read workloads, use instance endpoints.
- The reader endpoint may redirect connection to a primary instance during the promotion of an Aurora Replica to a new primary instance.

Amazon Aurora Storage

Aurora PostgreSQL data is stored in a cluster volume. The cluster volume is a single, virtual volume that uses fast solid-state disk (SSD) drives. The cluster volume is comprised of multiple copies of the data distributed between availability zones in a region. This configuration minimizes the chances of data loss and allows for the failover scenarios mentioned in the preceding sections.

Amazon Aurora cluster volumes automatically grow to accommodate the growth in size of your databases. An Aurora cluster volume has a maximum size of 64 terabytes (TiB). Since table size is theoretically limited to the size of the cluster volume, the maximum table size in an Aurora DB cluster is 64 TiB.

Storage Auto-Repair

The chance of data loss due to disk failure is greatly minimize due to the fact that Aurora PostgreSQL maintains multiple copies of the data in three Availability Zones. Aurora PostgreSQL detects failures in the disks that make up the cluster volume. If a disk segment fails, Aurora repairs the segment automatically. Repairs to the disk segments are made using data from the other cluster volumes to ensure correctness. This process allows Aurora to significantly minimize the potential for data loss and the subsequent need to restore a database.

Survivable Cache Warming

When a database instance starts, Aurora PostgreSQL performs a warming process for the buffer pool. Aurora PostgreSQL pre-loads the buffer pool with pages that have been frequently used in the past. This approach improves performance and shortens the natural cache filling process for the initial period when the database instance starts servicing requests. Aurora PostgreSQL maintains a separate process to manage the cache, which can stay alive even when the database process restarts. The buffer pool entries remain in memory regardless of the database restart providing the database instance with a fully warm buffer pool.

Crash Recovery

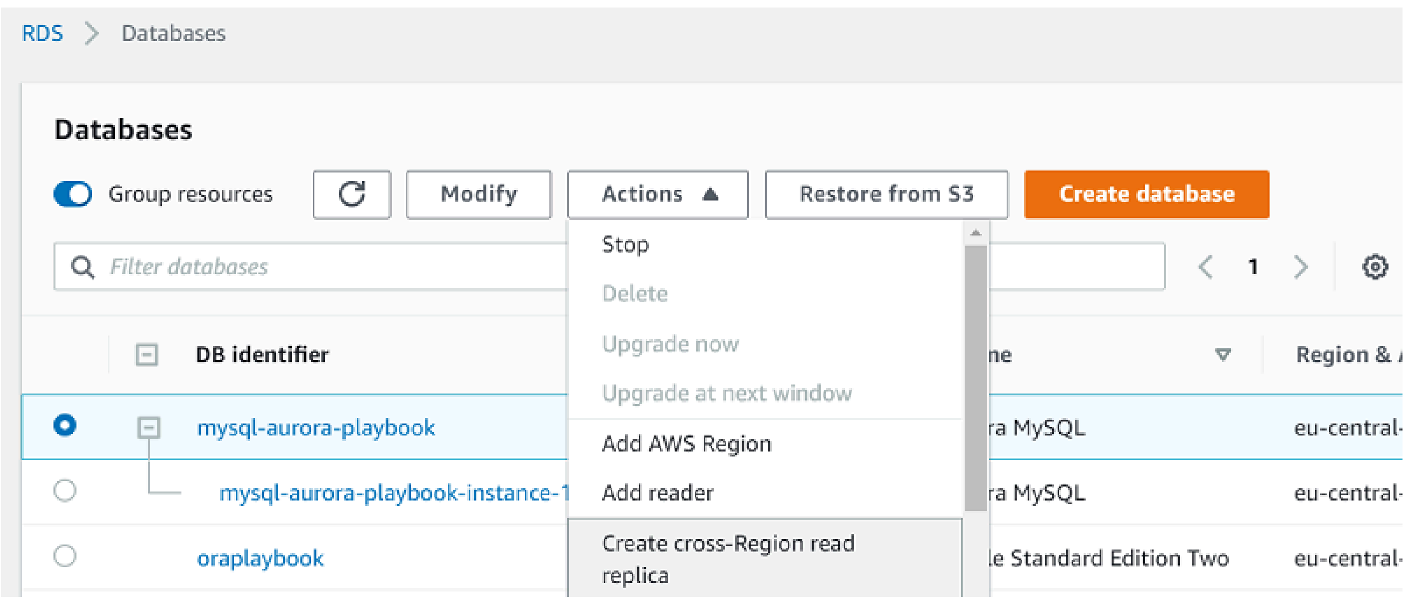
Aurora PostgreSQL can instantaneously recover from a crash and continue to serve requests. Crash recovery is performed asynchronously using parallel threads enabling the database to remain open and available immediately after a crash.

For more information, see [Fault tolerance for an Aurora DB cluster]() in the *User Guide for Aurora*.

Examples

The following walkthrough demonstrates how to create a read-replica:

1. Log in to the AWS Console, and choose **RDS**.
2. Select the instance and choose **Instance actions**, **Create cross-region read replica**.



3. On the next page, enter all required details and choose **Create**.

After the replica is created, you can run read and write operations on the primary instance and read-only operations on the replica.

Summary

Feature	SQL Server	Aurora PostgreSQL	Comments
Server level failure protection	Failover Cluster Instances	N/A	Not applicable. Clustering is handled by Aurora PostgreSQL.
Database level failure protection	Always On Availability Groups	Amazon Aurora Replicas	
Log replication	Log Shipping	N/A	Not applicable. Aurora PostgreSQL handles data replication at the storage level.
Disk error protection	`RESTORE… PAGE=`	Automatically	
Maximum read-only replicas	8 + Primary	15 + Primary	
Failover address	Availability group listener	Cluster endpoint	
Read-only workloads	`READ INTENT` connection	Read Endpoint	

For more information, see [Amazon Aurora DB clusters](#) in the *User Guide for Aurora* and [Regions and Zones](#) in the *User Guide for Linux Instances*.

Indexes

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Indexes	PostgreSQL doesn't support CLUSTERED INDEX. Few missing options.

SQL Server Usage

Indexes are physical disk structures used to optimize data access. They are associated with tables or materialized views and allow the query optimizer to access rows and individual column values without scanning an entire table.

An index consists of index keys, which are columns from a table or view. They are sorted in ascending or descending order providing quick access to individual values for queries that use equality or range predicates. Database indexes are similar to book indexes that list page numbers for common terms. Indexes created on multiple columns are called composite indexes.

SQL Server implements indexes using the balanced tree algorithm (B-tree).

> **Note**
>
> SQL Server supports additional index types such as hash indexes (for memory-optimized tables), spatial indexes, full text indexes, and XML indexes.

Indexes are created automatically to support table primary keys and unique constraints. They are required to efficiently enforce uniqueness. You can create up to 250 indexes on a table to support common queries.

SQL Server provides two types of B-tree indexes: clustered indexes and non-clustered indexes.

Clustered Indexes

Clustered indexes include all the table's column data in their leaf level. The entire table data is sorted and logically stored in order on disk. A clustered index is similar to a phone directory index where the entire data is contained for every index entry. Clustered indexes are created by default for primary key constraints. However, a primary key doesn't necessarily need to use a clustered index if it is explicitly specified as non-clustered.

Clustered indexes are created using the `CREATE CLUSTERED INDEX` statement. You can create only one clustered index for each table because the index itself is the table's data. A table having a clustered index is called a clustered table (also known as an index-organized table in other relational database management systems). A table with no clustered index is called a heap.

Examples

Create a Clustered Index as part of table definition.

```
CREATE TABLE MyTable
(
  Col1 INT NOT NULL
  PRIMARY KEY,
  Col2 VARCHAR(20) NOT NULL
);
```

Create an explicit clustered index using `CREATE INDEX`.

```
CREATE TABLE MyTable
(
  Col1 INT NOT NULL
  PRIMARY KEY NONCLUSTERED,
  Col2 VARCHAR(20) NOT NULL
);
```

```
CREATE CLUSTERED INDEX IDX1
ON MyTable(Col2);
```

Non-Clustered Indexes

Non-clustered indexes also use the B-tree algorithm but consist of a data structure separate from the table itself. They are also sorted by the index keys, but the leaf level of a non-clustered index contains pointers to the table rows; not the entire row as with a clustered index.

You can create up to 999 non-clustered indexes on a SQL Server table. The type of pointer used at the lead level of a non-clustered index (also known as a row locator) depends on whether the table has a clustered index (clustered table) or not (heap). For heaps, the row locators use a physical pointer (RID). For clustered tables, row locators use the clustering key plus a potential uniquifier. This approach minimizes non-clustered index updates when rows move around, or the clustered index key value changes.

Both clustered and non-clustered indexes may be defined as `UNIQUE` using the `CREATE UNIQUE INDEX` statement. SQL Server maintains indexes automatically for a table or view and updates the relevant keys when table data is modified.

Examples

Create a unique non-clustered index as part of table definition.

```
CREATE TABLE MyTable
(
  Col1 INT NOT NULL
  PRIMARY KEY,
  Col2 VARCHAR(20) NOT NULL
  UNIQUE
);
```

Create a unique non-clustered index using CREATE INDEX.

```
CREATE TABLE MyTable
(
  Col1 INT NOT NULL
  PRIMARY KEY CLUSTERED,
  Col2 VARCHAR(20) NOT NULL
);
```

```
CREATE UNIQUE NONCLUSTERED INDEX IDX1 ON MyTable(Col2);
```

Filtered Indexes and Covering Indexes

SQL Server also supports two special options for non-clustered indexes. You can create filtered indexes to index only a subset of a table's data. They are useful when it is known that the application will not need to search for specific values such as NULLs.

For queries that typically require searching on particular columns but also need additional column data from the table, you can configure non-clustered indexes. They include additional column data in the index leaf level in addition to the row locator. This may prevent expensive lookup operations, which follow the pointers to either the physical row location (in a heap) or traverse the clustered index key to fetch the rest of the data not part of the index. If a query can get all the data it needs from the non-clustered index leaf level, that index is considered a covering index.

Examples

Create a filtered index to exclude NULL values.

```
CREATE NONCLUSTERED INDEX IDX1
ON MyTable(Col2)
WHERE Col2 IS NOT NULL;
```

Create a covering index for queries that search on col2 but also need data from col3.

```
CREATE NONCLUSTERED INDEX IDX1
ON MyTable (Col2)
INCLUDE (Col3);
```

Indexes On Computed Columns

In SQL Server, you can create indexes on persisted computed columns. Computed columns are table or view columns that derive their value from an expression based on other columns in the table. They aren't explicitly specified when data is inserted or updated. This feature is useful when a query's filter predicates aren't based on the column table data as-is, but on a function or expression.

Examples

For example, consider the following table that stores phone numbers for customers, but the format isn't consistent for all rows; some include country code and some don't:

```
CREATE TABLE PhoneNumbers
(
  PhoneNumber VARCHAR(15) NOT NULL
  PRIMARY KEY,
  Customer VARCHAR(20) NOT NULL
);
```

```
INSERT INTO PhoneNumbers
VALUES
('+1-510-444-3422','Dan'),
('644-2442-3119','John'),
('1-402-343-1991','Jane');
```

The following query to look up the owner of a specific phone number must scan the entire table because the index can’t be used due to the preceding % wild card.

```
SELECT Customer
FROM PhoneNumbers
WHERE PhoneNumber LIKE '%510-444-3422';
```

A potential solution would be to add a computed column that holds the phone number in reverse order.

```
ALTER TABLE PhoneNumbers
ADD ReversePhone AS REVERSE(PhoneNumber)
PERSISTED;
```

```
CREATE NONCLUSTERED INDEX IDX1
ON PhoneNumbers (ReversePhone)
INCLUDE (Customer);
```

Now, you can use the following query to search for the customer based on the reverse string, which places the wild card at the end of the `LIKE` predicate. This approach provides an efficient index seek to retrieve the customer based on the phone number value.

```
DECLARE @ReversePhone VARCHAR(15) = REVERSE('510-444-3422');
SELECT Customer
FROM PhoneNumbers
WHERE ReversePhone LIKE @ReversePhone + '%';
```

For more information, see Clustered and nonclustered indexes described and CREATE INDEX (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) supports balanced tree (B-tree) indexes similar to SQL Server. However, the terminology, use, and options for these indexes are different.

Aurora PostgreSQL is missing the `CLUSTERED INDEX` feature but has other options which SQL Server doesn't have, index prefix, and binary large object (BLOB) indexing.

Starting with PostgreSQL 10, there are many improvements in performance, related to joins and parallel scans of the indexes.

Starting with PostgreSQL 12, you can monitor progress of `CREATE INDEX` and `REINDEX` operations by querying the `pg_stat_progress_create_index` system view.

Cluster Table

PostgreSQL doesn't support cluster tables directly, but provides similar functionality using the `CLUSTER` feature. The PostgreSQL `CLUSTER` statement specifies table sorting based on an index already associated with the table. When using the PostgreSQL `CLUSTER` command, the data in the table is physically sorted based on the index, possibly using a primary key column.

You can use the `CLUSTER` statement to re-cluster the table.

Examples

```
CREATE TABLE SYSTEM_EVENTS (
  EVENT_ID NUMERIC,
  EVENT_CODE VARCHAR(10) NOT NULL,
  EVENT_DESCIPTION VARCHAR(200),
  EVENT_TIME DATE NOT NULL,
  CONSTRAINT PK_EVENT_ID PRIMARY KEY(EVENT_ID));

INSERT INTO SYSTEM_EVENTS VALUES(9, 'EV-A1-10', 'Critical', '01-JAN-2017');
INSERT INTO SYSTEM_EVENTS VALUES(1, 'EV-C1-09', 'Warning', '01-JAN-2017');
INSERT INTO SYSTEM_EVENTS VALUES(7, 'EV-E1-14', 'Critical', '01-JAN-2017');
```

```
CLUSTER SYSTEM_EVENTS USING PK_EVENT_ID;
SELECT * FROM SYSTEM_EVENTS;

event_id  event_code  event_desciption  event_time
1         EVNT-C1-09  Warning           2017-01-01
7         EVNT-E1-14  Critical          2017-01-01
9         EVNT-A1-10  Critical          2017-01-01

INSERT INTO SYSTEM_EVENTS VALUES(2, 'EV-E2-02', 'Warning', '01-JAN-2017');
SELECT * FROM SYSTEM_EVENTS;

event_id  event_code  event_desciption  event_time
1         EVNT-C1-09  Warning           2017-01-01
7         EVNT-E1-14  Critical          2017-01-01
9         EVNT-A1-10  Critical          2017-01-01
2         EVNT-E2-02  Warning           2017-01-01

CLUSTER SYSTEM_EVENTS USING PK_EVENT_ID; -- Run CLUSTER again to re-cluster
SELECT * FROM SYSTEM_EVENTS;

event_id  event_code  event_desciption  event_time
1         EVNT-C1-09  Warning           2017-01-01
2         EVNT-E2-02  Warning           2017-01-01
7         EVNT-E1-14  Critical          2017-01-01
9         EVNT-A1-10  Critical          2017-01-01
```

B-tree Indexes

When you create an index in PostgreSQL, a B-tree index is created by default, similar to the behavior in SQL Server. PostgreSQL B-tree indexes have the same characteristics as SQL Server and can handle equality and range queries on data. The PostgreSQL optimizer considers using B-tree indexes especially for one or more of the following operators in queries: >, >=, <, #, =.

In addition, you can achieve performance improvements when using `IN`, `BETWEEN`, `IS NULL`, or `IS NOT NULL`.

Starting with PostgreSQL 10, there is a support of parallel B-tree index scans. This change allows this index type pages to be searched by separate parallel workers.

Example

Create a PostgreSQL B-Tree Index.

```
CREATE INDEX IDX_EVENT_ID ON SYSTEM_LOG(EVENT_ID);
OR
CREATE INDEX IDX_EVENT_ID1 ON SYSTEM_LOG USING BTREE (EVENT_ID);
```

For more information, see CREATE INDEX in the *PostgreSQL documentation*.

Column and Multiple Column Secondary Indexes

Currently, only B-tree, GiST, GIN, and BRIN support multicolumn indexes. You can specify 32 columns when you create a multicolumn index.

PostgreSQL uses the same syntax as SQL Server to create multicolumn indexes.

Examples

Create a multicolumn index on the EMPLOYEES table.

```
CREATE INDEX IDX_EMP_COMPI
ON EMPLOYEES (FIRST_NAME, EMAIL, PHONE_NUMBER);
```

Drop a multicolumn index.

```
DROP INDEX IDX_EMP_COMPI;
```

For more information, see Multicolumn Indexes in the *PostgreSQL documentation*.

Expression Indexes and Partial Indexes

Create an Expression Index in PostgreSQL.

```
CREATE TABLE SYSTEM_EVENTS(
  EVENT_ID NUMERIC PRIMARY KEY,
  EVENT_CODE VARCHAR(10) NOT NULL,
  EVENT_DESCIPTION VARCHAR(200),
  EVENT_TIME TIMESTAMP NOT NULL);

CREATE INDEX EVNT_BY_DAY ON SYSTEM_EVENTS(EXTRACT(DAY FROM EVENT_TIME));
```

Insert records into the SYSTEM_EVENTS table, gathering table statistics using the ANALYZE statement and verifying that the EVNT_BY_DAY expression index is being used for data access.

```
INSERT INTO SYSTEM_EVENTS
  SELECT ID AS event_id,
    'EVNT-A'||ID+9||'-'||ID AS event_code,
    CASE WHEN mod(ID,2) = 0 THEN 'Warning' ELSE 'Critical' END AS event_desc,
    now() + INTERVAL '1 minute' * ID AS event_time
    FROM
    (SELECT generate_series(1,1000000) AS ID) A;
INSERT 0 1000000

ANALYZE SYSTEM_EVENTS;
ANALYZE

EXPLAIN
  SELECT * FROM SYSTEM_EVENTS
  WHERE EXTRACT(DAY FROM EVENT_TIME) = '22';

QUERY PLAN

Bitmap Heap Scan on system_events (cost=729.08..10569.58 rows=33633 width=41)
Recheck Cond: (date_part('day'::text, event_time) = '22'::double precision)
-> Bitmap Index Scan on evnt_by_day (cost=0.00..720.67 rows=33633 width=0)
Index Cond: (date_part('day'::text, event_time) = '22'::double precision)
```

Partial Indexes

PostgreSQL also provides partial indexes, which are indexes that use a WHERE clause when created. The most significant benefit of using partial indexes is a reduction of the overall subset of indexed data, allowing users to index relevant table data only. You can use partial indexes to increase efficiency and reduce the size of the index.

Example

The following example creates a PostgreSQL partial Index.

```
CREATE TABLE SYSTEM_EVENTS(
  EVENT_ID NUMERIC PRIMARY KEY,
  EVENT_CODE VARCHAR(10) NOT NULL,
  EVENT_DESCIPTION VARCHAR(200),
  EVENT_TIME DATE NOT NULL);

CREATE INDEX IDX_TIME_CODE ON SYSTEM_EVENTS(EVENT_TIME)
```

```
WHERE EVENT_CODE like '01-A%';
```

For more information, see Building Indexes Concurrently in the *PostgreSQL documentation*.

BRIN Indexes

PostgreSQL doesn't provide native support for BITMAP indexes. However, you can use a BRIN index, which splits table records into block ranges with `MIN/MAX` summaries. A BRIN index is a partial alternative for certain analytic workloads. For example, BRIN indexes are suited for queries that rely heavily on aggregations to analyze large numbers of records.

Example

The following example creates a PostgreSQL BRIN index.

```
CREATE INDEX IDX_BRIN_EMP ON EMPLOYEES USING BRIN(salary);
```

Summary

The following table summarizes the key differences to consider when migrating b-tree indexes from SQL Server to Aurora PostgreSQL.

Index feature	SQL Server	Aurora PostgreSQL
Clustered indexes supported for	Table keys, composite or single column, unique and non-unique, null or not null.	On indexes.
Non-clustered indexes supported for	Table keys, composite or single column, unique and non-unique, null or not null.	Table keys, composite or single column, unique and non-unique, null or not null.
Max number of non-clustered indexes	999	N/A
Max total index key size	900 bytes	N/A
Max columns for each index	32	32

Index feature	SQL Server	Aurora PostgreSQL
Index prefix	N/A	Supported
Filtered indexes	Supported	Supported (partial indexes)
Indexes on BLOBs	N/A	Supported

For more information, see Index Types, CREATE INDEX, CLUSTER, and Building Indexes Concurrently in the *PostgreSQL documentation*.

Management

Topics

- SQL Server Agent and PostgreSQL Scheduled Lambda
- Alerting
- Database Mail
- ETL
- SQL Server Export and Import with Text Files and PostgreSQL pg_dump and pg_restore
- Viewing Server Logs
- Maintenance Plans
- Monitoring
- SQL Server Resource Governor and PostgreSQL Dedicated Amazon Aurora Clusters or Aurora Read-Replicas
- SQL Server Linked Servers and PostgreSQL DBLink and FDWrapper
- Scripting

SQL Server Agent and PostgreSQL Scheduled Lambda

SQL Server Usage

SQL Server Agent provides two main functions: scheduling automated maintenance jobs and alerting.

> **Note**
>
> Other SQL Server built-in frameworks such as replication, also use SQL Server Agent jobs.

For more information, see Maintenance Plans and Alerting.

PostgreSQL Usage

Currently, there is no equivalent in Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) for scheduling tasks but you can create scheduled AWS Lambda that will run a stored procedure. Find an example in Database Mail.

Alerting

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Use Amazon RDS event notification with Amazon Simple Notification Service.

SQL Server Usage

SQL Server provides SQL Server Agent to generate alerts. When running, SQL Server Agent constantly monitors SQL Server windows application log messages, performance counters, and Windows Management Instrumentation (WMI) objects. When a new error event is detected, the agent checks the `msdb` database for configured alerts and runs the specified action.

You can define SQL Server Agent alerts for the following categories:

- SQL Server events.
- SQL Server performance conditions.
- WMI events.

For SQL Server events, the alert options include the following settings:

- **Error Number** — Alert when a specific error is logged.
- **Severity Level** — Alert when any error in the specified severity level is logged.
- **Database** — Filter the database list for which the event will generate an alert.

- **Event Text** — Filter specific text in the event message.

> **Note**
>
> SQL Server Agent is pre-configured with several high severity alerts. It is highly recommended to turn on these alerts.

To generate an alert in response to a specific performance condition, specify the performance counter to be monitored, the threshold values for the alert, and the predicate for the alert to occur. The following list identifies the performance alert settings:

- **Object** — The Performance counter category or the monitoring area of performance.
- **Counter** — A counter is a specific attribute value of the object.
- **Instance** — Filter by SQL Server instance (multiple instances can share logs).
- **Alert if counter and Value** — The threshold for the alert and the predicate. The threshold is a number. Predicates are **falls below**, **becomes equal to**, or **rises above** the threshold.

WMI events require the WMI namespace and the WMI Query Language (WQL) query for specific events.

You can assign alerts to specific operators with schedule limitations and multiple response types including:

- Run an SQL Server Agent job.
- Send Email, Net Send command, or a pager notification.

You can configure alerts and responses with SQL Server Management Studio or system stored procedures.

Example

Configure an alert for all errors with severity 20.

```
EXEC msdb.dbo.sp_add_alert
@name = N'Severity 20 Error Alert',
@severity = 20,
```

```
@notification_message = N'A severity 20 Error has occurred. Initiating emergency
 procedure',
@job_name = N'Error 20 emergency response';
```

For more information, see Alerts in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) doesn't support direct configuration of engine alerts. Use the Event Notifications Infrastructure to collect history logs or receive event notifications in near real-time.

Amazon Relational Database Service (Amazon RDS) uses Amazon Simple Notification Service (Amazon SNS) to provide notifications for events. SNS can send notifications in any form supported by the region including email, text messages, or calls to HTTP endpoints for response automation.

Events are grouped into categories. You can only subscribe to event categories, not individual events. SNS sends notifications when any event in a category occurs.

You can subscribe to alerts for database instances, database clusters, database snapshots, database cluster snapshots, database security groups, and database parameter groups. For example, a subscription to the Backup category for a specific database instance sends notifications when backup related events occur on that instance. A subscription to a Configuration Change category for a database security group sends notifications when the security group changes.

Note

For Amazon Aurora, some events occur at the cluster rather than instance level. You will not receive those events if you subscribe to an Amazon Aurora DB instance.

SNS sends event notifications to the address specified when the subscription was created. Typically, administrators create several subscriptions. For example, one subscription to receive logging events and another to receive only critical events for a production environment requiring immediate responses.

You can disable notifications without deleting a subscription by setting the Enabled radio button to No in the Amazon RDS console. Alternatively, use the Command Line Interface (CLI) or Amazon RDS API to change the Enabled setting.

Subscriptions are identified by the Amazon Resource Name (ARN) of an Amazon SNS topic. The Amazon RDS console creates ARNs when subscriptions are created. When using the CLI or API, you must create the ARN using the Amazon SNS console or the Amazon SNS API.

Examples

The following walkthrough demonstrates how to create an event notification subscription.

1. Sign in to your AWS account, and choose **RDS**.
2. Choose **Events** on the left navigation pane. This screen that presents relevant Amazon RDS events occurs.
3. Choose **Event subscriptions** and then choose **Create event subscription**.
4. Enter the **Name of the subscription** and select a **Target of ARN** or **Email**. For email subscriptions, enter values for **Topic** name and **With these recipients**.
5. Select the event source, choose specific event categories to be monitored, and choose **Create**.
6. On the Amazon RDS dashboard, choose **Recent events**.

For more information, see Using Amazon RDS event notification in the *Amazon Relational Database Service User Guide*.

Raising Errors from Within the Database

The following table shows the PostgreSQL log severity levels.

Log type	**Information written to log**
`DEBUG1…DEBUG5`	Provides successively-more-detailed information for use by developers.
`INFO`	Provides information implicitly requested by the user.
`NOTICE`	Provides information that might be helpful to users.
`WARNING`	Provides warnings of likely problems.

Log type	Information written to log
ERROR	Reports the error that caused the current command to abort.
LOG	Reports information of interest to administr ators.
FATAL	Reports the error that caused the current session to abort.
PANIC	Reports the error that caused all database sessions to abort.

Several parameters control how and where PostgreSQL log and errors files are placed:

Parameter	Description
log_filename	Sets the file name pattern for log files. To modify, use an Aurora Database Parameter Group.
log_rotation_age	(min) Automatic log file rotation will occur after N minutes. To modify, use an Aurora Database Parameter Group.
log_rotation_size	(kB) Automatic log file rotation will occur after N kilobytes. To modify, use an Aurora Database Parameter Group.
log_min_messages	Sets the message levels that are logged, such as DEBUG, ERROR, INFO, and so on. To modify, use an Aurora Database Parameter Group.
log_min_error_statement	Causes all statements generating errors at or above this level to be logged, such as DEBUG,

Parameter	Description
	`ERROR`, `INFO`, and so on. To modify, use an Aurora Database Parameter Group.
`log_min_duration_statement`	Sets the minimum run time above which statements will be logged (ms). To modify, use an Aurora Database Parameter Group.

> **Note**
>
> Modifications to certain parameters such as `log_directory` (which sets the destination directory for log files) or `logging_collector` (which starts a subprocess to capture the `stderr` output or `csvlogs` into log files) are disabled for an Aurora PostgreSQL instance.

For more information, see Error Reporting and Logging in the *PostgreSQL documentation*.

Database Mail

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	SQL Server Mail	Use Lambda integration.

SQL Server Usage

The Database Mail framework is an email client solution for sending messages directly from SQL Server. Email capabilities and APIs within the database server provide easy management of the following messages:

- Server administration messages such as alerts, logs, status reports, and process confirmations.
- Application messages such as user registration confirmation and action verifications.

Note

Database Mail is turned off by default.

The main features of the Database Mail framework are:

- Database Mail sends messages using the standard and secure Simple Mail Transfer Protocol (SMTP) .
- The email client engine runs asynchronously and sends messages in a separate process to minimize dependencies.
- Database Mail supports multiple SMTP Servers for redundancy.
- Full support and awareness of Windows Server Failover Cluster for high availability environments.
- Multi-profile support with multiple failover accounts in each profile.
- Enhanced security management with separate roles in the `msdb` database.
- Security is enforced for mail profiles.
- Administrators can monitor and cap attachment sizes.
- You can add attachment file types to a deny list.
- You can log Email activity to SQL Server, the Windows application event log, and a set of system tables in the `msdb` database.
- Supports full auditing capabilities with configurable retention policies.
- Supports both plain text and HTML messages.

Architecture

Database Mail is built on top of the Microsoft SQL Server Service Broker queue management framework.

The system stored procedure `sp_send_dbmail` sends email messages. When you run this stored procedure, it inserts a row to the mail queue and records the Email message.

The queue insert operation triggers the run of the Database Mail process (`DatabaseMail.exe`). The Database Mail process then reads the Email information and sends the message to the SMTP servers.

When the SMTP servers acknowledge or reject the message, the Database Mail process inserts a status row into the status queue, including the result of the send attempt. This insert operation triggers the run of a system stored procedure that updates the status of the Email message send attempt.

Database Mail records all Email attachments in the system tables. SQL Server provides a set of system views and stored procedures for troubleshooting and administration of the Database Mail queue.

Deprecated SQL Mail framework

The previous SQL Mail framework using `xp_sendmail` has been deprecated as of SQL Server 2008R2. For more information, see [Deprecated Database Engine Features in SQL Server 2008 R2] in the *SQL Server documentation*.

The legacy mail system has been completely replaced by the greatly enhanced DB mail framework described here. The previous system has been out of use for many years because it was prone to synchronous run issues and windows mail profile quirks.

Syntax

```
EXECUTE sp_send_dbmail
    [[,@profile_name =] '<Profile Name>']
    [,[,@recipients =] '<Recipients>']
    [,[,@copy_recipients =] '<CC Recipients>']
    [,[,@blind_copy_recipients =] '<BCC Recipients>']
    [,[,@from_address =] '<From Address>']
    [,[,@reply_to =] '<Reply-to Address>']
    [,[,@subject =] '<Subject>']
    [,[,@body =] '<Message Body>']
    [,[,@body_format =] '<Message Body Format>']
    [,[,@importance =] '<Importance>']
    [,[,@sensitivity =] '<Sensitivity>']
    [,[,@file_attachments =] '<Attachments>']
    [,[,@query =] '<SQL Query>']
    [,[,@execute_query_database =] '<Execute Query Database>']
    [,[,@attach_query_result_as_file =] <Attach Query Result as File>]
    [,[,@query_attachment_filename =] <Query Attachment Filename>]
    [,[,@query_result_header =] <Query Result Header>]
    [,[,@query_result_width =] <Query Result Width>]
    [,[,@query_result_separator =] '<Query Result Separator>']
```

```
    [,[,@exclude_query_output =] <Exclude Query Output>]
    [,[,@append_query_error =] <Append Query Error>]
    [,[,@query_no_truncate =] <Query No Truncate>]
    [,[,@query_result_no_padding =] @<Parameter for Query Result No Padding>]
    [,[,@mailitem_id =] <Mail item id>] [,OUTPUT]
```

Examples

Create a Database Mail account.

```
EXECUTE msdb.dbo.sysmail_add_account_sp
    @account_name = 'MailAccount1',
    @description = 'Mail account for testing DB Mail',
    @email_address = 'Address@MyDomain.com',
    @replyto_address = 'ReplyAddress@MyDomain.com',
    @display_name = 'Mailer for registration messages',
    @mailserver_name = 'smtp.MyDomain.com' ;
```

Create a Database Mail profile.

```
EXECUTE msdb.dbo.sysmail_add_profile_sp
    @profile_name = 'MailAccount1 Profile',
    @description = 'Mail Profile for testing DB Mail' ;
```

Associate the account with the profile.

```
EXECUTE msdb.dbo.sysmail_add_profileaccount_sp
    @profile_name = 'MailAccount1 Profile',
    @account_name = 'MailAccount1',
    @sequence_number =1 ;
```

Grant the profile access to the `DBMailUsers` role.

```
EXECUTE msdb.dbo.sysmail_add_principalprofile_sp
    @profile_name = 'MailAccount1 Profile',
    @principal_name = 'ApplicationUser',
    @is_default = 1 ;
```

Send a message with `sp_db_sendmail`.

```
EXEC msdb.dbo.sp_send_dbmail
```

```
    @profile_name = 'MailAccount1 Profile',
    @recipients = 'Recipient@Mydomain.com',
    @query = 'SELECT * FROM fn_WeeklySalesReport(GETDATE())',
    @subject = 'Weekly Sales Report',
    @attach_query_result_as_file = 1 ;
```

For more information, see Database Mail in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) doesn't provide native support for sending email message from the database. For alerting purposes, use the Event Notification Subscription feature to send email notifications to operators. For more information, see Alerting.

The only way to send an Email from the database is to use AWS Lambda integration. For more information, see AWS Lambda.

Examples

The following walkthrough shows how to send an Email from Aurora PostgreSQL using AWS Lambda integration.

First, configure Amazon Simple Email Service (Amazon SES). For more information, see What is Amazon SES? in the *Amazon Simple Email Service Developer Guide*.

1. In the AWS console, choose **SES**, **SMTP Settings**, and then choose **Create My SMTP Credentials**. Copy the SMTP server name, which you will use in the AWS Lambda function.
2. For **IAM User Name**, enter the SMTP user name, and then choose **Create**.
3. Save the credentials, which you will use to authenticate with the SMTP server. After you leave this page, you can't retrieve these credentials.
4. In the AWS console, choose **SES**, **Email Addresses**, and then choose **Verify a New Email Address**. Before you send emails, verify the email address.
5. After you verify the email, create a table to store messages to be sent by the AWS Lambda function.

```
CREATE TABLE emails (title varchar(600), body varchar(600), recipients varchar(600));
```

6. In the AWS console, choose **Lambda**, and then choose **Create function**.

7. Select **Author from scratch**, enter a name for your project, and select Python 2.7 as the runtime. Make sure that you use a role with the correct permissions. Choose **Create function**.
8. Download this GitHub project.
9. In your local environment, create two files: `main.py` and `db_util.py`. Copy and paste the following content into these files. Make sure that you replace the code placeholders with values for your environment.

main.py

```
#!/usr/bin/python
import sys
import logging
import psycopg2

from db_util import make_conn, fetch_data
def lambda_handler(event, context):
    query_cmd = "select * from mails"
    print query_cmd

    # get a connection, if a connect can't be made an exception will be raised here
    conn = make_conn()

    result = fetch_data(conn, query_cmd)
    conn.close()

    return result
```

db_util.py:

```
#!/usr/bin/python
import psycopg2
import smtplib
import email.utils
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

db_host = 'YOUR_RDS_HOST'
db_port = 'YOUR_RDS_PORT'
db_name = 'YOUR_RDS_DBNAME'
db_user = 'YOUR_RDS_USER'
db_pass = 'YOUR_RDS_PASSWORD'
```

```
def sendEmail(recp, sub, message):
    # Replace sender@example.com with your "From" address.
    # This address must be verified.
    SENDER = 'PUT HERE THE VERIFIED EMAIL'
    SENDERNAME = 'Lambda'

    # Replace recipient@example.com with a "To" address. If your account
    # is still in the sandbox, this address must be verified.
    RECIPIENT = recp

    # Replace smtp_username with your Amazon SES SMTP user name.
    USERNAME_SMTP = "YOUR_SMTP_USERNAME"

    # Replace smtp_password with your Amazon SES SMTP password.
    PASSWORD_SMTP = "YOUR_SMTP PASSWORD"

    # (Optional) the name of a configuration set to use for this message.
    # If you comment out this line, you also need to remove or comment out
    # the "X-SES-CONFIGURATION-SET:" header.
    CONFIGURATION_SET = "ConfigSet"

    # If you're using Amazon SES in a region other than US West (Oregon),
    # replace email-smtp.us-west-2.amazonaws.com with the Amazon SES SMTP
    # endpoint in the appropriate region.
    HOST = "YOUR_SMTP_SERVERNAME"
    PORT = 587

    # The subject line of the email.
    SUBJECT = sub

    # The email body for recipients with non-HTML email clients.
    BODY_TEXT = ("Amazon SES Test\r\n"
        "This email was sent through the Amazon SES SMTP "
        "Interface using the Python smtplib package."
    )

    # The HTML body of the email.
    BODY_HTML = """<html>
    <head></head>
    <body>
    <h1>Amazon SES SMTP Email Test</h1>""" + message + """</body>
    </html>
```

```
        """

    # Create message container - the correct MIME type is multipart/alternative.
    msg = MIMEMultipart('alternative')
    msg['Subject'] = SUBJECT
    msg['From'] = email.utils.formataddr((SENDERNAME, SENDER))
    msg['To'] = RECIPIENT
    # Comment or delete the next line if you aren't using a configuration set
    #msg.add_header('X-SES-CONFIGURATION-SET',CONFIGURATION_SET)

    # Record the MIME types of both parts - text/plain and text/html.
    part1 = MIMEText(BODY_TEXT, 'plain')
    part2 = MIMEText(BODY_HTML, 'html')

    # Attach parts into message container.
    # According to RFC 2046, the last part of a multipart message, in this case
    # the HTML message, is best and preferred.
    msg.attach(part1)
    msg.attach(part2)

    # Try to send the message.
    try:
        server = smtplib.SMTP(HOST, PORT)
        server.ehlo()
        server.starttls()
        #stmplib docs recommend calling ehlo() before & after starttls()
        server.ehlo()
        server.login(USERNAME_SMTP, PASSWORD_SMTP)
        server.sendmail(SENDER, RECIPIENT, msg.as_string())
        server.close()

    # Display an error message if something goes wrong.
    except Exception as e:
        print ("Error: ", e)
    else:
        print ("Email sent!")

def make_conn():
    conn = None
    try:
        conn = psycopg2.connect("dbname='%s' user='%s' host='%s' password='%s'" %
 (db_name, db_user, db_host, db_pass))
    except:
```

```
        print "I am unable to connect to the database"
    return conn

def fetch_data(conn, query):
    result = []
    print "Now running: %s" % (query)
    cursor = conn.cursor()
    cursor.execute(query)

    print("Number of new mails to be sent: ", cursor.rowcount)

    raw = cursor.fetchall()

    for line in raw:
        print(line[0])
        sendEmail(line[2],line[0],line[1])
        result.append(line)

    cursor.execute('delete from mails')
    cursor.execute('commit')

    return result
```

> **Note**
>
> In the body of `db_util.py`, AWS Lambda deletes the content of the mails table.

10. Place the `main.py` and `db_util.py` files inside the GitHub extracted folder and create a new archive file using the ZIP file format that includes your two new files.
11. Return to your Lambda project and change the **Code entry type** to **Upload a .ZIP file**, change the Handler to `mail.lambda_handler`, and upload the file. Choose Save.
12. To test the lambda function, choose **Test** and enter the **Event name**.

> **Note**
>
> You can trigger the AWS Lambda function by multiple options. This walkthrough demonstrates how to schedule it to run every minute. Remember, you are paying for each AWS Lambda run.

13. To create a scheduled trigger, use Amazon CloudWatch, enter all details, and choose **Add**.

> **Note**
>
> This example runs every minute, but you can use a different interval. For more information, see Schedule expressions using rate or cron.

14. Choose **Save**.

ETL

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Use AWS Glue for ETL.

SQL Server Usage

SQL Server offers a native extract, transform, and load (ETL) framework of tools and services to support enterprise ETL requirements. The legacy Data Transformation Services (DTS) has been deprecated as of SQL Server 2008 and replaced with SQL Server Integration Services (SSIS), which was introduced in SQL Server 2005. For more information, see Data Transformation Services (DTS) in the *SQL Server documentation*.

DTS

DTS was introduced in SQL Server version 7 in 1998. It was significantly expanded in SQL Server 2000 with features such as FTP, database level operations, and Microsoft Message Queuing (MSMQ) integration. It included a set of objects, utilities, and services that enabled easy, visual construction of complex ETL operations across heterogeneous data sources and targets.

DTS supported OLE DB, ODBC, and text file drivers. It allowed transformations to be scheduled using SQL Server Agent. For more information, see SQL Server Agent. DTS also provided version control and backup capabilities with version control systems such as Microsoft Visual SourceSafe.

The fundamental entity in DTS was the DTS Package. Packages were the logical containers for DTS objects such as connections, data transfers, transformations, and notifications. The DTS framework also included the following tools:

- DTS Wizards.
- DTS Package Designers.
- DTS Query Designer.
- DTS Run Utility.

SSIS

The SSIS framework was introduced in SQL Server 2005, but was limited to the top-tier editions only, unlike DTS which was available with all editions.

SSIS has evolved over DTS to offer a true modern, enterprise class, heterogeneous platform for a broad range of data migration and processing tasks. It provides a rich workflow-oriented design with features for all types of enterprise data warehousing. It also supports scheduling capabilities for multi-dimensional cubes management.

SSIS provides the following tools:

- SSIS Import/Export Wizard is an SQL Server Management Studio extension that enables quick creation of packages for moving data between a wide array of sources and destinations. However, it has limited transformation capabilities.
- SQL Server Business Intelligence Development Studio (BIDS) is a developer tool for creating complex packages and transformations. It provides the ability to integrate procedural code into package transformations and provides a scripting environment. Recently, BIDS has been replaced by SQL Server Data Tools - Business intelligence (SSDT-BI).

SSIS objects include:

- Connections.
- Event handlers.
- Workflows.
- Error handlers.

- Parameters (starting with SQL Server 2012).
- Precedence constraints.
- Tasks.
- Variables.

SSIS packages are constructed as XML documents and you can save them to the file system or store within a SQL Server instance using a hierarchical name space.

For more information, see SQL Server Integration Services in the *SQL Server documentation* and Data Transformation Services in *Wikipedia*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) provides AWS Glue for enterprise class extract, transform, and load (ETL). It is a fully managed service that performs data cataloging, cleansing, enriching, and movement between heterogeneous data sources and destinations. Being a fully managed service, the user doesn't need to be concerned with infrastructure management.

AWS Glue Key Features

Integrated data catalog

The AWS Glue Data Catalog is a persistent metadata store, that you can use to store all data assets, whether in the cloud or on-premises. It stores table schemas, job steps, and additional meta data information for managing these processes. AWS Glue can automatically calculate statistics and register partitions to make queries more efficient. It maintains a comprehensive schema version history for tracking changes over time.

Automatic schema discovery

AWS Glue provides automatic crawlers that can connect to source or target data providers. The crawler uses a prioritized list of classifiers to determine the schema for your data and then generates and stores the metadata in the AWS Glue Data Catalog. You can schedule crawlers or run on-demand. You can also trigger a crawler when an event occurs to keep metadata current.

Code generation

AWS Glue automatically generates the code to extract, transform, and load data. All you need to do is point Glue to your data source and target. The ETL scripts to transform, flatten, and enrich data are created automatically. You can generate AWS Glue scripts in Scala or Python and use them in Apache Spark.

Developer endpoints

When interactively developing AWS Glue ETL code, AWS Glue provides development endpoints for editing, debugging, and testing. You can use any IDE or text editor for ETL development. You can import custom readers, writers, and transformations into Glue ETL jobs as libraries. You can also use and share code with other developers in the AWS Glue GitHub repository.

Flexible job scheduler

You can trigger AWS Glue jobs for running either on a pre-defined schedule, on-demand, or as a response to an event.

You can start multiple jobs in parallel and explicitly define dependencies across jobs to build complex ETL pipelines. AWS Glue handles all inter-job dependencies, filters bad data, and retries failed jobs. All logs and notifications are pushed to Amazon CloudWatch; you can monitor and get alerts from a central service.

Migration Considerations

You can use AWS Schema Conversion Tool (AWS SCT) to convert your Microsoft SSIS ETL scripts to AWS Glue. For more information, see Converting SSIS.

Examples

The following walkthrough describes how to create an AWS Glue job to upload a comma-separated values (CSV) file from Amazon S3 to Aurora PostgreSQL.

The source file for this walkthrough is a simple Visits table in CSV format. The objective is to upload this file to an Amazon S3 bucket and create an AWS Glue job to discover and copy it into an Aurora PostgreSQL database.

Step 1 — Create a Bucket in Amazon S3 and Upload the CSV File

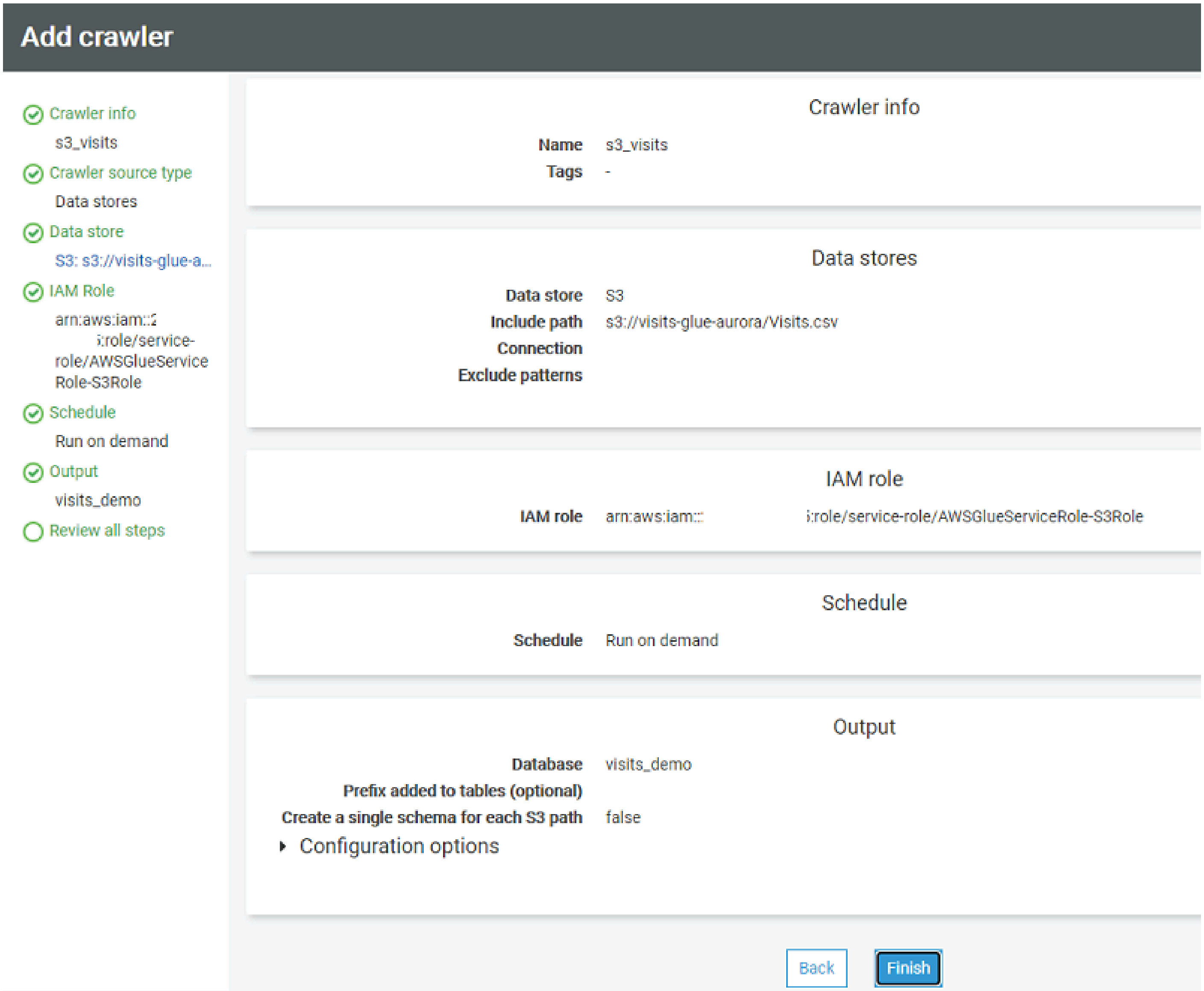
1. In the AWS console, choose **S3**, and then choose **Create bucket**.

> **Note**
>
> This walkthrough demonstrates how to create the buckets and upload the files manually, which is automated using the Amazon S3 API for production ETLs. Using the console to manually run all the settings will help you get familiar with the terminology, concepts, and workflow.

2. Enter a unique name for the bucket, select a region, and define the level of access.
3. Turn on versioning, add tags, turn on server-side encryption, and choose **Create bucket**.
4. On the Amazon S3 Management Console, choose the newly created bucket.
5. On the bucket page, choose **Upload**.
6. Choose **Add files**, select your CSV file, and choose **Upload**.

Step 2 — Add an Amazon Glue Crawler to Discover and Catalog the Visits File

1. In the AWS console, choose **AWS Glue** .
2. Choose **Tables**, and then choose **Add tables using a crawler**.
3. Enter the name of the crawler and choose **Next**.
4. On the **Specify crawler source type** page, leave the default values, and choose **Next**.
5. On the **Add a data store** page, specify a valid Amazon S3 path, and choose **Next**.
6. On the **Choose an IAM role** page, choose an existing IAM role, or create a new IAM role. Choose **Next**.
7. On the **Create a schedule for this crawler** page, choose **Run on demand**, and choose **Next**.
8. On the **Configure the crawler's output** page, choose a database for the crawler's output, enter an optional table prefix for easy reference, and choose **Next**.
9. Review the information that you provided and choose **Finish** to create the crawler.



Step 3 — Run the Amazon Glue Crawler

1. In the AWS console, choose **AWS Glue**, and then choose **Crawlers**.
2. Choose the crawler that you created on the previous step, and choose **Run crawler**.

After the crawler completes, the table should be discovered and recorded in the catalog in the table specified.

Click the link to get to the table that was just discovered and then click the table name.

Verify the crawler identified the table's properties and schema correctly.

Note

You can manually adjust the properties and schema JSON files using the buttons on the top right.

If you don't want to add a crawler, you can add tables manually.

1. In the AWS console, choose **AWS Glue** .
2. Choose **Tables**, and then choose **Add table manually**.

Step 4 — Create an ETL Job to Copy the Visits Table to an Aurora PostgreSQL Database

1. In the AWS console, choose **AWS Glue** .
2. Choose **Jobs (legacy)**, and then choose **Add job**.
3. Enter a name for the ETL job and pick a role for the security context. For this example, use the same role created for the crawler. The job may consist of a pre-existing ETL script, a manually-authored script, or an automatic script generated by Amazon Glue. For this example, use Amazon Glue. Enter a name for the script file or accept the default, which is also the job's name. Configure advanced properties and parameters if needed and choose **Next**.
4. Select the data source for the job and choose **Next**.
5. On the **Choose a transform type** page, choose **Change schema**.
6. On the **Choose a data target** page, choose **Create tables in your data target**, use the JDBC Data store, and the `gluerds` connection type. Choose **Add Connection**.
7. On the **Add connection** page, enter the access details for the Amazon Aurora Instance and choose **Add**.
8. Choose **Next** to display the column mapping between the source and target. Leave the default mapping and data types, and choose **Next**.
9. Review the job properties and choose **Save job and edit script**.
10. Review the generated script and make manual changes if needed. You can use the built-in templates for source, target, target location, transform, and spigot using the buttons at the top right section of the screen.

11. Choose **Run job**.
12. In the AWS console, choose **AWS Glue**, and then choose **Jobs (legacy)**.
13. On the history tab, verify that the job status is set to **Succeeded**.
14. Open your query IDE, connect to the Aurora PostgreSQL cluster, and query the visits database to make sure the data has been transferred successfully.

For more information, see AWS Glue Developer Guide and AWS Glue resources.

SQL Server Export and Import with Text Files and PostgreSQL pg_dump and pg_restore

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Non-compatible tool.

SQL Server Usage

SQL Server provides many options for exporting and importing text files. These operations are commonly used for data migration, scripting, and backup.

- Save results to a file in SQL Server Management Studio (SSMS). For more information, see KB - How to create .csv or .rpt files from an SQL statement in Microsoft SQL Server in the *SQL Server documentation*. l SQLCMD. For more information, see Run the script file in the *SQL Server documentation*. l PowerShell wrapper for SQLCMD l SSMS Import/Export Wizard. For more information, see Start the SQL Server Import and Export Wizard in the *SQL Server documentation*. l SQL Server Reporting Services (SSRS) l Bulk Copy Program (BCP). For more information, see Import and export bulk data using bcp (SQL Server) in the *SQL Server documentation*.

All of the options described before required additional tools to export data. Most of the tools are open source and provide support for a variety of databases.

SQLCMD is a command line utility for running T-SQL statements, system procedures, and script files. It uses ODBC to run T-SQL batches. For example:

```
SQLCMD -i C:\sql\myquery.sql -o C:\sql\output.txt
```

SQLCMD utility syntax:

```
sqlcmd
    -a packet_size
    -A (dedicated administrator connection)
    -b (terminate batch job if there is an error)
    -c batch_terminator
    -C (trust the server certificate)
    -d db_name
    -e (echo input)
    -E (use trusted connection)
    -f codepage | i:codepage[,o:codepage] | o:codepage[,i:codepage]
    -g (enable column encryption)
    -G (use Azure Active Directory for authentication)
    -h rows_per_header
    -H workstation_name
    -i input_file
    -I (enable quoted identifiers)
    -j (Print raw error messages)
    -k[1 | 2] (remove or replace control characters)
    -K application_intent
    -l login_timeout
    -L[c] (list servers, optional clean output)
    -m error_level
    -M multisubnet_failover
    -N (encrypt connection)
    -o output_file
    -p[1] (print statistics, optional colon format)
    -P password
    -q "cmdline query"
    -Q "cmdline query" (and exit)
    -r[0 | 1] (msgs to stderr)
    -R (use client regional settings)
    -s col_separator
    -S [protocol:]server[instance_name][,port]
    -t query_timeout
    -u (unicode output file)
```

```
    -U login_id
    -v var = "value"
    -V error_severity_level
    -w column_width
    -W (remove trailing spaces)
    -x (disable variable substitution)
    -X[1] (disable commands, startup script, environment variables, optional exit)
    -y variable_length_type_display_width
    -Y fixed_length_type_display_width
    -z new_password
    -Z new_password (and exit)
    -? (usage)
```

Examples

Connect to a named instance using Windows Authentication and specify input and output files.

```
sqlcmd -S MyMSSQLServer\MyMSSQLInstance -i query.sql -o outputfile.txt
```

If the file is needed for import to another database, query the data as `INSERT` commands and `CREATE` for the object.

You can export data with SQLCMD and import with the Export/Import wizard.

For more information, see [sqlcmd Utility](#) in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL provides the native utilities pg_dump and `pg_restore` to perform logical database exports and imports with comparable functionality to the SQl Server SQLCMD utility. For example, moving data between two databases and creating logical database backups.

- **pg_dump** to export data.
- **pg_restore** to import data.

The binaries for both utilities must be installed on your local workstation or on an Amazon EC2 server as part of the PostgreSQL client binaries.

You can export and copy PostgreSQL dump files created using pg_dump to an Amazon S3 bucket as cloud backup storage or for maintaining the desired backup retention policy. Later, when you need the dump files for database restore, you can copy them copied back to a desktop or server that has a PostgreSQL client, such as your workstation or an Amazon EC2 server. Then you can issue the `pg_restore` command.

Starting with PostgreSQL 10, these capabilities were added:

- You can exclude a schema in `pg_dump` and `pg_restore` commands.
- Can create dumps with no blobs.
- Allow to run `pg_dumpall` by non-superusers, using the `--no-role-passwords` option.
- Create additional integrity option to ensure that the data is stored in disk using `fsync()` method.

Starting with PostgreSQL 11, the following capabilities were added: * `pg_dump` and `pg_restore` now export or import relationships between extensions and database objects established with `ALTER … DEPENDS ON EXTENSION` , which allows these objects to be dropped when extension is dropped with `CASCADE` option.

Notes

- `pg_dump` creates consistent backups even if the database is being used concurrently.
- `pg_dump` doesn't block other users accessing the database (readers or writers).
- `pg_dump` only exports a single database. To backup global objects common to all databases in a cluster (such as roles and tablespaces), use `pg_dumpall`.
- PostgreSQL dump files can be plain-text and custom format files.

Another option to export and import data from PostgreSQL database is to use `COPY TO/COPY FROM` commands. Starting with PostgreSQL 12, you can use the `COPY FROM` command to load data into DB. This command has support for filtering incoming rows with the `WHERE` condition.

```
CREATE TABLE tst_copy(v TEXT);

COPY tst_copy FROM '/home/postgres/file.csv' WITH (FORMAT CSV) WHERE v LIKE '%apple%';
```

Examples

Export data using pg_dump. Use a workstation or server with the PostgreSQL client installed to connect to the Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) instance. Issue the pg_dump command providing the hostname (-h), database user name (-U), and database name (-d).

```
$ pg_dump -h hostname.rds.amazonaws.com -U username -d db_name -f dump_file_name.sql
```

The output `dump_file_name.sql` file is stored on the server where the pg_dump command runs. You can copy the output file to an Amazon S3 bucket if needed.

Run pg_dump and copy the backup file to an Amazon S3 bucket using a pipe and the AWS CLI.

```
$ pg_dump -h hostname.rds.amazonaws.com -U username -d db_name -f dump_file_name.sql |
 aws s3 cp - s3://pg-backup/pg_bck-$(date"+%Y-%m-%d-%H-%M-%S")
```

Restore data using `pg_restore`. Use a workstation or server with the PostgreSQL client installed to connect to the Aurora PostgreSQL instance. Issue the `pg_restore` command providing the hostname (-h), database user name (-U), database name (-d), and the dump file.

```
$ pg_restore -h hostname.rds.amazonaws.com -U username -d dbname_restore
 dump_file_name.sql
```

Copy the output file from the local server to an Amazon S3 Bucket using the AWS CLI. Upload the dump file to an Amazon S3 bucket.

```
$ aws s3 cp /usr/Exports/hr.dmp s3://my-bucket/backup-$(date "+%Y-%m-%d-%H-%M-%S")
```

Note

The `{-$(date "+%Y-%m-%d-%H-%M-%S")}` format is valid on Linux servers only.

Download the output file from the Amazon S3 bucket.

```
$ aws s3 cp s3://my-bucket/backup-2017-09-10-01-10-10 /usr/Exports/hr.dmp
```

> **Note**
>
> You can create a copy of an existing database without having to use pg_dump or `pg_restore`. Instead, use the template keyword to specify the source database.

```
CREATE DATABASE mydb_copy TEPLATE mydb;
```

Summary

Description	SQL Server export / import
PostgreSQL Dump	Export data to a file
Using SQLCMD or Export/Import Wizard `SQLCMD -i C:\sql\myquery.sql -o C:\sql \output.txt`	`pg_dump -F c -h hostname.rds.amazo naws.com -U username -d hr -p 5432 > c: \Export\hr.dmp`
Import data to a new database with a new name	Run SQLCMD with objects and data creation script `SQLCMD -i C:\sql\myquery.sql`

For more information, see SQL Dump and pg_restore in the *PostgreSQL documentation*.

Viewing Server Logs

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	View logs from the Amazon RDS console, the Amazon RDS API,

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
			the AWS CLI, or the AWS SDKs.

SQL Server Usage

SQL Server logs system and user generated events to the *SQL Server Error Log* and to the *Windows Application Log*. It logs recovery messages, kernel messages, security events, maintenance events, and other general server level error and informational messages. The Windows Application Log contains events from all windows applications including SQL Server and SQL Server agent.

SQL Server Management Studio Log Viewer unifies all logs into a single consolidated view. You can also view the logs with any text editor.

Administrators typically use the SQL Server Error Log to confirm successful completion of processes, such as backup or batches, and to investigate the cause of run time errors. These logs can help detect current risks or potential future problem areas.

To view the log for SQL Server, SQL Server Agent, Database Mail, and Windows applications, open the SQL Server Management Studio Object Explorer pane, navigate to **Management**, **SQL Server Logs**, and choose the current log.

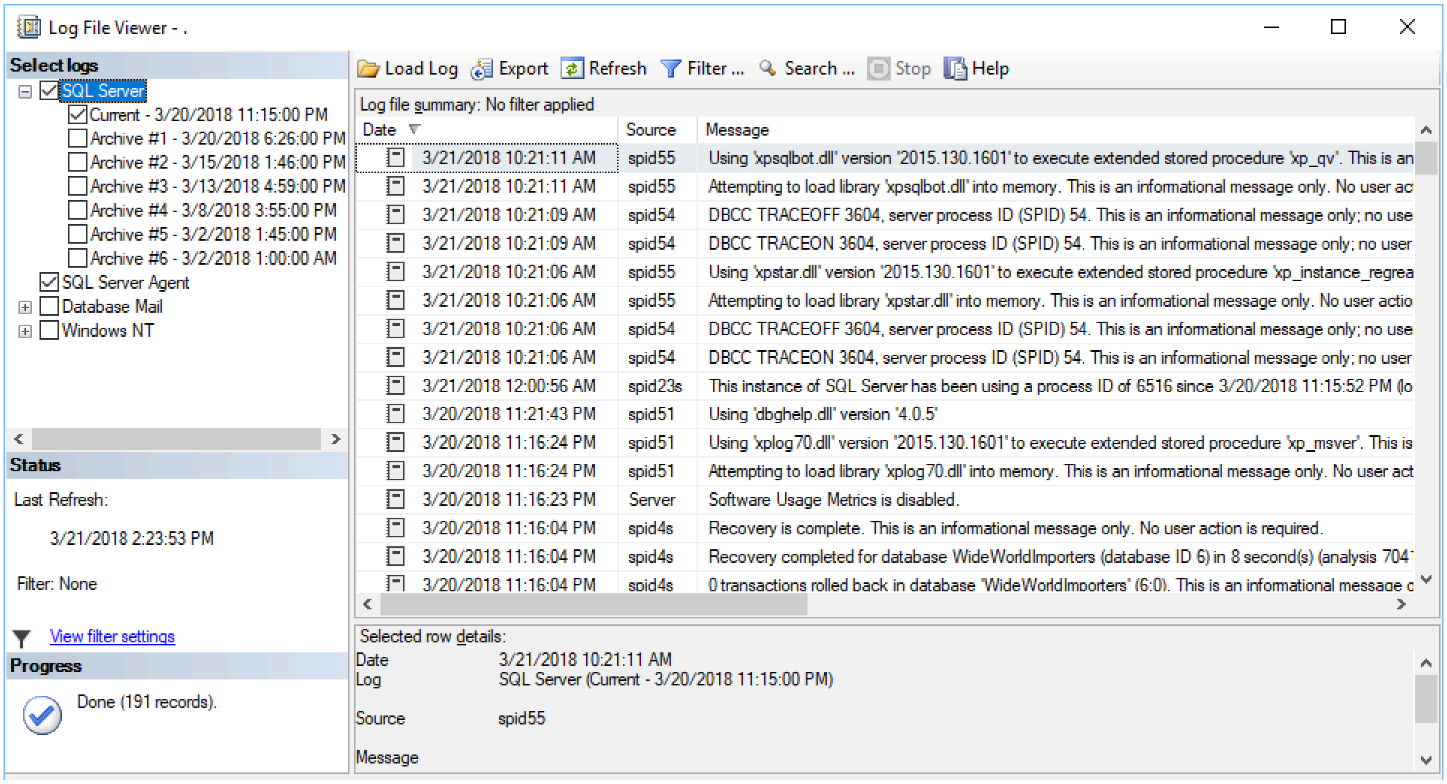
The following table identifies some common error codes database administrators typically look for in the error logs:

Error code	Error message
1105	Couldn’t allocate space.
3041	Backup failed.
9002	Transaction log full.
14151	Replication agent failed.
17053	Operating system error.

Error code	Error message
18452	Login failed.
9003	Possible database corruption.

Examples

The following screenshot shows the typical log file viewer content:



For more information, see [Monitoring the Error Logs](https://docs.microsoft.com) in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) provides administrators with access to the PostgreSQL error log.

The PostgreSQL error log is generated by default. To generate the slow query and general logs, set the corresponding parameters in the database parameter group. For more information, see Server Options in SQL Server and Parameter Groups in Amazon Aurora.

You can view Aurora PostgreSQL logs directly from the Amazon RDS console, the Amazon RDS API, the AWS CLI, or the AWS SDKs. You can also direct the logs to a database table in the main database and use SQL queries to view the data. To download a binary log, use the AWS Console.

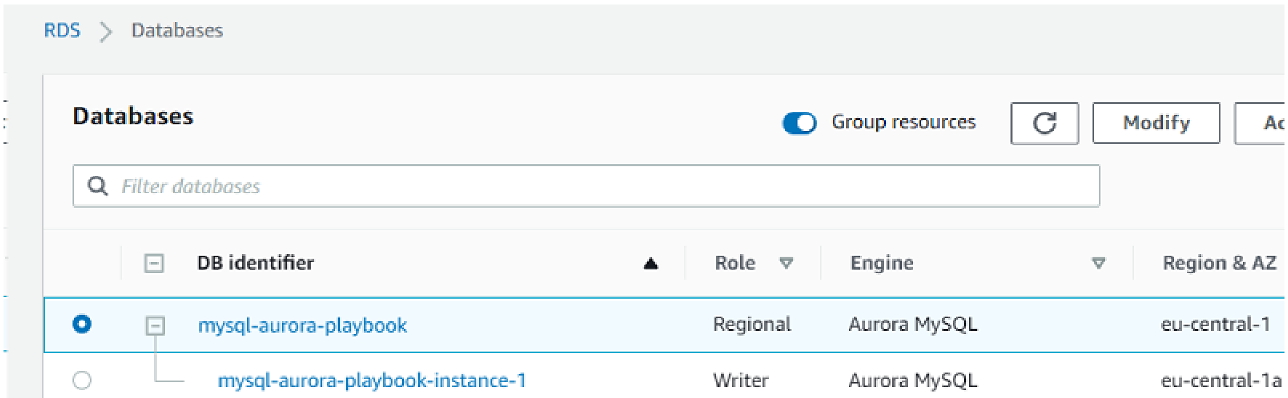
The following table includes the parameters, which control how and where PostgreSQL places log and errors files.

Parameter	Description
`log_filename`	Sets the file name pattern for log files. You can modify this parameter in an Aurora Database Parameter Group.
`log_rotation_age`	(min) Automatic log file rotation will occur after N minutes. You can modify this parameter in an Aurora Database Parameter Group.
`log_rotation_size`	(kB) Automatic log file rotation will occur after N kilobytes. You can modify this parameter in an Aurora Database Parameter Group.
`log_min_messages`	Sets the message levels that are logged such as `DEBUG`, `ERROR`, `INFO`, and so on. You can modify this parameter in an Aurora Database Parameter Group.
`log_min_error_statement`	Causes all statements generating errors at or above this level to be logged such as `DEBUG`, `ERROR`, `INFO`, and so on. You can modify this parameter in an Aurora Database Parameter Group.
`log_min_duration_statement`	Sets the minimum run time above which statements will be logged (ms). You can modify this parameter in an Aurora Database Parameter Group.

Examples

The following walkthrough demonstrates how to view the Aurora PostgreSQL error logs in the Amazon RDS console.

1. In the AWS console, choose **RDS**, and then choose **Databases**.
2. Choose the instance for which you want to view the error log.



3. Scroll down to the logs section and choose the log name. The log viewer displays the log content.

For more information, see PostgreSQL database log files in the *Amazon Relational Database Service User Guide*.

Maintenance Plans

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Backups using the Amazon RDS services. Table maintenance using SQL commands.

SQL Server Usage

A *maintenance plan* is a set of automated tasks used to optimize a database, performs regular backups, and ensure it is free of inconsistencies. Maintenance plans are implemented as SQL Server Integration Services (SSIS) packages and are run by SQL Server Agent jobs. You can run them manually or automatically at scheduled time intervals.

SQL Server provides a variety of pre-configured maintenance tasks. You can create custom tasks using TSQL scripts or operating system batch files.

Maintenance plans are typically used for the following tasks:

- Backing up database and transaction log files.
- Performing cleanup of database backup files in accordance with retention policies.
- Performing database consistency checks.
- Rebuilding or reorganizing indexes.
- Decreasing data file size by removing empty pages (shrink a database).
- Updating statistics to help the query optimizer obtain updated data distributions.
- Running SQL Server Agent jobs for custom actions.
- Running a T-SQL task.

Maintenance plans can include tasks for operator notifications and history or maintenance cleanup. They can also generate reports and output the contents to a text file or the maintenance plan tables in the `msdb` database.

You can create and manage maintenance plans using the maintenance plan wizard in SQL Server Management Studio, Maintenance Plan Design Surface (provides enhanced functionality over the wizard), Management Studio Object Explorer, and T-SQL system stored procedures.

For more information, see SQL Server Agent and PostgreSQL Scheduled Lambda.

Deprecated DBCC Index and Table Maintenance Commands

The DBCC DBREINDEX, INDEXDEFRAG, and SHOWCONTIG commands have been deprecated as of SQL Server 2008R2. For more information, see Deprecated Database Engine Features in SQL Server 2008 R2 in the *SQL Server documentation*.

In place of the deprecated DBCC, SQL Server provides newer syntax alternatives as detailed in the following table.

Deprecated DBCC command	Use instead
DBCC DBREINDEX	ALTER INDEX … REBUILD
DBCC INDEXDEFRAG	ALTER INDEX … REORGANIZE
DBCC SHOWCONTIG	sys.dm_db_index_physical_stats

For the Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) alternatives to these maintenance commands, see Aurora PostgreSQL Maintenance Plans.

Examples

Enable Agent XPs, which are disabled by default.

```
EXEC [sys].[sp_configure] @configname = 'show advanced options', @configvalue = 1
 RECONFIGURE ;
```

```
EXEC [sys].[sp_configure] @configname = 'agent xps', @configvalue = 1 RECONFIGURE;
```

Create a T-SQL maintenance plan for a single index rebuild.

```
USE msdb;
```

Add the Index Maintenance `IDX1` job to SQL Server Agent.

```
EXEC dbo.sp_add_job @job_name = N'Index Maintenance IDX1', @enabled = 1, @description =
 N'Optimize IDX1 for INSERT' ;
```

Add the T-SQL job step `Rebuild IDX1 to 50 percent fill`.

```
EXEC dbo.sp_add_jobstep @job_name = N'Index Maintenance IDX1', @step_name = N'Rebuild
 IDX1 to 50 percent fill', @subsystem = N'TSQL',
```

```
@command = N'Use MyDatabase; ALTER INDEX IDX1 ON Shcema.Table REBUILD WITH
 ( FILL_FACTOR = 50), @retry_attempts = 5, @retry_interval = 5;
```

Add a schedule to run every day at 01:00 AM.

```
EXEC dbo.sp_add_schedule @schedule_name = N'Daily0100', @freq_type = 4, @freq_interval
 = 1, @active_start_time = 010000;
```

Associate the schedule `Daily0100` with the job index maintenance `IDX1`.

```
EXEC sp_attach_schedule @job_name = N'Index Maintenance IDX1' @schedule_name =
 N'Daily0100' ;
```

For more information, see Maintenance Plans in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Relational Database Service (Amazon RDS) performs automated database backups by creating storage volume snapshots that back up entire instances, not individual databases.

Amazon RDS creates snapshots during the backup window for individual database instances and retains snapshots in accordance with the backup retention period. You can use the snapshots to restore a database to any point in time within the backup retention period.

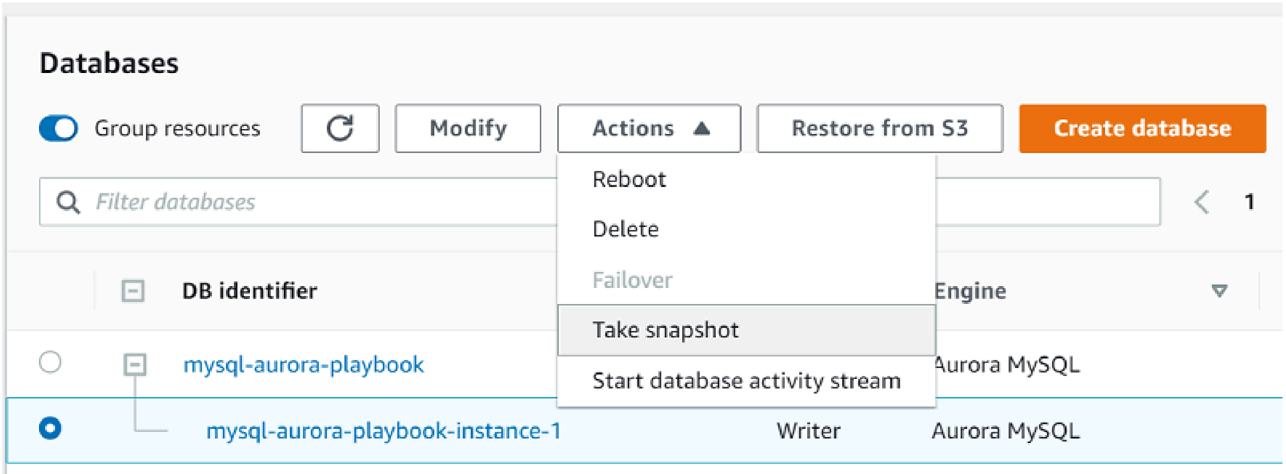
> **Note**
>
> The state of a database instance must be ACTIVE for automated backups to occur.

You can backup database instances manually by creating an explicit database snapshot. Use the AWS console, the AWS CLI, or the AWS API to take manual snapshots.

Examples

Create a manual database snapshot using the Amazon RDS console

1. In the AWS console, choose **RDS**, and then choose **Databases**.
2. Choose your Aurora PostgreSQL instance, and for **Instance actions** choose **Take snapshot**.



Restore a snapshot using the Amazon RDS console

1. In the AWS console, choose **RDS**, and then choose **Snapshots**.
2. Choose the snapshot to restore, and for **Actions** choose **Restore snapshot**.

   This action creates a new instance.
3. Enter the required configuration options in the wizard for creating a new Amazon Aurora database instance. Choose **Restore DB Instance**.

You can also restore a database instance to a point-in-time. For more information, see Backup and Restore.

For all other tasks, use a third-party or a custom application scheduler.

Rebuild and reorganize a table

Aurora PostgreSQL supports the `VACUUM`, `ANALYZE`, and `REINDEX` commands, which are similar to the `REORGANIZE` option of SQL Server indexes.

```
VACUUM MyTable;
ANALYZE MyTable;
REINDEX TABLE MyTable;
```

- `VACUUM` reclaims storage.
- `ANALYZE` collects statistics.

- REINDEX recreates all indexes.

For more information, see ANALYZE, VACUUM, and REINDEX in the *PostgreSQL documentation*.

Convert deprecated DBCC index and table maintenance commands

Deprecated DBCC command	Aurora PostgreSQL equivalent
DBCC DBREINDEX	REINDEX INDEX or REINDEX TABLE
DBCC INDEXDEFRAG	VACUUM table_name or VACUUM table_name column_name

Update statistics to help the query optimizer get updated data distribution

For more information, see SQL Server Managing Statistics and PostgreSQL Table Statistics.

Summary

The following table summarizes the key tasks that use SQL Server maintenance plans and a comparable Aurora PostgreSQL solutions.

Task	SQL Server	Aurora PostgreSQL
Rebuild or reorganize indexes	ALTER INDEX or ALTER TABLE	REINDEX INDEX or REINDEX TABLE
Decrease data file size by removing empty pages	DBCC SHRINKDATABASE or DBCC SHRINKFILE	VACUUM
Update statistics to help the query optimizer get updated data distribution	UPDATE STATISTICS or sp_updatestats	ANALYZE
Perform database consistency checks	DBCC CHECKDB or DBCC CHECKTABLE	N/A

Task	SQL Server	Aurora PostgreSQL
Back up the database and transaction log files	`BACKUP DATABASE` or `BACKUP LOG`	Automatically (for example, using AWS CLI)
Run SQL Server Agent jobs for custom actions	`sp_start_job` or `scheduled`	N/A

For more information, see Working with backups in the *PostgreSQL documentation*.

Monitoring

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Use Amazon CloudWatch service. For more information, see Monitoring metrics in an Amazon RDS instance in the *Amazon Relational Database Service User Guide*.

SQL Server Usage

Monitoring server performance and behavior is a critical aspect of maintaining service quality and includes ad-hoc data collection, ongoing data collection, root cause analysis, preventative actions, and reactive actions. SQL Server provides an array of interfaces to monitor and collect server data.

SQL Server 2017 introduces several new dynamic management views:

- `sys.dm_db_log_stats` exposes summary level attributes and information on transaction log files, helpful for monitoring transaction log health.

- `sys.dm_tran_version_store_space_usage` tracks version store usage for each database, useful for proactively planning `tempdb` sizing based on the version store usage for each database.
- `sys.dm_db_log_info` exposes VLF information to monitor, alert, and avert potential transaction log issues.
- `sys.dm_db_stats_histogram` is a new dynamic management view for examining statistics.
- `sys.dm_os_host_info` provides operating system information for both Windows and Linux.

SQL Server 2019 adds new configuration parameter, `LIGHTWEIGHT_QUERY_PROFILING`. It turns on or turns off the lightweight query profiling infrastructure. The lightweight query profiling infrastructure (LWP) provides query performance data more efficiently than standard profiling mechanisms and is enabled by default. For more information, see Query Profiling Infrastructure in the *SQL Server documentation*.

Windows Operating System Level Tools

You can use the Windows Scheduler to trigger run of script files such as CMD, PowerShell, and so on to collect, store, and process performance data.

System Monitor is a graphical tool for measuring and recording performance of SQL Server and other Windows-related metrics using the Windows Management Interface (WMI) performance objects.

Note

Performance objects can also be accessed directly from T-SQL using the SQL Server Operating System Related DMVs. For a full list of the DMVs, see SQL Server Operating System Related Dynamic Management Views (Transact-SQL) in the *SQL Server documentation*.

Performance counters exist for real-time measurements such as CPU Utilization and for aggregated history such as average active transactions. For a full list of the object hierarchy, see: Use SQL Server Objects in the *SQL Server documentation*.

SQL Server Extended Events

SQL Server’s latest tracing framework provides very lightweight and robust event collection and storage. SQL Server Management Studio features the New Session Wizard and New Session graphic user interfaces for managing and analyzing captured data. SQL Server Extended Events consists of the following items:

- SQL Server Extended Events Package is a logical container for Extended Events objects.
- SQL Server Extended Events Targets are consumers of events. Targets include Event File, which writes data to the file Ring Buffer for retention in memory, or for processing aggregates such as Event Counters and Histograms.
- SQL Server Extended Events Engine is a collection of services and tools that comprise the framework.
- SQL Server Extended Events Sessions are logical containers mapped many-to-many with packages, events, and filters.

The following example creates a session that logs lock escalations and lock timeouts to a file.

```
CREATE EVENT SESSION Locking_Demo
ON SERVER
    ADD EVENT sqlserver.lock_escalation,
    ADD EVENT sqlserver.lock_timeout
    ADD TARGET package0.etw_classic_sync_target
        (SET default_etw_session_logfile_path = N'C:\ExtendedEvents\Locking
\Demo_20180502.etl')
    WITH (MAX_MEMORY=8MB, MAX_EVENT_SIZE=8MB);
GO
```

SQL Server Tracing Framework and the SQL Server Profiler Tool

The SQL Server trace framework is the predecessor to the Extended Events framework and remains popular among database administrators. The lighter and more flexible Extended Events Framework is recommended for development of new monitoring functionality. For more information, see SQL Server Profiler in the *SQL Server documentation*.

SQL Server Management Studio

SQL Server Management Studio (SSMS) provides several monitoring extensions:

- **SQL Server Activity Monitor** is an in-process, real-time, basic high-level information graphical tool.
- **Query Graphical Show Plan** provides easy exploration of estimated and actual query run plans.
- **Query Live Statistics** displays query run progress in real time.
- **Replication Monitor** presents a publisher-focused view or distributor-focused view of all replication activity. For more information, see Overview of the Replication Monitor Interface in the *SQL Server documentation*.
- **Log Shipping Monitor** displays the status of any log shipping activity whose status is available from the server instance to which you are connected. For more information, see View the Log Shipping Report (SQL Server Management Studio) in the *SQL Server documentation*.
- **Standard Performance Reports** is set of reports that show the most important performance metrics such as change history, memory usage, activity, transactions, HA, and more.

T-SQL

From the T-SQL interface, SQL Server provides many system stored procedures, system views, and functions for monitoring data.

System stored procedures such as `sp_who` and `sp_lock` provide real-time information. The `sp_monitor` procedure provides aggregated data.

Built in functions such as `@@CONNECTIONS`, `@@IO_BUSY`, `@@TOTAL_ERRORS`, and others provide high level server information.

A rich set of System Dynamic Management functions and views are provided for monitoring almost every aspect of the server. These functions reside in the sys schema and are prefixed with `dm_string`. For more information, see System Dynamic Management Views in the *SQL Server documentation*.

Trace Flags

You can set trace flags to log events. For example, set trace flag 1204 to log deadlock information. For more information, see DBCC TRACEON - Trace Flags (Transact-SQL) in the *SQL Server documentation*.

SQL Server Query Store

Query Store is a database-level framework supporting automatic collection of queries, run plans, and run time statistics. This data is stored in system tables. You can use this data to diagnose performance issues, understand patterns, and understand trends. It can also be set to automatically revert plans when a performance regression is detected.

For more information, see Monitoring performance by using the Query Store in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Relational Database Service (Amazon RDS) provides a rich monitoring infrastructure for Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) clusters and instances with the Amazon CloudWatch service. For more information, see Monitoring metrics in an Amazon RDS instance and Monitoring OS metrics with Enhanced Monitoring in the *Amazon Relational Database Service User Guide*.

You can also use the AWS Performance Insights tool to monitor PostgreSQL.

PostgreSQL can also be monitored by querying system catalog table and views.

Starting with PostgreSQL 12, you can monitor progress of `CREATE INDEX`, `REINDEX`, `CLUSTER`, and `VACUUM FULL` operations by querying system views `pg_stat_progress_create_index` and `pg_stat_progress_cluster`.

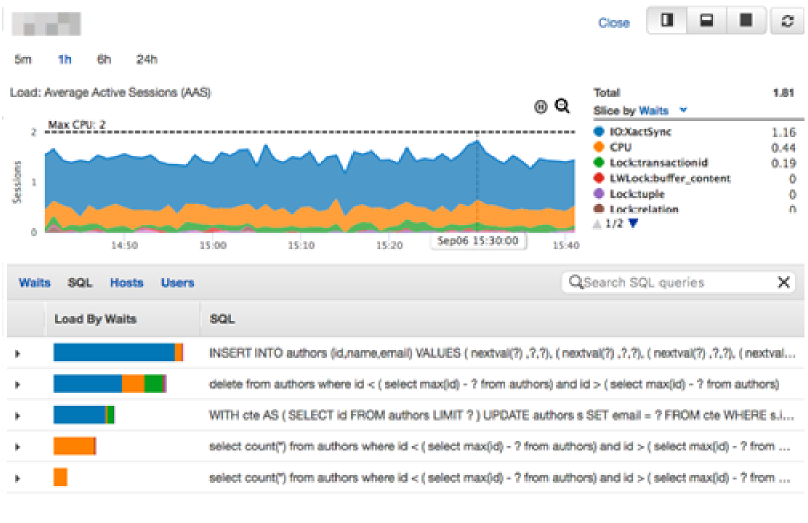
Starting with PostgreSQL 13, you can monitor progress of `ANALYZE` operations by querying system view `pg_stat_progress_analyze`. Also, you can monitor shared memory usage with system view `pg_shmem_allocations`.

Example

The following walkthrough demonstrates how to access the Amazon Aurora Performance Insights Console.

1. In the AWS console, choose **RDS**, and then choose **Performance insights**.
2. The web page displays a dashboard containing current and past database performance metrics. You can choose the period of the displayed performance data (5 minutes, 1 hour, 6 hours, or 24

hours) as well as different criteria to filter and slice the information such as waits, SQL, hosts, users, and so on.



Turning on Performance Insights

Performance insights are turned on by default for Amazon Aurora clusters. If you have more than one database in your Amazon Aurora cluster, performance data for all databases is aggregated. Database performance data is retained for 24 hours.

For more information, see Monitoring DB load with Performance Insights on Amazon RDS in the *Amazon Relational Database Service User Guide*.

SQL Server Resource Governor and PostgreSQL Dedicated Amazon Aurora Clusters or Aurora Read-Replicas

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Distribute load, applications, or users across multiple instances.

SQL Server Usage

SQL Server Resource Governor provides the capability to control and manage resource consumption. Administrators can specify and enforce workload limits on CPU, physical I/O, and Memory. Resource configurations are dynamic and you can change them in real time.

In SQL Server 2019 configurable value for the `REQUEST_MAX_MEMORY_GRANT_PERCENT` option of `CREATE WORKLOAD GROUP` and `ALTER WORKLOAD GROUP` has been changed from an integer to a float data type to allow more granular control of memory limits. For more information, see ALTER WORKLOAD GROUP (Transact-SQL) and CREATE WORKLOAD GROUP (Transact-SQL) in the *SQL Server documentation*.

Use Cases

The following list identifies typical Resource Governor use cases:

- **Minimize performance bottlenecks and inconsistencies** to better support Service Level Agreements (SLA) for multiple workloads and users.
- **Protect against runaway queries** that consume a large amount of resources or explicitly throttle I/O intensive operations. For example, consistency checks with DBCC that may bottleneck the I/O subsystem and negatively impact concurrent workloads.
- **Allow tracking and control for resource-based pricing scenarios** to improve predictability of user charges.

Concepts

The three basic concepts in Resource Governor are Resource Pools, Workload Groups, and Classification.

- **Resource Pools** represent physical resources. Two built-in resource pools, internal and default, are created when SQL Server is installed. You can create custom user-defined resource pools for specific workload types.
- **Workload Groups** are logical containers for session requests with similar characteristics. Workload Groups allow aggregate resource monitoring of multiple sessions. Resource limit policies are defined for a Workload Group. Each Workload Group belongs to a Resource Pool.

* **Classification** is a process that inspects incoming connections and assigns them to a specific Workload Group based on the common attributes. User-defined functions are used to implement Classification. For more information, see User-Defined Functions.

Examples

Enable the Resource Governor.

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

Create a Resource Pool.

```
CREATE RESOURCE POOL ReportingWorkloadPool
    WITH (MAX_CPU_PERCENT = 20);
```

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

Create a Workload Group.

```
CREATE WORKLOAD GROUP ReportingWorkloadGroup USING poolAdhoc;
```

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

Create a classifier function.

```
CREATE FUNCTION dbo.WorkloadClassifier()
RETURNS sysname WITH SCHEMABINDING
AS
BEGIN
    RETURN (CASE
        WHEN HOST_NAME()= 'ReportServer'
        THEN 'ReportingWorkloadGroup'
        ELSE 'Default'
    END)
END;
```

Register the classifier function.

```
ALTER RESOURCE GOVERNOR with (CLASSIFIER_FUNCTION = dbo.WorkloadClassifier);
```

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

For more information, see Resource Governor in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL doesn't have built-in resource management capabilities equivalent to the functionality provided by SQL Server's Resource Governor. However, due to the elasticity and flexibility provided by cloud economics, workarounds could be applicable and such capabilities might not be as of similar importance to monolithic on-premises databases.

The SQL Server's Resource Governor primarily exists because traditionally, SQL Server instances were installed on very powerful monolithic servers that powered multiple applications simultaneously. The monolithic model made the most sense in an environment where the licensing for the SQL Server database was per-CPU and where SQL Server instances were deployed on physical hardware. In these scenarios, it made sense to consolidate as many workloads as possible into fewer servers. With cloud databases, the strict requirement to maximize the usage of each individual server is often not as important and you can use a different approach.

You can deploy individual Amazon Aurora clusters with varying sizes, each dedicated to a specific application or workload. You can use additional read-only Amazon Aurora Replica servers to offload any reporting workloads from the master instance.

With Amazon Aurora, you can deploy separate and dedicated database clusters, each dedicated to a specific application or workload creating isolation between multiple connected sessions and applications.

Each Amazon Aurora instance (primary or replica) can scale independently in terms of CPU and memory resources using different instance types. Because you can instantly deploy multiple Amazon Aurora Instances and much less overhead is associated with the deployment and management of Amazon Aurora instances when compared to physical servers, separating different workloads to different instance classes could be a suitable solution for controlling resource management.

For more information, see Amazon EC2 Instance Types.

In addition, each Amazon Aurora instance can also be directly accessed from your applications using its own endpoint. This capability is especially useful if you have multiple Amazon Aurora read-replicas for a given cluster and you want to use different Amazon Aurora replicas to segment your workload.

You can adjust the resources and some parameters for Amazon Aurora read-replicas in the same cluster to avoid having additional cluster, however, this will allow to be used only for read operations.

Examples

Follow these steps to create an Amazon Aurora cluster.

1. In the AWS console, choose **RDS**.
2. Choose **Databases**, and then choose **Create database**.
3. Follow the wizard. Your new cluster appears in the **Databases** section.

Suppose that you were using a single SQL Server instance for multiple separate applications and used SQL Server Resource Governor to enforce a workload separation, allocating a specific amount of server resources for each application. With Amazon Aurora, you might want to create multiple separate databases for each individual application.

Follow these steps to add additional replica instances to an existing Amazon Aurora cluster:

1. In the AWS console, choose **RDS**.
2. Choose the Amazon Aurora cluster that you want to scale-out by adding an additional read replica.
3. For **Instance actions**, choose **Create Aurora Replica**.
4. Select the instance class depending on the amount of compute resources your application requires.
5. Choose **Create Aurora Replica**.

Dedicated Aurora PostgreSQL Instances

<table>
<tr><th>Feature</th><th>Amazon Aurora instances</th></tr>
<tr><td>Set the maximum CPU usage for a resource group.</td><td>Create a dedicated Amazon Aurora instance for a specific application.</td></tr>
<tr><td>Limit the degree of parallelism for specific queries.</td><td>

```
SET max_parallel_workers_per_gather
 TO x;
```

Setting the PostgreSQL max_parallel_workers_per_gather parameter should be done as part of your application database connection.</td></tr>
<tr><td>Limit parallel runs</td><td>

```
SET max_parallel_workers_per_gather
 TO 0;
```

or

```
SET max_parallel_workers TO x; -- for
 the whole system (since PostgreSQL
 10)
```

</td></tr>
<tr><td>Limit the number of active sessions.</td><td>Manually detect the number of connectio ns that are open from a specific application and restrict connectivity either with database procedures or within the application DAL itself.

```
select pid from pg_stat_activity
 where usename in( select usename from
 pg_stat_activity
    where state = 'active' group by
 usename having count(*) > 10)
    and state = 'active' order by
 query_Start;
```

</td></tr>
</table>

<table>
<tr><th>Feature</th><th>Amazon Aurora instances</th></tr>
<tr><td>Restrict maximum runtime of queries.</td><td>Manually terminate sessions that exceed the required threshold. You can detect the length of running queries using SQL commands and restrict max run duration using either database procedures or within the application DAL itself.
<pre>SELECT pg_terminate_backend(pid)
    FROM pg_stat_activity
    WHERE now()-pg_stat_activity.quer
y_start > interval '5 minutes';</pre></td></tr>
<tr><td>Limit the maximum idle time for sessions.</td><td>Manually terminate sessions that exceed the required threshold. You can detect the length of your idle sessions using SQL queries and restrict maximum run using either database procedures or within the application DAL itself.
<pre>SELECT pg_terminate_backend(pid)
    FROM pg_stat_activity
    WHERE datname = 'regress' AND pid
 <> pg_backend_pid()
    AND state = 'idle' AND state_cha
nge < current_timestamp - INTERVAL
 '5' MINUTE;</pre></td></tr>
</table>

Feature	Amazon Aurora instances
Limit the time that an idle session holding open locks can block other sessions.	Manually terminate sessions that exceed the required threshold. You can detect the length of blocking idle sessions using SQL queries and restrict max run duration using either database procedures or within the application DAL itself. `SELECT pg_terminate_backend(blocking_locks.pid) FROM pg_catalog.pg_locks AS blocked_locks JOIN pg_catalog.pg_stat_activity AS blocked_activity ON blocked_activity.pid = blocked_locks.pid JOIN pg_catalog.pg_locks AS blocking_locks ON blocking_locks.locktype = blocked_locks.locktype AND blocking_locks.DATABASE IS NOT DISTINCT FROM blocked_locks.DATABASE AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple AND blocking_locks.virtualxid IS NOT DISTINCT FROM blocked_locks.virtualxid AND blocking_locks.transactionid IS NOT DISTINCT FROM blocked_locks.transactionid AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid`

<table>
<tr><th>Feature</th><th>Amazon Aurora instances</th></tr>
<tr><td></td><td><pre>   AND blocking_locks.pid !=
blocked_locks.pid
   JOIN pg_catalog.pg_stat_activity
AS blocking_activity
   ON blocking_activity.pid =
blocking_locks.pid
   WHERE NOT blocked_locks.granted
and blocked_activity.state_change
< current_timestamp - INTERVAL '5'
minute;</pre></td></tr>
</table>

For more information, see Resource Consumption in the *PostgreSQL documentation*.

SQL Server Linked Servers and PostgreSQL DBLink and FDWrapper

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	Linked Servers	Syntax and option differences, similar functionality.

SQL Server Usage

Linked servers enable the database engine to connect to external Object Linking and Embedding for databases (OLE-DB) sources. They are typically used to run T-SQL commands and include tables in other instances of SQL Server, or other RDBMS engines such as Oracle. SQL Server supports multiple types of OLE-DB sources as linked servers, including Microsoft Access, Microsoft Excel, text files and others.

The main benefits of using linked servers are:

- Reading external data for import or processing.

- Running distributed queries, data modifications, and transactions for enterprise-wide data sources.
- Querying heterogeneous data source using the familiar T-SQL API.

You can configure linked servers using either SQL Server Management Studio, or the system stored procedure `sp_addlinkedserver`. The available functionality and the specific requirements vary significantly between the various OLE-DB sources. Some sources may allow read only access, others may require specific security context settings, and so on.

The linked server definition contains the linked server alias, the OLE DB provider, and all the parameters needed to connect to a specific OLE-DB data source.

The OLE-DB provider is a .NET Dynamic Link Library (DLL) that handles the interaction of SQL Server with all data sources of its type. For example, OLE-DB Provider for Oracle. The OLE-DB data source is the specific data source to be accessed, using the specified OLE-DB provider.

Note

You can use SQL Server distributed queries with any custom OLE DB provider as long as the required interfaces are implemented correctly.

SQL Server parses the T-SQL commands that access the linked server and sends the appropriate requests to the OLE-DB provider. There are several access methods for remote data, including opening the base table for read or issuing SQL queries against the remote data source.

You can manage linked servers using SQL Server Management Studio graphical user interface or T-SQL system stored procedures.

- `EXECUTE sp_addlinkedserver` to add new server definitions.
- `EXECUTE sp_addlinkedserverlogin` to define security context.
- `EXECUTE sp_linkedservers` or `SELECT * FROM sys.servers` system catalog view to retrieve meta data.
- `EXECUTE sp_dropserver` to delete a linked server.

You can access linked server data sources from T-SQL using a fully qualified, four-part naming scheme: `<Server Name>.<Database Name>.<Schema Name>.<Object Name>`.

Additionally, you can use the OPENQUERY row set function to explicitly invoke pass-through queries on the remote linked server. Also, you can use the OPENROWSET and OPENDATASOURCE row set functions for one-time remote data access without defining the linked server in advance.

Syntax

```
EXECUTE sp_addlinkedserver
    [ @server= ] <Linked Server Name>
    [ , [ @srvproduct= ] <Product Name>]
    [ , [ @provider= ] <OLE DB Provider>]
    [ , [ @datasrc= ] <Data Source>]
    [ , [ @location= ] <Data Source Address>]
    [ , [ @provstr= ] <Provider Connection String>]
    [ , [ @catalog= ] <Database>];
```

Examples

Create a linked server to a local text file.

```
EXECUTE sp_addlinkedserver MyTextLinkedServer, N'Jet 4.0',
    N'Microsoft.Jet.OLEDB.4.0',
    N'D:\TextFiles\MyFolder',
    NULL,
    N'Text';
```

Define security context.

```
EXECUTE sp_addlinkedsrvlogin MyTextLinkedServer, FALSE, Admin, NULL;
```

Use sp_tables_ex to list tables in a folder.

```
EXEC sp_tables_ex MyTextLinkedServer;
```

Issue a SELECT query using a four-part name.

```
SELECT *
FROM MyTextLinkedServer...[FileName#text];
```

For more information, see sp_addlinkedserver (Transact-SQL) and Distributed Queries Stored Procedures (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Querying data in remote databases is available through two primary options:

- `dblink` database link function.
- Foreign data wrapper (FDW) `postgresql_fdw` extension.

The PostgreSQL foreign data wrapper extension is new to PostgreSQL and provides functionality similar to `dblink`. However, the PostgreSQL foreign data wrapper aligns closer with the SQL standard and can provide improved performance.

Examples

Load the `dblink` extension into PostgreSQL.

```
CREATE EXTENSION dblink;
```

Create a persistent connection to a remote PostgreSQL database using the `dblink_connect` function specifying a connection name (`myconn`), database name (`postgresql`), port (`5432`), host (`hostname`), user (`username`), and password (`password`).

```
SELECT dblink_connect ('myconn',
    'dbname=postgres port=5432 host=hostname user=username password=password');
```

You can use the connection to run queries against the remote database.

Run a query using the previously created `myconn` connection by using the `dblink` function. The query returns the id and name columns from the employees table. On the remote database, you must specify the connection name and the SQL query to run as well as parameters and datatypes for selected columns (id and name in this example).

```
SELECT * from dblink ('myconn',
    'SELECT id, name FROM EMPLOYEES') AS p(id int,fullname text);
```

Close the connection using the `dblink_disconnect` function.

```
SELECT dblink_disconnect('myconn');
```

Alternatively, you can use the `dblink` function specifying the full connection string to the remote PostgreSQL database including the database name, port, hostname, username, and password. You can do this instead of using a previously defined connection. Make sure that you specify the SQL query to run as well as parameters and data types for the selected columns (id and name, in this example).

```
SELECT * from dblink ('dbname=postgres port=5432 host=hostname user=username
 password=password',
    'SELECT id, name FROM EMPLOYEES') AS p(id int,fullname text);
```

DML commands are supported on tables referenced by the `dblink` function. For example, you can insert a new row and then delete it from the remote table.

```
SELECT * FROM dblink('myconn',$$INSERT into employees VALUES (3,'New Employees No.
 3!')$$) AS t(message text);

SELECT * FROM dblink('myconn',$$DELETE FROM employees WHERE id=3$$) AS t(message text);
```

Create a new `new_employees_table` local table by querying data from a remote table.

```
SELECT emps.* INTO new_employees_table
    FROM dblink('myconn','SELECT * FROM employees')
    AS emps(id int, name varchar);
```

Join remote data with local data.

```
SELECT local_emps.id , local_emps.name, s.sale_year, s.sale_amount
    FROM local_emps INNER JOIN
    dblink('myconn','SELECT * FROM working_hours') AS s(id int, hours worked int)
    ON local_emps.id = s.id;
```

Run DDL statements in the remote database.

```
SELECT * FROM dblink('myconn',$$CREATE table new_remote_tbl (a int, b text)$$) AS t(a
 text);
```

For more information, see dblink in the *PostgreSQL documentation*.

Scripting

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Non-compatible tool sets and scripting languages . Use PostgreSQL `pgAdmin`, Amazon RDS API, AWS Management Console, and AWS CLI.

SQL Server Usage

SQL Server supports T-SQL and XQuery scripting within multiple run frameworks such as SQL Server Agent, and stored procedures.

The `SQLCMD` command line utility can also be used to run T-SQL scripts. However, the most extensive and feature-rich scripting environment is PowerShell.

SQL Server provides two PowerShell snap-ins that implement a provider exposing the entire SQL Server Management Object Model (SMO) as PowerShell paths. Additionally, you can use `cmd` in SQL Server to run specific SQL Server commands.

> **Note**
>
> You can use `Invoke-Sqlcmd` to run scripts using the SQLCMD utility.

The `sqlps` utility launches the PowerShell scripting environment and automatically loads the SQL Server modules. You can launch `sqlps` from a command prompt or from the Object Explorer pane of SQL Server Management Studio. You can run one-time PowerShell commands and script files (for example, `.\SomeFolder\SomeScript.ps1`).

Note

SQL Server Agent supports running PowerShell scripts in job steps. For more information, see SQL Server Agent.

SQL Server also supports three types of direct database engine queries: T-SQL, XQuery, and the SQLCMD utility. You can call T-SQL and XQuery from stored procedures, SQL Server Management Studio (or other IDE), and SQL Server agent jobs. The SQLCMD utility also supports commands and variables.

Examples

Backup a database with PowerShell using the default backup options.

```
PS C:\> Backup-SqlDatabase -ServerInstance "MyServer\SQLServerInstance" -Database
 "MyDB"
```

Get all rows from the `MyTable` table in the `MyDB` database.

```
PS C:\> Read-SqlTableData -ServerInstance MyServer\SQLServerInstance" -DatabaseName
 "MyDB" -TableName "MyTable"
```

For more information, see SQL Server PowerShell, Database Engine Scripting, and sqlcmd Utility in the *SQL Server documentation*.

PostgreSQL Usage

As a Platform as a Service (PaaS), Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) accepts connections from any compatible client, but you can't access the PostgreSQL command line utility typically used for database administration. However, you can use PostgreSQL tools installed on a network host and the Amazon Relational Database Service (Amazon RDS) API. The most common tools for Aurora PostgreSQL scripting and automation include PostgreSQL `pgAdmin`, PostgreSQL utilities, and the Amazon RDS API. The following sections describe each tool.

PostgreSQL pgAdmin

PostgreSQL `pgAdmin` is the most commonly used tool for development and administration of PostgreSQL servers. It is available as a free Community Edition and paid support is available.

The PostgreSQL `pgAdmin` also supports a Python scripting shell that you can use interactively and programmatically. For more information see: pgAdmin.

Amazon RDS API

The Amazon RDS API is a web service for managing and maintaining Aurora PostgreSQL and other relational databases. You can use Amazon RDS API to setup, operate, scale, backup, and perform many common administration tasks. The Amazon RDS API supports multiple database platforms and can integrate administration seamlessly for heterogeneous environments.

> **Note**
>
> The Amazon RDS API is asynchronous. Some interfaces may require polling or callback functions to receive command status and results.

You can access Amazon RDS using the AWS Management Console, the AWS Command Line Interface (CLI), and the Amazon RDS Programmatic API as described in the following sections.

AWS Management Console

The AWS Management Console is a simple web-based set of tools for interactive management of Aurora PostgreSQL and other Amazon RDS services. To access the AWS Management Console, sign in to your AWS account, and choose **RDS**.

AWS Command Line Interface

The AWS Command Line Interface is an open source tool that runs on Linux, Windows, or macOS having Python 2 version 2.6.5 and higher or Python 3 version 3.3 and higher.

The AWS CLI is built on top of the AWS SDK for Python (Boto), which provides commands for interacting with AWS services. With minimal configuration, you can start using all AWS Management Console functionality from your favorite terminal application.

- **Linux shells** — Use common shell programs such as Bash, Zsh, or tsch.
- **Windows command line** — Run commands in PowerShell or the Windows Command Processor.
- **Remotely** — Run commands on Amazon EC2 instances through a remote terminal such as PuTTY or SSH.

The AWS Tools for Windows PowerShell and AWS Tools for PowerShell Core are PowerShell modules built on the functionality exposed by the AWS SDK for .NET. These Tools enable scripting operations for AWS resources using the PowerShell command line.

> **Note**
>
> You can't use SQL Server cmdlets in PowerShell.

Amazon RDS Programmatic API

You can use the Amazon RDS API to automate management of database instances and other Amazon RDS objects.

For more information, see Actions, Data Types, Common Parameters, and Common Errors in the *Amazon Relational Database Service API Reference*.

Examples

The following walkthrough describes how to connect to an Aurora PostgreSQL database instance using the PostgreSQL utility.

1. Sign in to your AWS account, choose **RDS**, and then choose **Databases**.
2. Choose the PostgreSQL database you want to connect to and copy the cluster endpoint address.

   > **Note**
   >
   > You can also connect to individual database instances. For more information, see High Availability Essentials.

3. In the command shell, enter the following:

```
psql --host=mypostgresql.c6c8mwvfdgv0.us-west-2.rds.amazonaws.com
    --port=5432 --username=awsuser --password --dbname=mypgdb
```

In the preceding example, the `--host` parameter is the endpoint DNS name of the Aurora PostgreSQL database cluster.

In the preceding example, the `--port` parameter is the port number.

For more information, see Command Line Interface Command Reference and Amazon Relational Database Service API Reference.

Performance Tuning

Topics

- Run Plans
- SQL Server Query Hints and Plan Guides and PostgreSQL DB Query Planning
- SQL Server Managing Statistics and PostgreSQL Table Statistics

Run Plans

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Syntax differences. Completely different optimizer with different operators and rules.

SQL Server Usage

Run plans provide users detailed information about the data access and processing methods chosen by the SQL Server Query Optimizer. They also provide estimated or actual costs of each operator and sub-tree. Run plans provide critical data for troubleshooting query performance issues.

SQL Server creates run plans for most queries and returns them to client applications as plain text or XML documents. SQL Server produces an run plan when a query runs, but it can also generate estimated plans without running a query.

SQL Server Management Studio provides a graphical view of the underlying XML plan document using icons and arrows instead of textual information. This graphical view is extremely helpful when investigating the performance aspects of a query.

To request an estimated run plan, use the `SET SHOWPLAN_XML`, `SHOWPLAN_ALL`, or `SHOWPLAN_TEXT` statements.

SQL Server 2017 introduces automatic tuning, which notifies users whenever a potential performance issue is detected and lets them apply corrective actions, or lets the Database Engine automatically fix performance problems.

Automatic tuning SQL Server enables users to identify and fix performance issues caused by query run plan choice regressions. For more information, see Automatic tuning in the *SQL Server documentation*.

Examples

Show the estimated run plan for a query.

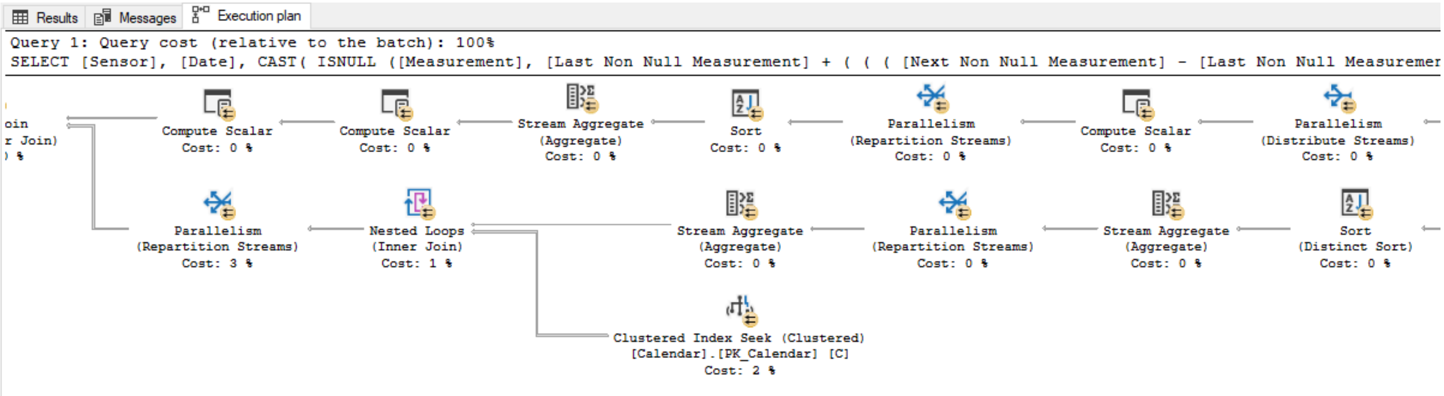
```
SET SHOWPLAN_XML ON;
SELECT *
FROM MyTable
WHERE SomeColumn = 3;
SET SHOWPLAN_XML OFF;
```

Actual run plans return after run of the query or batch of queries completes. Actual run plans include run-time statistics about resource usage and warnings. To request the actual run plan, use the `SET STATISTICS XML` statement to return the XML document object. Alternatively, use the `STATISTICS PROFILE` statement, which returns an additional result set containing the query run plan.

Show the actual run plan for a query.

```
SET STATISTICS XML ON;
SELECT *
FROM MyTable
WHERE SomeColumn = 3;
SET STATISTICS XML OFF;
```

The following example shows a partial graphical run plan from SQL Server Management Studio.



For more information, see Display and Save Execution Plans in the *SQL Server documentation*.

PostgreSQL Usage

When using the EXPLAIN command, PostgreSQL will generate the estimated run plan for actions, such as SELECT, INSERT, UPDATE, and DELETE. EXPLAIN builds a structured tree of plan nodes representing the different actions taken (the sign # represents a root line in the PostgreSQL run plan). In addition, the EXPLAIN statement will provide statistical information regarding each action, such as cost, rows, time and loops.

When using the EXPLAIN command as part of a SQL statement, the statement will not run, and the run plan will be an estimation. By using the EXPLAIN ANALYZE command, the statement will run in addition to displaying the run plan.

PostgreSQL EXPLAIN Synopsis

```
EXPLAIN [ ( option value[, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

where option and values can be one of:
  ANALYZE [ boolean ]
  VERBOSE [ boolean ]
  COSTS [ boolean ]
  BUFFERS [ boolean ]
  TIMING [ boolean ]
  SUMMARY [ boolean ] (since PostgreSQL 10)
  FORMAT { TEXT | XML | JSON | YAML }
```

By default, planning and run time are displayed when using EXPLAIN ANALYZE, but not in other cases. A new option SUMMARY gives explicit control of this information. Use SUMMARY to include planning and run time metrics in your output.

PostgreSQL provides configurations options that will cancel SQL statements running longer than provided time limit. To use this option, you can set the `statement_timeout` instance-level parameter. If the value is specified without units, it is taken as milliseconds. A value of zero (the default) disables the timeout.

Third-party connection pooler solutions like `Pgbouncer` and `PgPool` build on that and allow more flexibility in controlling how long connection to DB can run, be in idle state, and so on.

Aurora PostgreSQL Query Plan Management

The Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) Query Plan Management (QPM) feature solves the problem of plan instability by allowing database users to maintain stable, yet optimal, performance for a set of managed SQL statements. QPM primarily serves two main objectives:

- **Plan stability**. QPM prevents plan regression and improves plan stability when any of the preceding changes occur in the system.
- **Plan adaptability**. QPM automatically detects new minimum-cost plans and controls when new plans may be used and adapts to the changes.

The quality and consistency of query optimization have a major impact on the performance and stability of any relational database management system (RDBMS). Query optimizers create a query run plan for a SQL statement at a specific point in time. As conditions change, the optimizer might pick a different plan that makes performance better or worse. In some cases, a number of changes can all cause the query optimizer to choose a different plan and lead to performance regression. These changes include changes in statistics, constraints, environment settings, query parameter bindings, and software upgrades. Regression is a major concern for high-performance applications.

With query plan management, you can control run plans for a set of statements that you want to manage. You can do the following:

- Improve plan stability by forcing the optimizer to choose from a small number of known, good plans.
- Optimize plans centrally and then distribute the best plans globally.

- Identify indexes that aren't used and assess the impact of creating or dropping an index.
- Automatically detect a new minimum-cost plan discovered by the optimizer.
- Try new optimizer features with less risk, because you can choose to approve only the plan changes that improve performance.

Examples

Display the run plan of a SQL statement using the EXPLAIN command.

```
EXPLAIN
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME FROM EMPLOYEES
WHERE LAST_NAME='King' AND FIRST_NAME='Steven';

Index Scan using idx_emp_name on employees (cost=0.14..8.16 rows=1 width=18)
Index Cond: (((last_name)::text = 'King'::text) AND ((first_name)::text =
 'Steven'::text))
(2 rows)
```

Run the same statement with the ANALYZE keyword.

```
EXPLAIN ANALYZE
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME FROM EMPLOYEES
WHERE LAST_NAME='King' AND FIRST_NAME='Steven';


Seq Scan on employees (cost=0.00..3.60 rows=1 width=18) (actual time=0.012..0.024
 rows=1 loops=1)
Filter: (((last_name)::text = 'King'::text) AND ((first_name)::text = 'Steven'::text))
Rows Removed by Filter: 106
Planning time: 0.073 ms
Execution time: 0.037 ms
(5 rows)
```

By adding the ANALYZE keyword and running the statement, we get additional information in addition to the run plan.

View a PostgreSQL run plan showing a FULL TABLE SCAN.

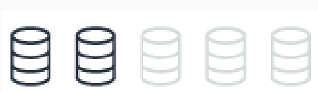
```
EXPLAIN ANALYZE
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME FROM EMPLOYEES
```

```
WHERE SALARY > 10000;

Seq Scan on employees (cost=0.00..3.34 rows=15 width=18) (actual time=0.012..0.036
 rows=15 loops=1)
Filter: (salary > '10000'::numeric)
Rows Removed by Filter: 92
Planning time: 0.069 ms
Execution time: 0.052 ms
(5 rows)
```

PostgreSQL can perform several scan types for processing and retrieving data from tables including sequential scans, index scans, and bitmap index scans. The sequential scan is PostgreSQL equivalent for SQL Server full table scan.

For more information, see EXPLAIN in the *PostgreSQL documentation*.

SQL Server Query Hints and Plan Guides and PostgreSQL DB Query Planning

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Very limited set of hints - Index hints and optimizer hints as comments. Syntax differences.

SQL Server Usage

SQL Server hints are instructions that override automatic choices made by the query processor for DML and DQL statements. The term hint is misleading because, in reality, it forces an override to any other choice of run plan.

JOIN Hints

You can explicitly add `LOOP`, `HASH`, `MERGE`, and `REMOTE` hints to a `JOIN` statement. For example, … `Table1 INNER LOOP JOIN Table2 ON … .`

These hints force the optimizer to use nested loops, hash match, or merge physical join algorithms.

`REMOTE` enables processing a join with a remote table on the local server.

Table Hints

Table hints override the default behavior of the query optimizer. Table hints are used to explicitly force a particular locking strategy or access method for a table operation clause. These hints don't modify the defaults and apply only for the duration of the DML or DQL statement.

Some common table hints are `INDEX = <Index value>`, `FORCESEEK`, `NOLOCK`, and `TABLOCKX`.

Query Hints

Query hints affect the entire set of query operators, not just the individual clause in which they appear. Query hints may be `JOIN` hints, table hints, or from a set of hints that are only relevant for query hints.

Some common table hints include `OPTIMIZE FOR`, `RECOMPILE`, `FORCE ORDER`, `FAST <rows>`.

You can specify query hints after the query itself following the `WITH` options clause.

Plan Guides

Plan guides provide similar functionality to query hints in the sense they allow explicit user intervention and control over query optimizer plan choices. Plan guides can use either query hints or a full fixed, pre-generated plan attached to a query. The difference between query hints and plan guides is the way they are associated with a query.

While query or table hints need to be explicitly stated in the query text, they aren't an option if you have no control over the source code generating these queries. If an application uses ad-hoc queries instead of stored procedures, views, and functions, the only way to affect query plans is to use plan guides. They are often used to mitigate performance issues with third-party software.

A plan guide consists of the statement whose run plan needs to be adjusted and either an `OPTION` clause that lists the desired query hints or a full XML query plan that is enforced as long it is valid.

At run time, SQL Server matches the text of the query specified by the guide and attaches the OPTION hints. Alternatively, it assigns the provided plan for running.

SQL Server supports three types of plan guides:

- **Object plan guides** target statements that run within the scope of a code object such as a stored procedure, function, or trigger. If the same statement is found in another context, the plan guide is not be applied.
- **SQL plan guides** are used for matching general ad-hoc statements not within the scope of code objects. In this case, any instance of the statement regardless of the originating client is assigned the plan guide.
- **Template plan guides** can be used to abstract statement templates that differ only in parameter values. You can use them to override the PARAMETERIZATION database option setting for a family of queries.

Syntax

The following example uses query hints in a SELECT statement. You can use query hints in all DQL and DML statements.

```
SELECT <statement>
OPTION
(
{{HASH|ORDER} GROUP
|{CONCAT |HASH|MERGE} UNION
|{LOOP|MERGE|HASH} JOIN
|EXPAND VIEWS
|FAST <Rows>
|FORCE ORDER
|{FORCE|DISABLE} EXTERNALPUSHDOWN
|IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX
|KEEP PLAN
|KEEPFIXED PLAN
|MAX_GRANT_PERCENT = <Percent>
|MIN_GRANT_PERCENT = <Percent>
|MAXDOP <Number of Processors>
|MAXRECURSION <Number>
|NO_PERFORMANCE_SPOOL
|OPTIMIZE FOR (@<Variable> {UNKNOWN|= <Value>}[,...])
|OPTIMIZE FOR UNKNOWN
|PARAMETERIZATION {SIMPLE|FORCED}
|RECOMPILE
|ROBUST PLAN
|USE HINT ('<Hint>' [,...])
|USE PLAN N'<XML Plan>'
```

```
|TABLE HINT (<Object Name> [,<Table Hint>[[,...]])
});
```

The following example creates a plan guide.

```
EXECUTE sp_create_plan_guide @name = '<Plan Guide Name>'
  ,@stmt = '<Statement>'
  ,@type = '<OBJECT|SQL|TEMPLATE>'
  ,@module_or_batch = 'Object Name>'|'<Batch Text>'| NULL
  ,@params = '<Parameter List>'|NULL }
  ,@hints = 'OPTION(<Query Hints>'|'<XML Plan>'|NULL;
```

Examples

Limit parallelism for a sales report query.

```
EXEC sp_create_plan_guide
  @name = N'SalesReportPlanGuideMAXDOP',
  @stmt = N'SELECT *
    FROM dbo.fn_SalesReport(GETDATE())
  @type = N'SQL',
  @module_or_batch = NULL,
  @params = NULL,
  @hints = N'OPTION (MAXDOP 1)';
```

Use table and query hints.

```
SELECT *
FROM MyTable1 AS T1
  WITH (FORCESCAN)
  INNER LOOP JOIN
  MyTable2 AS T2
  WITH (TABLOCK, HOLDLOCK)
  ON T1.Col1 = T2.Col1
WHERE T1.Date BETWEEN DATEADD(DAY, -7, GETDATE()) AND GETDATE()
```

For more information, see Hints (Transact-SQL) and Plan Guides in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL doesn't support database hints to influence the behavior of the query planner, and you can't influence how run plans are generated from within SQL queries. Although database hints aren't directly supported, session parameters (also known as Query Planning Parameters) can influence the behavior of the query optimizer at the session level.

Examples

Configure the query planner to use indexes instead of full table scans (disable SEQSCAN).

```
SET ENABLE_SEQSCAN=FALSE;
```

Set the query planner's estimated cost of a disk page fetch that is part of a series of sequential fetches (`SEQ_PAGE_COST`) and set the planner's estimate of the cost of a non-sequentially-fetched disk page (`RANDOM_PAGE_COST`). Reducing the value of `RANDOM_PAGE_COST` relative to `SEQ_PAGE_COST` causes the query planner to prefer index scans, while raising the value makes index scans more expensive.

```
SET SEQ_PAGE_COST to 4;
SET RANDOM_PAGE_COST to 1;
```

Turn on or turn off the query planner's use of nested-loops when performing joins. While it is impossible to completely disable the usage of nested-loop joins, setting the ENABLE_NESTLOOP to OFF discourages the query planner from choosing nested-loop joins compared to alternative join methods.

```
SET ENABLE_NESTLOOP to FALSE;
```

For more information, see Query Planning in the *PostgreSQL documentation*.

SQL Server Managing Statistics and PostgreSQL Table Statistics

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Syntax and option differences, similar functionality.

SQL Server Usage

Statistics objects in SQL Server are designed to support SQL Server cost-based query optimizer. It uses statistics to evaluate the various plan options and choose an optimal plan for optimal query performance.

Statistics are stored as BLOBs in system tables and contain histograms and other statistical information about the distribution of values in one or more columns. A histogram is created for the first column only and samples the occurrence frequency of distinct values. Statistics and histograms are collected by either scanning the entire table or by sampling only a percentage of the rows.

You can view Statistics manually using the `DBCC SHOW_STATISTICS` statement or the more recent `sys.dm_db_stats_properties` and `sys.dm_db_stats_histogram` system views.

SQL Server provides the capability to create filtered statistics containing a WHERE predicate. Filtered statistics are useful for optimizing histogram granularity by eliminating rows whose values are of less interest, for example NULLs.

SQL Server can manage the collection and refresh of statistics automatically (the default). Use the `AUTO_CREATE_STATISTICS` and `AUTO_UPDATE_STATISTICS` database options to change the defaults.

When a query is submitted with `AUTO_CREATE_STATISTICS` on and the query optimizer may benefit from a statistics that don't yet exist, SQL Server creates the statistics automatically. You can use the `AUTO_UPDATE_STATISTICS_ASYNC` database property to set new statistics creation to occur immediately (causing queries to wait) or to run asynchronously. When run asynchronously, the triggering run can't benefit from optimizations the optimizer may derive from it.

After creation of a new statistics object, either automatically or explicitly using the CREATE STATISTICS statement, the refresh of the statistics is controlled by the AUTO_UPDATE_STATISTICS database option. When set to ON, statistics are recalculated when they are stale, which happens when significant data modifications have occurred since the last refresh.

Syntax

```
CREATE STATISTICS <Statistics Name>
ON <Table Name> (<Column> [,...])
[WHERE <Filter Predicate>]
[WITH <Statistics Options>;
```

Examples

The following example creates new statistics on multiple columns. Set to use a full scan and to not refresh.

```
CREATE STATISTICS MyStatistics
ON MyTable (Col1, Col2)
WITH FULLSCAN, NORECOMPUTE;
```

The following example updates statistics with a 50% sampling rate.

```
UPDATE STATISTICS MyTable(MyStatistics)
WITH SAMPLE 50 PERCENT;
```

View the statistics histogram and data.

```
DBCC SHOW_STATISTICS ('MyTable','MyStatistics');
```

Turn off automatic statistics creation for a database.

```
ALTER DATABASE MyDB SET AUTO_CREATE_STATS OFF;
```

For more information, see Statistics, CREATE STATISTICS (Transact-SQL), and DBCC SHOW_STATISTICS (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Use the ANALYZE command to collect statistics about a database, a table, or a specific table column. The PostgreSQL ANALYZE command collects table statistics that support the generation of efficient query run plans by the query planner.

- **Histograms** — ANALYZE collects statistics on table column values and creates a histogram of the approximate data distribution in each column.
- **Pages and Rows** — ANALYZE collects statistics on the number of database pages and rows from which each table is comprised.
- **Data Sampling** — For large tables, the ANALYZE command takes random samples of values rather than examining each row. This allows the ANALYZE command to scan very large tables in a relatively small amount of time.
- **Statistic Collection Granularity** — Running the ANALYZE command without parameters instructs PostgreSQL to examine every table in the current schema. Supplying the table name or column name to ANALYZE instructs the database to examine a specific table or table column.

Automatic Statistics Collection

By default, PostgreSQL is configured with an AUTOVACUUM daemon which automates the run of statistics collection by using the ANALYZE commands (in addition to automation of the VACUUM command). The AUTOVACUUM daemon scans for tables that show signs of large modifications in data to collect the current statistics. AUTOVACUUM is controlled by several parameters.

Individual tables have several storage parameters which can trigger AUTOVACUUM process sooner or later. You can set or change such parameters as autovacuum_enabled, autovacuum_vacuum_threshold, and others, using CREATE TABLE or ALTER TABLE statements.

```
ALTER TABLE custom_autovaccum SET (autovacuum_enabled = true,
 autovacuum_vacuum_cost_delay = 10ms, autovacuum_vacuum_scale_factor = 0.01,
 autovacuum_analyze_scale_factor = 0.005);
```

The preceding command enables AUTOVACUUM for the custom_autovaccum table and specifies the AUTOVACUUM process to sleep for 10 milliseconds each run.

It also specifies a 1% of the table size to be added to `autovacuum_vacuum_threshold` and 0.5% of the table size to be added to `autovacuum_analyze_threshold` when deciding whether to trigger a `VACUUM`.

For more information, see [Automatic Vacuuming] in the *PostgreSQL documentation*.

Manual Statistics Collection

In PostgreSQL, you can collect statistics on-demand using the `ANALYZE` command at the database level, table level, or column level.

- `ANALYZE` on indexes isn't currently supported.
- `ANALYZE` requires only a read-lock on the target table. It can run in parallel with other activity on the table.
- For large tables, `ANALYZE` takes a random sample of the table contents. It is configured by the show `default_statistics_target` parameter. The default value is 100 entries. Raising the limit might allow more accurate planner estimates to be made at the price of consuming more space in the `pg_statistic` table.

Starting from PostgreSQL 10, there is a new command `CREATE STATISTICS`, which creates a new extended statistics object tracking data about the specified table.

The `STATISTICS` object tells the server to collect more detailed statistics.

Examples

The following example gathers statistics for the entire database.

```
ANALYZE;
```

The following example gathers statistics for a specific table. The `VERBOSE` keyword displays progress.

```
ANALYZE VERBOSE EMPLOYEES;
```

The following example gathers statistics for a specific column.

```
ANALYZE EMPLOYEES (HIRE_DATE);
```

Specify the default_statistics_target parameter for an individual table column and reset it back to default.

```
ALTER TABLE EMPLOYEES ALTER COLUMN SALARY SET STATISTICS 150;

ALTER TABLE EMPLOYEES ALTER COLUMN SALARY SET STATISTICS -1;
```

Larger values increase the time needed to complete an ANALYZE, but improve the quality of the collected planner's statistics, which can potentially lead to better run plans.

View the current (session or global) `default_statistics_target`, modify it to 150, and analyze the `EMPLOYEES` table:

```
SHOW default_statistics_target ;
SET default_statistics_target to 150;
ANALYZE EMPLOYEES ;
```

View the last time statistics were collected for a table.

```
select relname, last_analyze from pg_stat_all_tables;
```

Summary

Feature	SQL Server	PostgreSQL
Analyze a specific database table	`CREATE STATISTICS MyStatistics ON MyTable (Col1, Col2)`	`ANALYZE EMPLOYEES;`
Analyze a database table while only sampling certain rows	`UPDATE STATISTICS MyTable(MyStatistics) WITH SAMPLE 50 PERCENT;`	Configure the number of entries for the table: `SET default_statistics _target to 150; ANALYZE EMPLOYEES ;`

Feature	SQL Server	PostgreSQL
View last time statistics were collected	`DBCC SHOW_STATISTICS ('MyTable','MyStatistics');`	`select relname, last`

For more information, see [ANALYZE] and [The Autovacuum Daemon] in the *PostgreSQL documentation*.

Physical Storage

Topics

- SQL Server Columnstore Index and PostgreSQL Columnstore
- SQL Server Indexed Views and PostgreSQL Materialized Views
- SQL Server Partitioning and PostgreSQL Partitions or Table Inheritance

SQL Server Columnstore Index and PostgreSQL Columnstore

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Aurora PostgreSQL offers no comparable feature.

SQL Server Usage

SQL Server provides columnstore indexes that use column-based data storage to compress data and improve query performance in data warehouses. Columnstore indexes are the preferred data storage format for data warehousing and analytic workloads. As a best practice, use Columnstore indexes with fact tables and large dimension workloads.

Examples

The following example creates

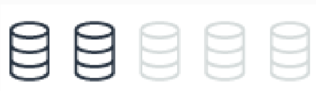
```
CREATE TABLE products(ID [int] NOT NULL, OrderDate [int] NOT NULL, ShipDate [int] NOT
 NULL);
GO

CREATE CLUSTERED COLUMNSTORE INDEX cci_T1 ON products;
GO
```

For more information, see Columnstore indexes: Overview in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) doesn't currently provide a directly comparable alternative for SQL Server columnstore index.

SQL Server Indexed Views and PostgreSQL Materialized Views

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Different paradigm and syntax will require rewriting the application.

SQL Server Usage

The first index created on a view must be a clustered index. Subsequent indexes can be non-clustered indexes. For more information, see [Clustered and nonclustered indexes described](#) in the *SQL Server documentation*.

Before creating an index on a view, the following requirements must be met:

- The `WITH SCHEMABINDING` option must be used when creating the view.
- Verify the `SET` options are correct for all existing tables referenced in the view and for the session. Find the link at the end of this section for required values.
- Ensure that a clustered index on the view is exists.

> **Note**
>
> You can't use indexed views with temporal queries (`FOR SYSTEM_TIME`).

Examples

Set the required SET options, create a view with the `WITH SCHEMABINDING` option, and create an index on this view.

```
SET NUMERIC_ROUNDABORT OFF;
SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT,
  QUOTED_IDENTIFIER, ANSI_NULLS ON;
GO

CREATE VIEW Sales.Ord_view
WITH SCHEMABINDING
AS
  SELECT SUM(Price*Qty*(1.00-Discount)) AS Revenue,
    OrdTime, ID, COUNT_BIG(*) AS COUNT
  FROM Sales.OrderDetail AS ordet, Sales.OrderHeader AS ordhead
  WHERE ordet.SalesOrderID = ordhead.SalesOrderID
  GROUP BY OrdTime, ID;
GO

CREATE UNIQUE CLUSTERED INDEX IDX_V1
  ON Sales.Ord_view (OrdTime, ID);
GO
```

For more information, see Create Indexed Views in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL doesn't support indexed views, but does provide similar functionality with materialized views. You can run queries associated with materialized views, and populate the view data with the `REFRESH` command.

The PostgreSQL implementation of materialized views has three primary limitations:

- You can refresh PostgreSQL materialized views either manually or using a job running the `REFRESH MATERIALIZED VIEW` command. To refresh materialized views automatically, create a trigger.
- PostgreSQL materialized views only support complete or full refresh.
- DML on materialized views isn't supported.

In some cases, when the tables are big, full REFRESH can cause performance issues. In this case, you can use triggers to sync between one table to the new table. You can use the new table as an indexed view.

Examples

The following example creates a materialized view named sales_summary using the sales table as the source.

```
CREATE MATERIALIZED VIEW sales_summary AS
SELECT seller_no,sale_date,sum(sale_amt)::numeric(10,2) as sales_amt
FROM sales
WHERE sale_date < CURRENT_DATE
GROUP BY seller_no, sale_date
ORDER BY seller_no, sale_date;
```

The following example runs a manual refresh of the materialized view:

```
REFRESH MATERIALIZED VIEW sales_summary;
```

Note

The materialized view data isn't refreshed automatically if changes occur to its underlying tables. For automatic refresh of materialized view data, a trigger on the underlying tables must be created.

Creating a Materialized View

When you create a materialized view in PostgreSQL, it uses a regular database table underneath. You can create database indexes on the materialized view directly and improve performance of queries that access the materialized view.

Example

The following example creates an index on the sellerno and sale_date columns of the sales_summary materialized view.

```
CREATE UNIQUE INDEX sales_summary_seller
```

```
ON sales_summary (seller_no, sale_date);
```

Summary

Feature	Indexed views	Materialized view

Create materialized view	<pre>SET NUMERIC_ROUNDABORT OFF;
	SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL,
	  ARITHABORT, QUOTED_IDENTIFIER, ANSI_NULLS ON;
	GO
	
	CREATE VIEW Sales.Ord_view WITH SCHEMABINDING
	  AS SELECT SUM(Price*Qty*(1.00-Discount)) AS Revenue,
	  OrdTime, ID, COUNT_BIG(*) AS
	    COUNT FROM Sales.OrderDetail AS ordet,
	  Sales.OrderHeader AS ordhead
	    WHERE ordet.SalesOrderID = ordhead.SalesOrderID
	    GROUP BY OrdTime, ID;
	GO
	
	CREATE UNIQUE CLUSTERED INDEX IDX_V1 ON Sales.Ord_view (OrdTime, ID);
	GO</pre>
		<pre>CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM employees;</pre>

<table>
<tr><th>Feature</th><th>Indexed views</th><th>Materialized view</th></tr>
<tr><td>Indexed refreshed</td><td>Automatic</td><td>Manual. You can automate refreshes using triggers.

Create a trigger that initiates a refresh after every DML command on the underlying tables:

<pre><code>CREATE OR REPLACE
 FUNCTION
refresh_mv1()
returns trigger
 language plpgsql as
$$ begin
refresh materialized
 view mv1;
return null;
end $$;</code></pre>
Create the <code>refresh_mv1</code> trigger after insert, update, delete, or truncate on employees. For each statement, run the <code>refresh_mv1();</code> procedure.</td></tr>
<tr><td>DML</td><td>Supported</td><td>Not Supported</td></tr>
</table>

For more information, see Materialized Views in the *PostgreSQL documentation*.

SQL Server Partitioning and PostgreSQL Partitions or Table Inheritance

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Partitioning	PostgreSQL doesn't support `LEFT` partition or foreign keys referencing partitioned tables.

SQL Server Usage

SQL Server provides a logical and physical framework for partitioning table and index data. SQL Server 2017 supports up to 15,000 partitions.

Partitioning separates data into logical units. You can store these logical units in more than one file group. SQL Server partitioning is horizontal, where data sets of rows are mapped to individual partitions. A partitioned table or index is a single object and must reside in a single schema within a single database. Objects composed of disjointed partitions aren't allowed.

All DQL and DML operations are partition agnostic except for the special `$partition` predicate. You can use the `$partition` predicate for explicit partition elimination.

Partitioning is typically needed for very large tables to address the following management and performance challenges:

- Deleting or inserting large amounts of data in a single operation with partition switching instead of individual row processing while maintaining logical consistency.
- You can split and customize maintenance operations for each partition. For example, you can compress older data partitions. Then you can rebuild and reorganize more frequently active partitions.
- Partitioned tables may use internal query optimization techniques such as collocated and parallel partitioned joins.

- You can optimize physical storage performance by distributing IO across partitions and physical storage channels.
- Concurrency improvements due to the engine's ability to escalate locks to the partition level rather than the whole table.

Partitioning in SQL Server uses the following three objects:

- **A partitioning column** is used by the partition function to partition the table or index. The value of this column determines the logical partition to which it belongs. You can use computed columns in a partition function as long as they are explicitly `PERSISTED`. Partitioning columns may be any data type that is a valid index column with less than 900 bytes for each key except timestamp and LOB data types.
- **A partition function** is a database object that defines how the values of the partitioning columns for individual tables or index rows are mapped to a logical partition. The partition function describes the partitions for the table or index and their boundaries.
- **A partition scheme** is a database object that maps individual logical partitions of a table or an index to a set of file groups, which in turn consist of physical operating system files. Placing individual partitions on individual file groups enables backup operations for individual partitions (by backing their associated file groups).

Syntax

```
CREATE PARTITION FUNCTION <Partition Function>(<Data Type>)
AS RANGE [ LEFT | RIGHT ]
FOR VALUES (<Boundary Value 1>,...)[;]
```

```
CREATE PARTITION SCHEME <Partition Scheme>
AS PARTITION <Partition Function>
[ALL] TO (<File Group> | [ PRIMARY ] [,...])[;]
```

```
CREATE TABLE <Table Name> (<Table Definition>)
ON <Partition Schema> (<Partitioning Column>);
```

Examples

The following example creates a partitioned table.

```
CREATE PARTITION FUNCTION PartitionFunction1 (INT)
AS RANGE LEFT FOR VALUES (1, 1000, 100000);
```

```
CREATE PARTITION SCHEME PartitionScheme1
AS PARTITION PartitionFunction1
ALL TO (PRIMARY);
```

```
CREATE TABLE PartitionTable (
  Col1 INT NOT NULL PRIMARY KEY,
  Col2 VARCHAR(20)
)
ON PartitionScheme1 (Col1);
```

For more information, see Partitioned Tables and Indexes, CREATE TABLE (Transact-SQL), CREATE PARTITION SCHEME (Transact-SQL), and CREATE PARTITION FUNCTION (Transact-SQL) in the *SQL Server documentation*.

PostgreSQL Usage

Starting with PostgreSQL 10, there is an equivalent option to the SQL Server Partitions when using RANGE or LIST partitions. Support for HASH partitions is expected to be included in PostgreSQL 11.

Prior to PostgreSQL 10, the table partitioning mechanism in PostgreSQL differed from SQL Server. Partitioning in PostgreSQL was implemented using table inheritance. Each table partition was represented by a child table which was referenced to a single parent table. The parent table remained empty and was only used to represent the entire table data set (as a meta-data dictionary and as a query source).

In PostgreSQL 10, you still need to create the partition tables manually, but you don't need to create triggers or functions to redirect data to the right partition.

Some of the partitioning management operations are performed directly on the sub-partitions (sub-tables). You can query the partitioned table.

Starting with PostgreSQL 11, the following features were added:

- For partitioned tables, a default partition can now be created that will store data which can't be redirected to any other explicit partitions

- In addition to partitioning by ranges and lists, tables can now be partitioned by a hashed key.
- When UPDATE changes values in a column that's used as partition key in partitioned table, data is moved to proper partitions.
- An index can now be created on a partitioned table. Corresponding indexes will be automatically created on individual partitions.
- Foreign keys can now be created on a partitioned table. Corresponding foreign key constraints will be propagated to individual partitions
- Triggers FOR EACH ROW can now be created on a partitioned table. Corresponding triggers will be automatically created on individual partitions as well.
- When attaching or detaching new partition to a partitioned table with the foreign key, foreign key enforcement triggers are correctly propagated to a new partition.

For more information, see Inheritance and Table Partitioning in the *PostgreSQL documentation*.

Using The Partition Mechanism

List Partition

```
CREATE TABLE emps (
  emp_id SERIAL NOT NULL,
  emp_name VARCHAR(30) NOT NULL)
PARTITION BY LIST (left(lower(emp_name), 1));

CREATE TABLE emp_abc
  PARTITION OF emps (
  CONSTRAINT emp_id_nonzero CHECK (emp_id != 0)
) FOR VALUES IN ('a', 'b', 'c');

CREATE TABLE emp_def
  PARTITION OF emps (
  CONSTRAINT emp_id_nonzero CHECK (emp_id != 0)
) FOR VALUES IN ('d', 'e', 'f');

INSERT INTO emps VALUES (DEFAULT, 'Andrew');

row inserted.

INSERT INTO emps VALUES (DEFAULT, 'Chris');
```

```
row inserted.

INSERT INTO emps VALUES (DEFAULT, 'Frank');

row inserted.

INSERT INTO emps VALUES (DEFAULT, 'Pablo');

SQL Error [23514]: ERROR: no partition of relation "emps" found for row
Detail: Partition key of the failing row contains ("left"(lower(emp_name::text), 1)) =
 (p).
```

To prevent the error shown in the preceding example, make sure that all partitions exist for all possible values in the column that partitions the table. The default partition feature was added in PostgreSQL 11.

Use the `MAXVALUE` and `MINVALUE` in your `FROM/TO` clause. This can help you get all values with `RANGE` partitions without the risk of creating new partitions.

Range partition

```
CREATE TABLE sales (
  saledate DATE NOT NULL,
  item_id INT,
  price FLOAT
) PARTITION BY RANGE (saledate);

CREATE TABLE sales_2018q1
  PARTITION OF sales (
  price DEFAULT 0
) FOR VALUES FROM ('2018-01-01') TO ('2018-03-31');

CREATE TABLE sales_2018q2
  PARTITION OF sales (
  price DEFAULT 0
) FOR VALUES FROM ('2018-04-01') TO ('2018-06-30');

CREATE TABLE sales_2018q3
  PARTITION OF sales (
  price DEFAULT 0
) FOR VALUES FROM ('2018-07-01') TO ('2018-09-30');
```

```
INSERT INTO sales VALUES (('2018-01-08'),3121121, 100);

row inserted.

INSERT INTO sales VALUES (('2018-04-20'),4378623);

row inserted.

INSERT INTO sales VALUES (('2018-08-13'),3278621, 200);

row inserted.
```

When you create a table with PARTITION OF clause, you can still use the PARTITION BY clause with it. In this case, the PARTITION BY clause creates a sub-partition.

A sub-partition can be the same type as the partition table it is related to, or another partition type.

List combined with range partition

The following example creates a list partition and sub partitions by range.

```
CREATE TABLE salers (
  emp_id serial not null,
  emp_name varchar(30) not null,
  sales_in_usd int not null,
  sale_date date not null
) PARTITION BY LIST (left(lower(emp_name), 1));

CREATE TABLE emp_abc
  PARTITION OF salers (
  CONSTRAINT emp_id_nonzero CHECK (emp_id != 0)
) FOR VALUES IN ('a', 'b', 'c') PARTITION BY RANGE (sale_date);

CREATE TABLE emp_def
  PARTITION OF salers (
  CONSTRAINT emp_id_nonzero CHECK (emp_id != 0)
) FOR VALUES IN ('d', 'e', 'f') PARTITION BY RANGE (sale_date);

CREATE TABLE sales_abc_2018q1
  PARTITION OF emp_abc (
  sales_in_usd DEFAULT 0
) FOR VALUES FROM ('2018-01-01') TO ('2018-03-31');
```

```
CREATE TABLE sales_abc_2018q2
  PARTITION OF emp_abc (
  sales_in_usd DEFAULT 0
) FOR VALUES FROM ('2018-04-01') TO ('2018-06-30');

CREATE TABLE sales_abc_2018q3
  PARTITION OF emp_abc (
  sales_in_usd DEFAULT 0
) FOR VALUES FROM ('2018-07-01') TO ('2018-09-30');

CREATE TABLE sales_def_2018q1
  PARTITION OF emp_def (
  sales_in_usd DEFAULT 0
) FOR VALUES FROM ('2018-01-01') TO ('2018-03-31');

CREATE TABLE sales_def_2018q2
  PARTITION OF emp_def (
  sales_in_usd DEFAULT 0
) FOR VALUES FROM ('2018-04-01') TO ('2018-06-30');

CREATE TABLE sales_def_2018q3
  PARTITION OF emp_def (
  sales_in_usd DEFAULT 0
) FOR VALUES FROM ('2018-07-01') TO ('2018-09-30');
```

Implementing List Table Partitioning with Inheritance Tables

For older PostgreSQL versions, follow these steps to implement list table partitioning using inherited tables:

1. Create a parent table from which all child tables or partitions will inherit.
2. Create child tables that inherit from the parent table. This is similar to creating table partitions. The child tables should have an identical structure to the parent table.
3. Create indexes on each child table. Optionally, add constraints to define allowed values in each table. For example, add primary keys or check constraints.
4. Create a database trigger to redirect data inserted into the parent table to the appropriate child table.
5. Make sure that the PostgreSQL `constraint_exclusion` parameter is turned on and set to partition. This parameter ensures the queries are optimized for working with table partitions.

```
show constraint_exclusion;

constraint_exclusion
partition
```

For more information, see constraint_exclusion in the *PostgreSQL documentation*.

PostgreSQL 9.6 doesn't support declarative partitioning, nor several of the table partitioning features available in SQL Server.

PostgreSQL 9.6 table partitioning doesn't support the creation of foreign keys on the parent table. Alternative solutions include application-centric methods such as using triggers and functions or creating these on the individual tables.

PostgreSQL doesn't support `SPLIT` and `EXCHANGE` of table partitions. For these actions, you will need to plan your data migrations manually (between tables) to replace the data into the right partition.

Examples

The following examples create a PostgreSQL list-partitioned table.

Create the parent table.

```
CREATE TABLE SYSTEM_LOGS
  (EVENT_NO NUMERIC NOT NULL,
  EVENT_DATE DATE NOT NULL,
  EVENT_STR VARCHAR(500),
  ERROR_CODE VARCHAR(10));
```

Create child tables or partitions with check constraints.

```
CREATE TABLE SYSTEM_LOGS_WARNING (
  CHECK (ERROR_CODE IN('err1', 'err2', 'err3'))) INHERITS (SYSTEM_LOGS);

CREATE TABLE SYSTEM_LOGS_CRITICAL (
  CHECK (ERROR_CODE IN('err4', 'err5', 'err6'))) INHERITS (SYSTEM_LOGS);
```

Create indexes on each of the child tables.

```
CREATE INDEX IDX_SYSTEM_LOGS_WARNING ON SYSTEM_LOGS_WARNING(ERROR_CODE);
```

```
CREATE INDEX IDX_SYSTEM_LOGS_CRITICAL ON SYSTEM_LOGS_CRITICAL(ERROR_CODE);
```

Create a function to redirect data inserted into the parent table.

```
CREATE OR REPLACE FUNCTION SYSTEM_LOGS_ERR_CODE_INS()
  RETURNS TRIGGER AS
  $$
  BEGIN
    IF (NEW.ERROR_CODE IN('err1', 'err2', 'err3')) THEN
      INSERT INTO SYSTEM_LOGS_WARNING VALUES (NEW.*);
    ELSIF (NEW.ERROR_CODE IN('err4', 'err5', 'err6')) THEN
      INSERT INTO SYSTEM_LOGS_CRITICAL VALUES (NEW.*);
    ELSE
      RAISE EXCEPTION 'Value out of range, check SYSTEM_LOGS_ERR_CODE_INS ()
 Function!';
    END IF;
    RETURN NULL;
  END;
$$
LANGUAGE plpgsql;
```

Attach the trigger function created in the preceding example to log to the table.

```
CREATE TRIGGER SYSTEM_LOGS_ERR_TRIG
  BEFORE INSERT ON SYSTEM_LOGS
  FOR EACH ROW EXECUTE PROCEDURE SYSTEM_LOGS_ERR_CODE_INS();
```

Insert data directly into the parent table.

```
INSERT INTO SYSTEM_LOGS VALUES(1, '2015-05-15', 'a...', 'err1');
INSERT INTO SYSTEM_LOGS VALUES(2, '2016-06-16', 'b...', 'err3');
INSERT INTO SYSTEM_LOGS VALUES(3, '2017-07-17', 'c...', 'err6');
```

View results from across all the different child tables.

```
SELECT * FROM SYSTEM_LOGS;

event_no  event_date  event_str
1         2015-05-15  a...
2         2016-06-16  b...
```

```
3         2017-07-17  c...

SELECT * FROM SYSTEM_LOGS_WARNING;

event_no  event_date  event_str  error_code
1         2015-05-15  a...       err1
2         2016-06-16  b...       err3

SELECT * FROM SYSTEM_LOGS_CRITICAL;

event_no  event_date  event_str  error_cod
3         2017-07-17  c...       err6
```

The following examples create a PostgreSQL range-partitioned table:

Create the parent table.

```
CREATE TABLE SYSTEM_LOGS
(EVENT_NO NUMERIC NOT NULL,
EVENT_DATE DATE NOT NULL,
EVENT_STR VARCHAR(500));
```

Create the child tables or partitions with check constraints.

```
ExCREATE TABLE SYSTEM_LOGS_2015 (CHECK (EVENT_DATE >= DATE '2015-01-01' AND EVENT_DATE
 < DATE '2016- 01-01')) INHERITS (SYSTEM_LOGS);

CREATE TABLE SYSTEM_LOGS_2016 (CHECK (EVENT_DATE >= DATE '2016-01-01' AND EVENT_DATE <
 DATE '2017-01-01')) INHERITS (SYSTEM_LOGS);

CREATE TABLE SYSTEM_LOGS_2017 (CHECK (EVENT_DATE >= DATE '2017-01-01' AND EVENT_DATE <=
 DATE '2017-12-31')) INHERITS (SYSTEM_LOGS);ample
```

Create indexes on all child tables.

```
CREATE INDEX IDX_SYSTEM_LOGS_2015 ON SYSTEM_LOGS_2015(EVENT_DATE);
CREATE INDEX IDX_SYSTEM_LOGS_2016 ON SYSTEM_LOGS_2016(EVENT_DATE);
CREATE INDEX IDX_SYSTEM_LOGS_2017 ON SYSTEM_LOGS_2017(EVENT_DATE);
```

Create a function to redirect data inserted into the parent table.

```
CREATE OR REPLACE FUNCTION SYSTEM_LOGS_INS ()
```

```
  RETURNS TRIGGER AS
  $$
  BEGIN
    IF (NEW.EVENT_DATE >= DATE '2015-01-01' AND
      NEW.EVENT_DATE < DATE '2016-01-01') THEN
      INSERT INTO SYSTEM_LOGS_2015 VALUES (NEW.*);
    ELSIF (NEW.EVENT_DATE >= DATE '2016-01-01' AND
      NEW.EVENT_DATE < DATE '2017-01-01') THEN
      INSERT INTO SYSTEM_LOGS_2016 VALUES (NEW.*);
    ELSIF (NEW.EVENT_DATE >= DATE '2017-01-01' AND
      NEW.EVENT_DATE <= DATE '2017-12-31') THEN
      INSERT INTO SYSTEM_LOGS_2017 VALUES (NEW.*);
    ELSE
      RAISE EXCEPTION 'Date out of range. check SYSTEM_LOGS_INS () function!';
    END IF;
    RETURN NULL;
  END;
$$
LANGUAGE plpgsql;
```

Attach the trigger function created in the preceding example to log to the SYSTEM_LOGS table.

```
CREATE TRIGGER SYSTEM_LOGS_TRIG BEFORE INSERT ON SYSTEM_LOGS
  FOR EACH ROW EXECUTE PROCEDURE SYSTEM_LOGS_INS ();
```

Insert data directly to the parent table.

```
INSERT INTO SYSTEM_LOGS VALUES (1, '2015-05-15', 'a...');
INSERT INTO SYSTEM_LOGS VALUES (2, '2016-06-16', 'b...');
INSERT INTO SYSTEM_LOGS VALUES (3, '2017-07-17', 'c...');
```

Test the solution by selecting data from the parent and child tables.

```
SELECT * FROM SYSTEM_LOGS;

event_no  event_date  event_str
1         2015-05-15  a...
2         2016-06-16  b...
3         2017-07-17  c...

SELECT * FROM SYSTEM_LOGS_2015;
```

```
event_no  event_date  event_str
1         2015-05-15  a...
```

Examples of New Partitioning Features of PostgreSQL 11

The following example creates default partitions.

```
CREATE TABLE tst_part(i INT) PARTITION BY RANGE(i);
CREATE TABLE tst_part1 PARTITION OF tst_part FOR VALUES FROM (1) TO (5);
CREATE TABLE tst_part_dflt PARTITION OF tst_part DEFAULT;

INSERT INTO tst_part SELECT generate_series(1,10,1);

SELECT * FROM tst_part1;

i
1
2
3
4
(4 rows)

SELECT * FROM tst_part_dflt;

i
5
6
7
8
9
10
(6 rows)
```

The following example creates hash partitions.

```
CREATE TABLE tst_hash(i INT) PARTITION BY HASH(i);
CREATE TABLE tst_hash_1 PARTITION OF tst_hash FOR VALUES WITH (MODULUS 2, REMAINDER 0);
CREATE TABLE tst_hash_2 PARTITION OF tst_hash FOR VALUES WITH (MODULUS 2, REMAINDER 1);

INSERT INTO tst_hash SELECT generate_series(1,10,1);
```

```
SELECT * FROM tst_hash_1;

i
1
2
(2 rows)

SELECT * FROM tst_hash_2;
i
3
4
5
6
7
8
9
10
(8 rows)
```

The following example runs UPDATE on the partition key.

```
CREATE TABLE tst_part(i INT) PARTITION BY RANGE(i);
CREATE TABLE tst_part1 PARTITION OF tst_part FOR VALUES FROM (1) TO (5);
CREATE TABLE tst_part_dflt PARTITION OF tst_part DEFAULT;

INSERT INTO tst_part SELECT generate_series(1,10,1);

SELECT * FROM tst_part1;
i
1
2
3
4
(4 rows)

SELECT * FROM tst_part_dflt;
i
5
6
7
8
9
10
```

```
(6 rows)

UPDATE tst_part SET i=1 WHERE i IN (5,6);

SELECT * FROM tst_part_dflt;
i
7
8
9
10
(4 rows)

SELECT * FROM tst_part1;
i
1
2
3
4
1
1
(6 rows)
```

Index propagation on partitioned tables:

```
CREATE TABLE tst_part(i INT) PARTITION BY RANGE(i);

CREATE TABLE tst_part1 PARTITION OF tst_part FOR VALUES FROM (1) TO (5);

CREATE TABLE tst_part2 PARTITION OF tst_part FOR VALUES FROM (5) TO (10);

CREATE INDEX tst_part_ind ON tst_part(i);

\d+ tst_part

Partitioned table "public.tst_part"
Column | Type    | Collation | Nullable | Default | Storage | Stats target |
 Description
i      | integer |           |          |         | plain   |              |
Partition key: RANGE (i)
Indexes:
"tst_part_ind" btree (i)
Partitions: tst_part1 FOR VALUES FROM (1) TO (5),
  tst_part2 FOR VALUES FROM (5) TO (10)
```

```
\d+ tst_part1

Table "public.tst_part1"
Column | Type    | Collation | Nullable | Default | Storage | Stats target |
 Description
i      | integer |           |          |         | plain   |              |
Partition of: tst_part FOR VALUES FROM (1) TO (5)
Partition constraint: ((i IS NOT NULL) AND (i >= 1) AND (i < 5))
Indexes:
  "tst_part1_i_idx" btree (i)
Access method: heap

\d+ tst_part2

Table "public.tst_part2"
Column | Type    | Collation | Nullable | Default | Storage | Stats target |
 Description
i      | integer |           |          |         | plain   |              |
Partition of: tst_part FOR VALUES FROM (5) TO (10)
Partition constraint: ((i IS NOT NULL) AND (i >= 5) AND (i < 10))
Indexes:
  "tst_part2_i_idx" btree (i)
Access method: heap
```

Foreign keys propagation on partitioned tables:

```
CREATE TABLE tst_ref(i INT PRIMARY KEY);

ALTER TABLE tst_part ADD CONSTRAINT tst_part_fk FOREIGN KEY (i) REFERENCES tst_ref(i);

\d+ tst_part

Partitioned table "public.tst_part"
Column | Type    | Collation | Nullable | Default | Storage | Stats target |
 Description
i      | integer |           |          |         | plain   |              |
Partition key: RANGE (i)
Indexes:
  "tst_part_ind" btree (i)
Foreign-key constraints:
  "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)
Partitions: tst_part1 FOR VALUES FROM (1) TO (5),
```

```
  tst_part2 FOR VALUES FROM (5) TO (10)

\d+ tst_part1

Table "public.tst_part1"
Column | Type    | Collation | Nullable | Default | Storage | Stats target |
 Description
i      | integer |           |          |         | plain   |              |
Partition of: tst_part FOR VALUES FROM (1) TO (5)
Partition constraint: ((i IS NOT NULL) AND (i >= 1) AND (i < 5))
Indexes:
  "tst_part1_i_idx" btree (i)
Foreign-key constraints:
  TABLE "tst_part" CONSTRAINT "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)
Access method: heap

\d+ tst_part2

Table "public.tst_part2"
Column | Type    | Collation | Nullable | Default | Storage | Stats target |
 Description
i      | integer |           |          |         | plain   |              |
Partition of: tst_part FOR VALUES FROM (5) TO (10)
Partition constraint: ((i IS NOT NULL) AND (i >= 5) AND (i < 10))
Indexes:
  "tst_part2_i_idx" btree (i)
Foreign-key constraints:
  TABLE "tst_part" CONSTRAINT "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)
Access method: heap
```

Triggers propagation on partitioned tables:

```
CREATE TRIGGER some_trigger AFTER UPDATE ON tst_part FOR EACH ROW EXECUTE FUNCTION
 some_func();

\d+ tst_part

Partitioned table "public.tst_part"
Column | Type    | Collation | Nullable | Default | Storage | Stats target |
 Description
i      | integer |           |          |         | plain   |              |
Partition key: RANGE (i)
Indexes:
```

```
  "tst_part_ind" btree (i)
Foreign-key constraints:
  "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)
Triggers:
  some_trigger AFTER UPDATE ON tst_part FOR EACH ROW EXECUTE FUNCTION some_func()
Partitions: tst_part1 FOR VALUES FROM (1) TO (5),
  tst_part2 FOR VALUES FROM (5) TO (10)

\d+ tst_part1

Table "public.tst_part1"
Column | Type    | Collation | Nullable | Default | Storage | Stats target |
 Description
i      | integer |           |          |         | plain   |              |
Partition of: tst_part FOR VALUES FROM (1) TO (5)
Partition constraint: ((i IS NOT NULL) AND (i >= 1) AND (i < 5))
Indexes:
  "tst_part1_i_idx" btree (i)
Foreign-key constraints:
  TABLE "tst_part" CONSTRAINT "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)
Triggers:
  some_trigger AFTER UPDATE ON tst_part1 FOR EACH ROW EXECUTE FUNCTION some_func()
Access method: heap

\d+ tst_part2

Table "public.tst_part2"
Column | Type    | Collation | Nullable | Default | Storage | Stats target |
 Description
i      | integer |           |          |         | plain   |              |
Partition of: tst_part FOR VALUES FROM (5) TO (10)
Partition constraint: ((i IS NOT NULL) AND (i >= 5) AND (i < 10))
Indexes:
  "tst_part2_i_idx" btree (i)
Foreign-key constraints:
  TABLE "tst_part" CONSTRAINT "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)
Triggers:
  some_trigger AFTER UPDATE ON tst_part2 FOR EACH ROW EXECUTE FUNCTION some_func()
Access method: heap
```

Summary

The following table identifies similarities, differences, and key migration considerations.

Feature	SQL Server	Aurora PostgreSQL
Partition types	`RANGE` only	`RANGE`, `LIST`
Partitioned tables scope	All tables are partitioned, some have more than one partition	All tables are partitioned, some have more than one partition
Partition boundary direction	`LEFT` or `RIGHT`	`RIGHT`
Exchange partition	Any partition to any partition	N/A
Partition function	Abstract function object, independent of individual column	Abstract function object, independent of individual column
Partition scheme	Abstract partition storage mapping object	Abstract partition storage mapping object
Limitations on partitioned tables	None — all tables are partitioned	Not all commands are compatible with table inheritance

For more information, see Table Partitioning in the *PostgreSQL documentation*.

Security

Topics

- Column Encryption
- Data Control Language
- Transparent Data Encryption
- Users and Roles

Column Encryption

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Syntax and option differences, similar functionality.

SQL Server Usage

SQL Server provides encryption and decryption functions to secure the content of individual columns. The following list identifies common encryption functions:

- `EncryptByKey` and `DecryptByKey`.
- `EncryptByCert` and `DecryptByCert`.
- `EncryptByPassPhrase` and `DecryptByPassPhrase`.
- `EncryptByAsymKey` and `DecryptByAsymKey`.

You can use these functions anywhere in your code; they aren't limited to encrypting table columns. A common use case is to increase run time security by encrypting of application user security tokens passed as parameters.

These functions follow the general SQL Server encryption hierarchy, which in turn use the Windows Server Data Protection API.

Symmetric encryption and decryption consume minimal resources. You can use them for large data sets.

> **Note**
>
> This section doesn't cover Transparent Data Encryption (TDE) or Always Encrypted end-to-end encryption.

Syntax

General syntax for `EncryptByKey` and `DecryptByKey`:

```
EncryptByKey ( <key GUID> , { 'text to be encrypted' }, { <use authenticator flag>},
 { <authenticator> } );
```

```
DecryptByKey ( 'Encrypted Text' , <use authenticator flag>, { <authenticator> )
```

Examples

The following examples demonstrate how to encrypt an employee Social Security Number.

Create a database master key.

```
USE MyDatabase;
CREATE MASTER KEY
ENCRYPTION BY PASSWORD = '<MyPassword>';
```

Create a certificate and a key.

```
CREATE CERTIFICATE Cert01
WITH SUBJECT = 'SSN';
```

```
CREATE SYMMETRIC KEY SSN_Key
WITH ALGORITHM = AES_256
ENCRYPTION BY CERTIFICATE Cert01;
```

Create an `Employees` table.

```
CREATE TABLE Employees
(
  EmployeeID INT PRIMARY KEY,
  SSN_encrypted VARBINARY(128) NOT NULL
);
```

Open the symmetric key for encryption.

```
OPEN SYMMETRIC KEY SSN_Key
DECRYPTION BY CERTIFICATE Cert01;
```

Insert the encrypted data.

```
INSERT INTO Employees (EmployeeID, SSN_encrypted)
VALUES
(1, EncryptByKey(Key_GUID('SSN_Key') , '1112223333', 1, HashBytes('SHA1',
 CONVERT(VARBINARY, 1)));
```

```
SELECT EmployeeID,
CONVERT(CHAR(10), DecryptByKey(SSN, 1 , HashBytes('SHA1', CONVERT(VARBINARY,
 EmployeeID)))) AS SSN
FROM Employees;

EmployeeID  SSN_Encrypted               SSN
1           0x00F983FF436E32418132...   1112223333
```

For more information, see Encrypt a Column of Data and Encryption Hierarchy in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) provides encryption and decryption functions similar to SQL Server using the `pgcrypto` extension. To use this feature, you must first install the `pgcrypto` extension.

```
CREATE EXTENSION pgcrypto;
```

Aurora PostgreSQL supports many encryption algorithms:

- MD5
- SHA1
- SHA224/256/384/512
- Blowfish
- AES
- Raw encryption
- PGP Symmetric encryption
- PGP Public-Key encryption

This section describes the use of PGP_SYM_ENCRYPT and PGP_SYM_DECRYPT, but there are many more options available. For more information, see the link and the end of this section.

Syntax

Encrypt columns using PGP_SYM_ENCRYPT.

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_decrypt(msg bytea, psw text [, options text ]) returns text
```

Examples

The following examples demonstrate how to encrypt an employee's Social Security Number.

Create the users table.

```
CREATE TABLE users (id SERIAL, name VARCHAR(60), pass TEXT);
```

Insert the encrypted data.

```
INSERT INTO users (name, pass) VALUES ('John',PGP_SYM_ENCRYPT('123456', 'AES_KEY'));
```

Verify the data is encrypted.

```
SELECT * FROM users;
```

```
id  name  pass
2   John
 \xc30d04070302c30d07ff8b3b12f26ad233015a72bab4d3bb73f5a80d5187b1b043149dd961da58e76440ca9eb4a5
```

Query using the encryption key.

```
SELECT name, PGP_SYM_DECRYPT(pass::bytea, 'AES_KEY') as pass
FROM users WHERE (name LIKE '%John%');

name  pass
John  123456
```

Update the data.

```
UPDATE users SET (name, pass) = ('John',PGP_SYM_ENCRYPT('0000', 'AES_KEY')) WHERE
 id='2';

SELECT name, PGP_SYM_DECRYPT(pass::bytea, 'AES_KEY') as pass
  FROM users WHERE (name LIKE '%John%');

name  pass
John  0000
```

For more information, see pgcrypto in the *PostgreSQL documentation*.

Data Control Language

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Similar syntax and similar functionality.

SQL Server Usage

The ANSI standard specifies, and most Relational Database Management Systems (RDBMS) use, GRANT and REVOKE commands to control permissions.

However, SQL Server also provides a DENY command to explicitly restrict access to a resource. DENY takes precedence over GRANT and is needed to avoid potentially conflicting permissions for users having multiple logins. For example, if a user has DENY for a resource through group membership but GRANT access for a personal login, the user is denied access to that resource.

In SQL Server, you can grant permissions at multiple levels from lower-level objects such as columns to higher-level objects such as servers. Permissions are categorized for specific services and features such as the service broker.

You can use permissions in conjunction with database users and roles. For more information, see Users and Roles.

Syntax

Simplified syntax for SQL Server DCL commands:

```
GRANT { ALL [ PRIVILEGES ] } | <permission> [ ON <securable> ] TO <principal>

DENY { ALL [ PRIVILEGES ] } | <permission> [ ON <securable> ] TO <principal>

REVOKE [ GRANT OPTION FOR ] {[ ALL [ PRIVILEGES ] ]|<permission>} [ ON <securable> ]
 { TO | FROM } <principal>
```

For more information, see Permissions Hierarchy (Database Engine) in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) supports the ANSI Data Control Language (DCL) commands GRANT and REVOKE.

Administrators can grant or revoke permissions for individual objects such as a column, a stored function, or a table. You can grant permissions to multiple objects using ALL % IN SCHEMA. In the example preceding, % can be TABLES, SEQUENCES, or FUNCTIONS.

Use the following command to grant select on all tables in schema to a specific user.

```
GRANT SELECT ON ALL TABLES IN SCHEMA <Schema Name> TO <Role Name>;
```

Aurora PostgreSQL provides a `GRANT` permission option that is similar to SQL Server `WITH GRANT OPTION` clause. This permission grants a user permission to further grant the same permission to other users.

```
GRANT EXECUTE
ON FUNCTION demo.Procedure1
TO UserY
WITH GRANT OPTION;
```

The following table identifies Aurora PostgreSQL privileges.

Permissions	Use to
`SELECT`	Use to query rows from table.
`INSERT`	Use to insert rows into a table.
`UPDATE`	Use to update rows in table.
`DELETE`	Use to delete rows from table.
`TRUNCATE`	Use to truncate a table.
`REFERENCES`	Use to create a foreign key constraint.
`TRIGGER`	Use to create a trigger on the specified table.
`CREATE`	The purpose of this permission depends on the target object. For more information, see GRANT in the *PostgreSQL documentation*.
`CONNECT`	Use to connect to the specified database.
`TEMPORARY` or `TEMP`	Use to create temporary tables.
`EXECUTE`	Use to run a function.
`USAGE`	The purpose of this permission depends on the target object. For more information, see GRANT in the *PostgreSQL documentation*.

Permissions	Use to
ALL or ALL PRIVILEGES	Grant all available privileges.

Syntax

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
  [, ...] | ALL [ PRIVILEGES ] }
  ON { [ TABLE ] table_name [, ...]
    | ALL TABLES IN SCHEMA schema_name [, ...] }
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
  [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
  ON [ TABLE ] table_name [, ...]
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
  [, ...] | ALL [ PRIVILEGES ] }
  ON { SEQUENCE sequence_name [, ...]
    | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
  ON DATABASE database_name [, ...]
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON DOMAIN domain_name [, ...]
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON FOREIGN DATA WRAPPER fdw_name [, ...]
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON FOREIGN SERVER server_name [, ...]
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON { FUNCTION function_name ( [ [ argmode ] [ arg_name ] arg_type [, ...] ] ) [,...]
```

```
    | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON LANGUAGE lang_name [, ...]
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
  ON LARGE OBJECT loid [, ...]
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
  ON SCHEMA schema_name [, ...]
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
  ON TABLESPACE tablespace_name [, ...]
  TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON TYPE type_name [, ...]
  TO role_specification [, ...] [ WITH GRANT OPTION ]

where role_specification can be:
  [ GROUP ] role_name
  | PUBLIC
  | CURRENT_USER
  | SESSION_USER

GRANT role_name [, ...] TO role_name [, ...] [ WITH ADMIN OPTION ]
```

Examples

Grant SELECT permission to a user on all tables in the demo database.

```
GRANT SELECT ON ALL TABLES IN SCHEMA emps TO John;
```

Revoke EXECUTE permissions from a user on the EmployeeReport stored procedure.

```
REVOKE EXECUTE ON FUNCTION EmployeeReport FROM John;
```

For more information, see GRANT in the *PostgreSQL documentation*.

Transparent Data Encryption

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Storage level encryption managed by Amazon RDS.

SQL Server Usage

Transparent data encryption (TDE) is an SQL Server feature designed to protect data at rest in the event an attacker obtains the physical media containing database files.

TDE doesn’t require application changes and is completely transparent to users. The storage engine encrypts and decrypts data on-the-fly. Data isn’t encrypted while in memory or on the network. You can turn TDE on or off individually for each database.

TDE encryption uses a Database Encryption Key (DEK) stored in the database boot record, making it available during database recovery. The DEK is a symmetric key signed with a server certificate from the master system database.

In many instances, security compliance laws require TDE for data at rest.

Examples

The following example demonstrates how to enable TDE for a database:

Create a master key and certificate.

```
USE master;
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'MyPassword';
CREATE CERTIFICATE TDECert WITH SUBJECT = 'TDE Certificate';
```

Create a database encryption key.

```
USE MyDatabase;
```

```
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_128
ENCRYPTION BY SERVER CERTIFICATE TDECert;
```

Enable TDE.

```
ALTER DATABASE MyDatabase SET ENCRYPTION ON;
```

For more information, see Transparent data encryption (TDE) in the *SQL Server documentation*.

PostgreSQL Usage

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) provides the ability to encrypt data at rest (data stored in persistent storage) for new database instances. When data encryption is enabled, Amazon Relational Database Service (RDS) automatically encrypts the database server storage, automated backups, read replicas, and snapshots using the AES-256 encryption algorithm.

You can manage the keys used for Amazon Relational Database Service (Amazon RDS) encrypted instances from the Identity and Access Management (IAM) console using the AWS Key Management Service (AWS KMS). If you require full control of a key, you must manage it yourself. You can't delete, revoke, or rotate default keys provisioned by AWS KMS.

The following limitations exist for Amazon RDS encrypted instances:

- You can only enable encryption for an Amazon RDS database instance when you create it, not afterward. It is possible to encrypt an existing database by creating a snapshot of the database instance and then creating an encrypted copy of the snapshot. You can restore the database from the encrypted snapshot. For more information, see Copying a snapshot in the *Amazon Relational Database Service User Guide*.
- Encrypted database instances can't be modified to disable encryption.
- Encrypted Read Replicas must be encrypted with the same key as the source database instance.
- An unencrypted backup or snapshot can't be restored to an encrypted database instance.
- KMS encryption keys are specific to the region where they are created. Copying an encrypted snapshot from one region to another requires the KMS key identifier of the destination region.

Note

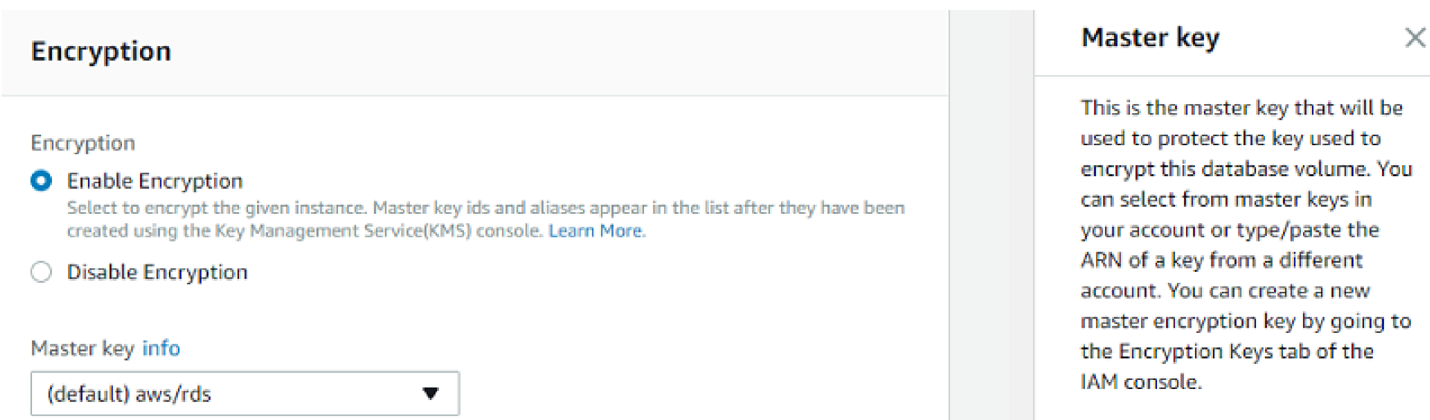
Disabling the key for an encrypted database instance prevents reading from, or writing to, that instance. When Amazon RDS encounters a database instance encrypted by a key to which Amazon RDS doesn't have access, it puts the database instance into a terminal state. In this state, the database instance is no longer available and the current state of the database can't be recovered. To restore the database instance, you must re-enable access to the encryption key for Amazon RDS and then restore the database instance from a backup.

Examples

The following walkthrough demonstrates how to enable TDE.

Enable encryption

In the database settings, enable encryption and choose a master key. You can choose the default key provided for the account or define a specific key based on an IAM KMS ARN from your account or a different account.

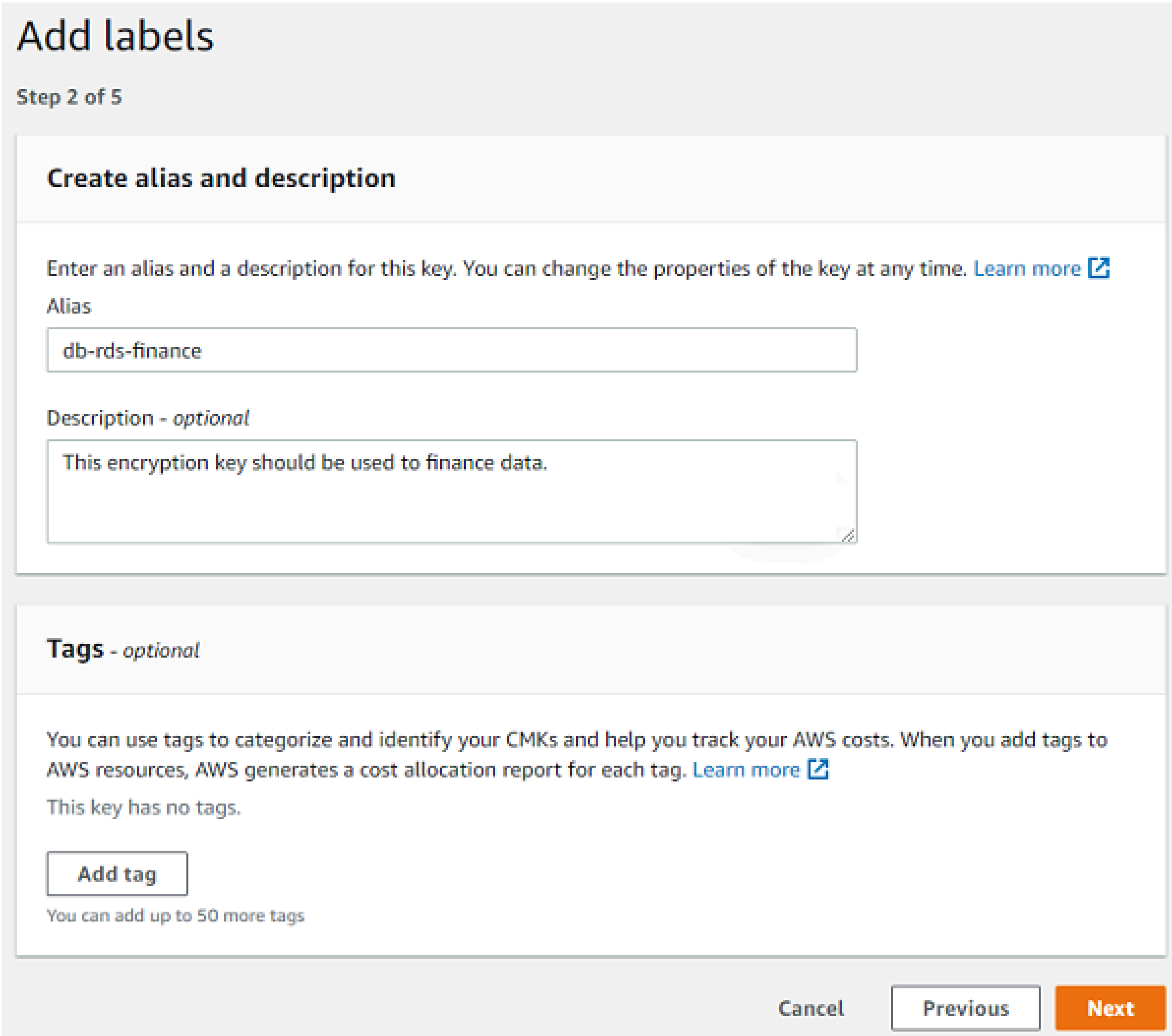


Create an encryption key

To create your own key, browse to the Key Management Service (KMS), choose **Customer managed keys**, and create a new key.

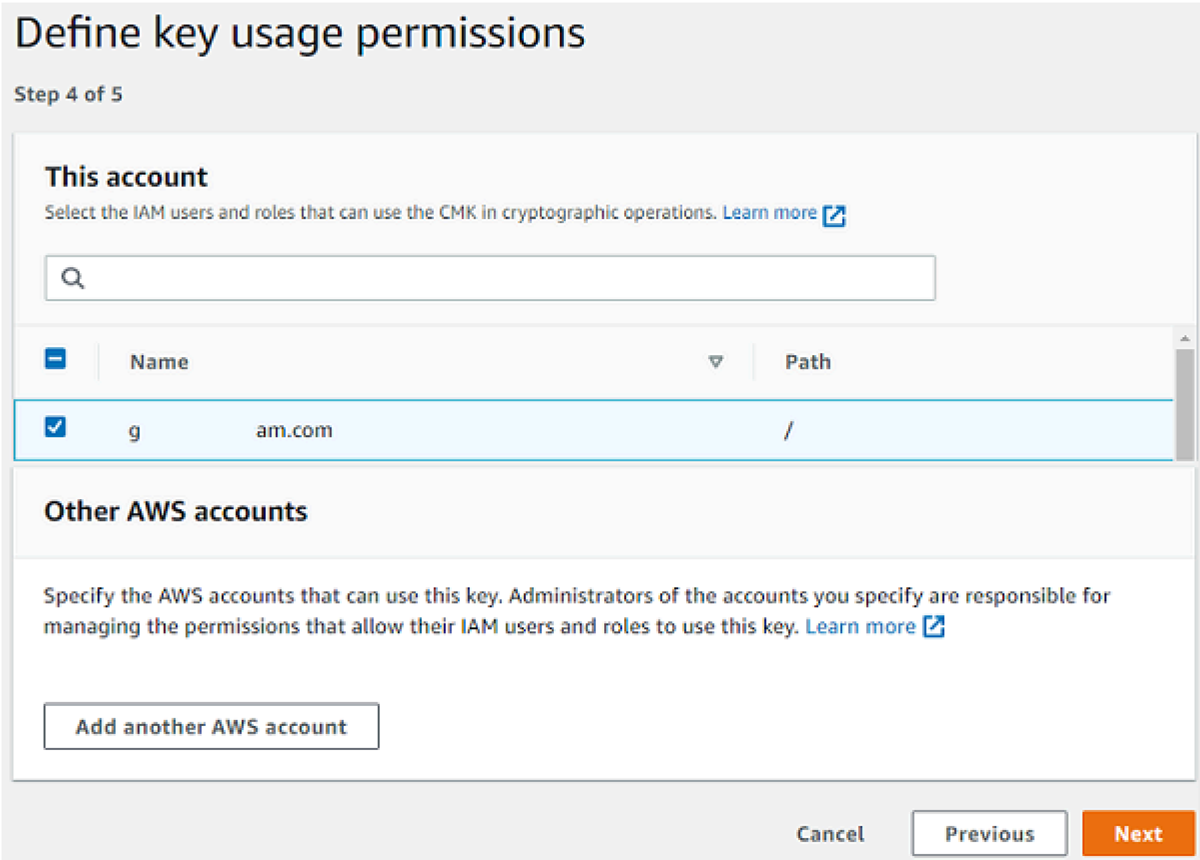
Choose the key type and the key material origin, and then choose **Next**.

Create alias and description, and then choose **Next**.

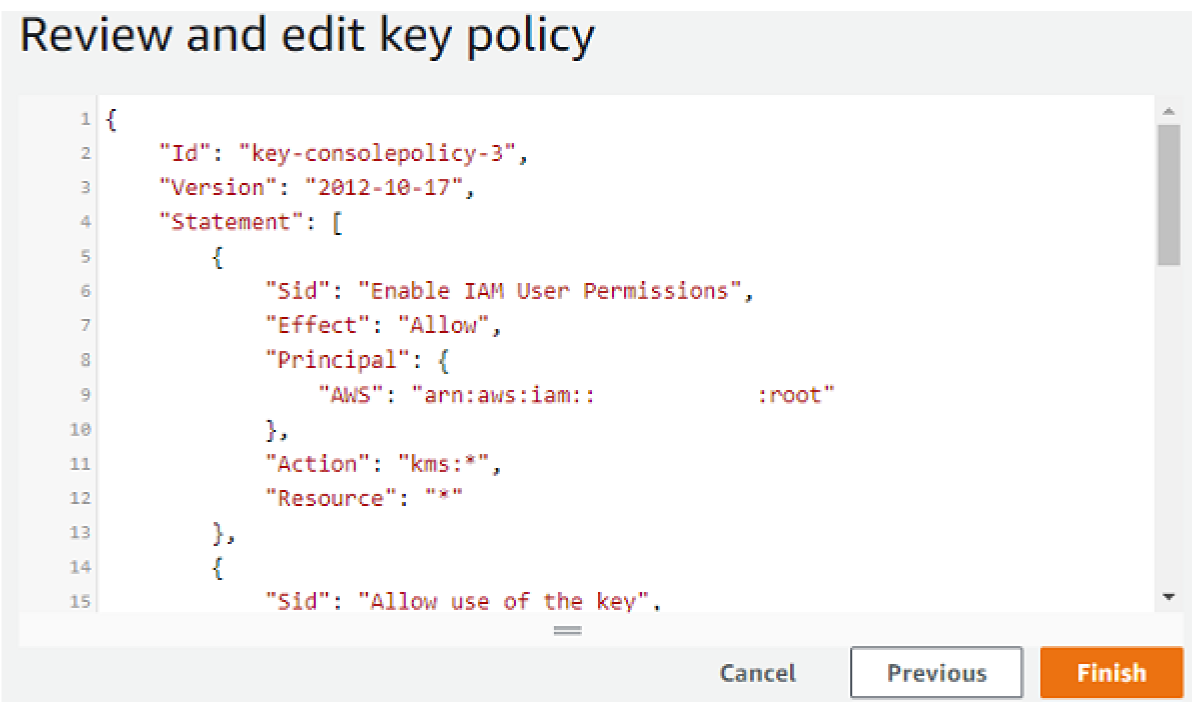


For **Define Key Administrative Permissions**, leave the default values and choose **Next**.

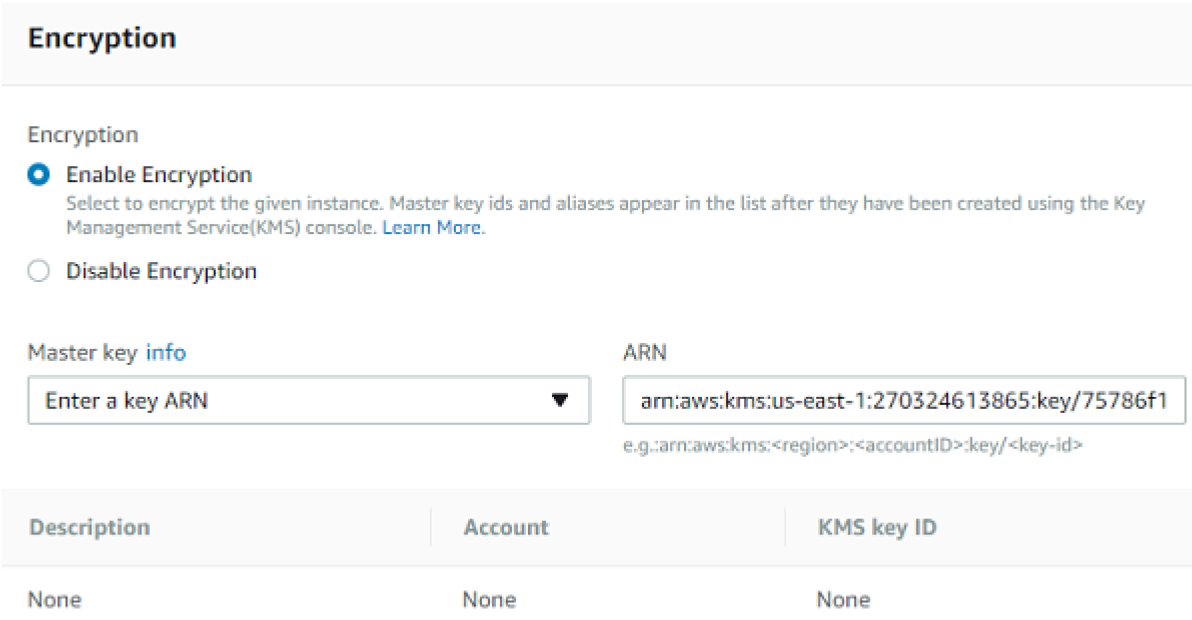
Make sure that you assigned the key to the relevant users who will need to interact with Amazon Aurora.



Review and edit the key policy, and then choose **Finish**.



Now, you can set the master encryption key by using the ARN of the key that you have created or picking it from the list.



Proceed to the finish and launch the instance.

For more information, see Specifying Amazon S3 encryption in the *Amazon Simple Storage Service User Guide* and s3 in the *Command Line Interface Command Reference*.

Users and Roles

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Syntax and option differences, similar functionality. There are no users in PostgreSQL, only roles.

SQL Server Usage

SQL Server provides two layers of security principals: logins at the server level and users at the database level. Logins are mapped to users in one or more databases. Administrators can grant logins server-level permissions that aren't mapped to particular databases such as database creator, system administrator, and security administrator.

SQL Server also supports roles for both the server and the database levels. At the database level, administrators can create custom roles in addition to the general purpose built-in roles.

For each database, administrators can create users and associate them with logins. At the database level, the built-in roles include `db_owner`, `db_datareader`, `db_securityadmin`, and others. A database user can belong to one or more roles (users are assigned to the public role by default and can't be removed). Administrators can grant permissions to roles and then assign individual users to the roles to simplify security management.

Logins are authenticated using either Windows Authentication, which uses the Windows Server Active Directory framework for integrated single sign-on, or SQL authentication, which is managed by the SQL Server service and requires a password, certificate, or asymmetric key for identification. You can create logins that use Windows Authentication for individual users and domain groups.

In previous versions of SQL server, the concepts of user and schema were interchangeable. For backward compatibility, each database has several existing schemas, including a default schema named dbo which is owned by the `db_owner` role. Logins with system administrator privileges are automatically mapped to the dbo user in each database. Typically, you don't need to migrate these schemas.

Examples

Create a login.

```
CREATE LOGIN MyLogin WITH PASSWORD = 'MyPassword'
```

Create a database user for `MyLogin`.

```
USE MyDatabase; CREATE USER MyUser FOR LOGIN MyLogin;
```

Assign `MyLogin` to a server role.

```
ALTER SERVER ROLE dbcreator ADD MEMBER 'MyLogin'
```

Assign `MyUser` to the `db_datareader` role.

```
ALTER ROLE db_datareader ADD MEMBER 'MyUser';
```

For more information, see Database-level roles in the *SQL Server documentation*.

PostgreSQL Usage

PostgreSQL supports only roles; there are no users. However, there is a `CREATE USER` command, which is an alias for `CREATE ROLE` that automatically includes the `LOGIN` permission.

Roles are defined at the database cluster level and are valid in all databases in the PostgreSQL cluster.

Syntax

The following example shows a simplified syntax for `CREATE ROLE` in Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL).

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

  SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | REPLICATION | NOREPLICATION
  | BYPASSRLS | NOBYPASSRLS
  | CONNECTION LIMIT connlimit
  | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
  | VALID UNTIL 'timestamp'
  | IN ROLE role_name [, ...]
  | IN GROUP role_name [, ...]
  | ROLE role_name [, ...]
  | ADMIN role_name [, ...]
  | USER role_name [, ...]
```

```
  | SYSID uid
```

The UNENCRYPTED PASSWORD option was dropped in PostgreSQL 10, the password must be kept encrypted.

Example

Create a new database role called hr_role. Users can use this role to create new databases in the PostgreSQL cluster. Note that this role isn't able to login to the database and act as a database user. In addition, grant SELECT, INSERT, and DELETE privileges on the hr.employees table to the role.

```
CREATE ROLE hr_role;
GRANT SELECT, INSERT,DELETE on hr.employees to hr_role;
```

Summary

The following table summarizes common security tasks and the differences between SQL Server and Aurora PostgreSQL.

Task	SQL Server	Aurora PostgreSQL
View database users	SELECT Name FROM sys.sysusers	SELECT * FROM pg_roles where rolcanlogin = true;
Create a user and password	CREATE USER <User Name> WITH PASSWORD = <PassWord>;	CREATE USER <User Name> WITH PASSWORD '<PassWord>';
Create a role	CREATE ROLE <Role Name>	CREATE ROLE <Role Name>
Change a user's password	ALTER LOGIN <SQL Login> WITH PASSWORD = <PassWord>;	ALTER USER <SQL Login> WITH PASSWORD '<PassWord>';
External authentication	Windows Authentication	N/A

Task	SQL Server	Aurora PostgreSQL
Add a user to a role	`ALTER ROLE <Role Name> ADD MEMBER <User Name>`	`ALTER ROLE <Role Name> SET <property and value>`
Lock a user	`ALTER LOGIN <Login Name> DISABLE`	`REVOKE CONNECT ON DATABASE <database _name> from <Role Name>;`
Grant SELECT on a schema	`GRANT SELECT ON SCHEMA::<Schema Name> to <User Name>`	`GRANT SELECT ON ALL TABLES IN SCHEMA <Schema Name> TO <User Name>;`

For more information, see CREATE ROLE in the *PostgreSQL documentation*.

SQL Server 2018 Deprecated Features List

SQL Server 2018 deprecated feature	Section
TEXT, NTEXT, and IMAGE data types	Data Types
SET ROWCOUNT for DML	Session Options
TIMESTAMP syntax for CREATE TABLE	Creating Tables
DBCC DBREINDEX, INDEXDEFRAG, and SHOWCONTIG	Maintenance Plans
Old SQL Mail	Database Mail
IDENTITY seed, increment, non primary key, and compound	Sequences and Identity
Stored procedures RETURN values	Stored Procedures
GROUP BY ALL, Cube, and Compute By	GROUP BY
DTS	ETL
Old outer join syntax = and =	Table JOIN
'String Alias' = Expression	Migration Quick Tips
DEFAULT keyword for INSERT statements	Migration Quick Tips

Migration Quick Tips

This section provides migration tips that can help save time as you transition from Microsoft SQL Server to Aurora PostgreSQL. They address many of the challenges faced by administrators new to Aurora PostgreSQL. Some of these tips describe functional differences in similar features between SQL Server and Aurora PostgreSQL.

Management

- The equivalent of SQL Server's `CREATE DATABASE… AS SNAPSHOT OF…` resembles Aurora PostgreSQL database cloning. However, unlike SQL Server snapshots, which are read-only, you can update Aurora PostgreSQL cloned databases.
- In Aurora PostgreSQL terminology, *Database Snapshot* is equivalent to SQL Server `BACKUP DATABASE… WITH COPY_ONLY` .
- Partitioning in Aurora PostgreSQL is called `INHERITS` tables and act completely different in terms of management.
- Unlike SQL Server's statistics, Aurora PostgreSQL doesn't collect detailed key value distribution; it relies on selectivity only. When troubleshooting run issues, be aware that parameter values are insignificant to plan choices.
- You can achieve many missing features, such as sending emails, with quick implementations of Amazon services such as Lambda.
- Parameters and backups are managed by Amazon RDS. It is very useful in terms of checking parameter's value against its default and comparing them to another parameter group.
- You can implement high availability in few clicks to create replicas.
- With Database Links, the `db_link` extension is similar to SQL Server.

SQL

- Triggers work differently in Aurora PostgreSQL. You can run triggers for each row. The syntax for inserted and deleted for each row is `new` and `old`.
- Aurora PostgreSQL doesn't support `@@FETCH_STATUS` system parameter for cursors. When you declare cursors in Aurora PostgreSQL, create an explicit `HANDLER` object.

- To run a stored procedure or function, use `SELECT` instead of `EXECUTE`.
- To run a string as a query, use Aurora PostgreSQL Prepared Statements instead of `EXECUTE (<String>)` syntax.
- In Aurora PostgreSQL, terminate `IF` blocks with `END IF` and the `WHILE..LOOP` loops with `END LOOP`.
- In Aurora PostgreSQL, use `START TRANSACTION` to open a transaction instead of `BEGIN TRANSACTION`. Use `COMMIT` and `ROLLBACK` without the `TRANSACTION` keyword.
- Aurora PostgreSQL doesn't use special data types for `UNICODE` data. All string types may use any character set and any relevant collation.
- You can define collations at the server, database, and column level, similar to SQL Server. You can't define collations at the table level.
- Aurora PostgreSQL doesn't support `DELETE <Table Name>` syntax, where you drop the `FROM` keyword. Add the `FROM` keyword to all `DELETE` statements.
- In Aurora PostgreSQL, you can use multiple rows with `NULL` for a `UNIQUE` constraint. In SQL Server, you can only use one. Aurora PostgreSQL follows the behavior specified in the ANSI standard.
- Aurora PostgreSQL `SERIAL` column property is similar to `IDENTITY` in SQL Server. However, there is a major difference in the way sequences are maintained. SQL Server caches a set of values in memory and records the last allocation on disk. When the service restarts, some values may be lost, but the sequence continues from where it left off. In Aurora PostgreSQL, each time you restart the service, the seed value to `SERIAL` is reset to one increment interval larger than the largest existing value. Sequence position isn't maintained across service restarts.
- Parameter names in Aurora PostgreSQL don't require a preceding `@`. You can declare local variables such as `SET schema.test = value` and get the value by running the `SELECT current_setting('username.test');` query.
- Local parameter scope isn't limited to the run scope. You can define or set a parameter in one statement, run it, and then query it in the following batch.
- Error handling in Aurora PostgreSQL has less features, but for special requirements, you can log or send alerts by inserting into tables or catching errors.
- Aurora PostgreSQL doesn't support the `MERGE` statement. Use the `REPLACE` statement and the `INSERT… ON DUPLICATE KEY UPDATE` statement as alternatives.
- In Aurora PostgreSQL, you can't concatenate strings with the + operator. Use the `CONCAT` function instead. For example, `CONCAT('A', 'B')`.

- Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) doesn't support aliasing in the select list using the `String Alias = Expression`. Aurora PostgreSQL treats it as a logical predicate, returns `0` or `FALSE`, and will alias the column with the full expression. Use the `AS` syntax instead. Also note that this syntax has been deprecated as of SQL Server 2008 R2.
- Aurora PostgreSQL has a large set of string functions that is much more diverse than SQL Server. Some of the more useful string functions are:
  - `TRIM` isn't limited to full trim or spaces. The syntax is `TRIM([{BOTH | LEADING | TRAILING} [<remove string>] FROM] <source string>))`.
  - `LENGTH` in PostgreSQL is equivalent to `DATALENGTH` in T-SQL. `CHAR_LENGTH` is the equivalent of T-SQL `LENGTH`.
  - `SUBSTRING_INDEX` returns a substring from a string before the specified number of occurrences of the delimiter.
  - `FIELD` returns the index position of the first argument in the subsequent arguments.
  - `POSITION` returns the index position of the first argument within the second argument.
  - `REGEXP_MATCHES` provides support for regular expressions.
  - For more information, see String Functions and Operators.
- The Aurora PostgreSQL `CAST` function is for casting between collation and not other data types. Use `CONVERT` for casting data types.
- Aurora PostgreSQL is much stricter than SQL Server in terms of statement terminators. Make sure that you always use a semicolon at the end of statements.
- In Aurora PostgreSQL, you can't use the `CREATE PROCEDURE` syntax. You can use only the `CREATE FUNCTION` syntax. You can create a function that returns void.
- Beware of control characters when copying and pasting a script to Aurora PostgreSQL clients. Aurora PostgreSQL is much more sensitive to control characters than SQL Server and they result in frustrating syntax errors that are hard to find.