

ABD304

AWS re:Invent

Best Practices for Data Warehousing with Amazon Redshift & Redshift Spectrum

Tony Gibbs, Senior Data Warehousing Solutions Architect

November 27, 2017

AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Most companies are over-run with data, yet they lack critical insights to make timely and accurate business decisions. They are missing the opportunity to combine large amounts of new, unstructured big data that resides outside their data warehouse with trusted, structured data inside their data warehouse. In this session, we take an in-depth look at how modern data warehousing blends and analyzes all your data, inside and outside your data warehouse without moving the data, to give you deeper insights to run your business. We will cover best practices on how to design optimal schemas, load data efficiently, and optimize your queries to deliver high throughput and performance.

Are you an Amazon Redshift user?



AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Amazon Redshift Best Practices Overview

- History and development
- Concepts and table design
- Data storage, ingestion, and ELT
- Node types and cluster sizing
- Additional resources
- Open Q&A

AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

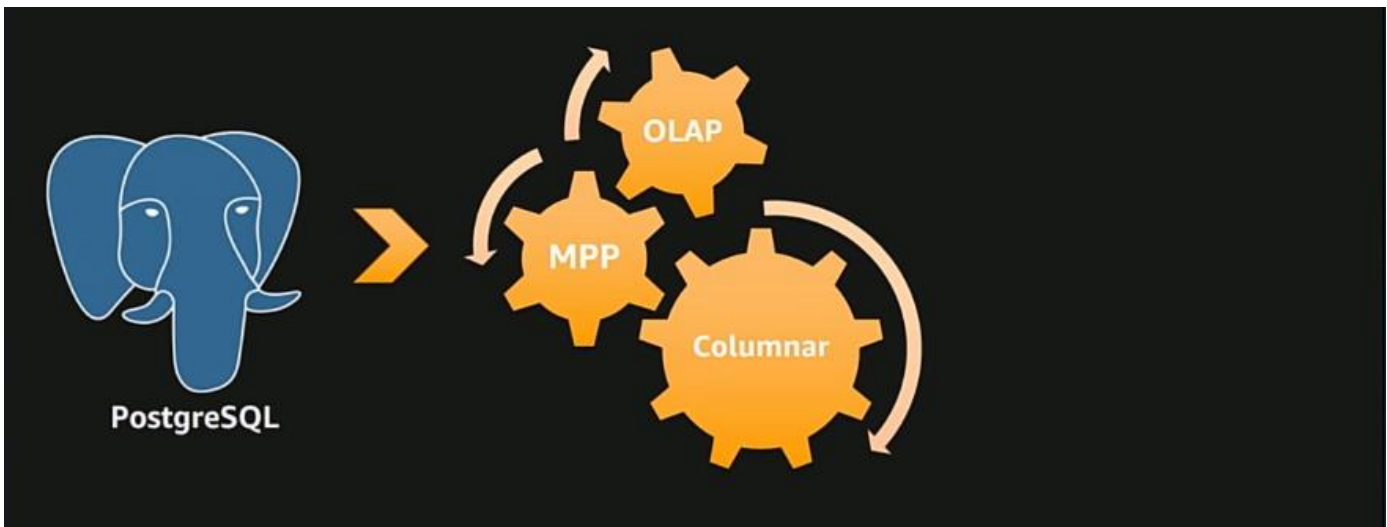


History and Development

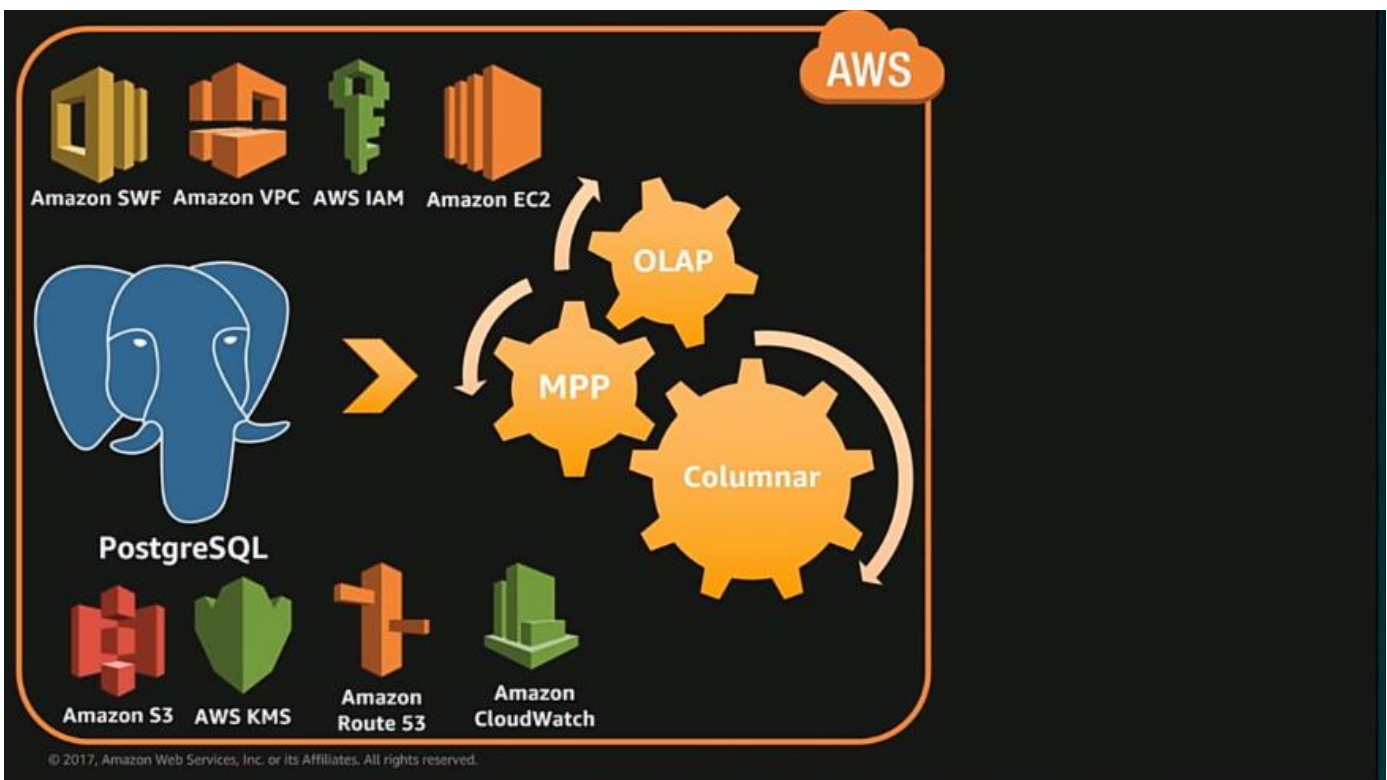


PostgreSQL

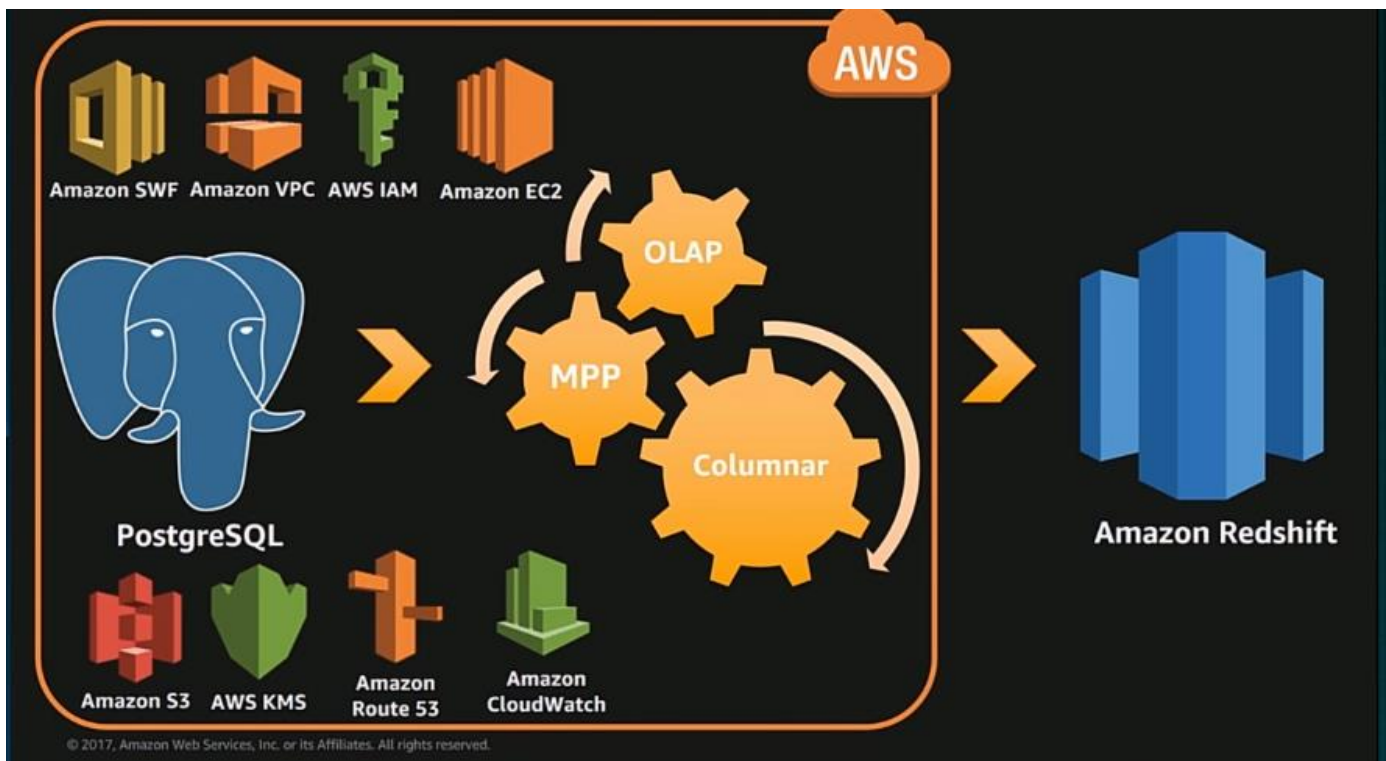
RedShift returns back a PostGRES connection string when you first connect to it.



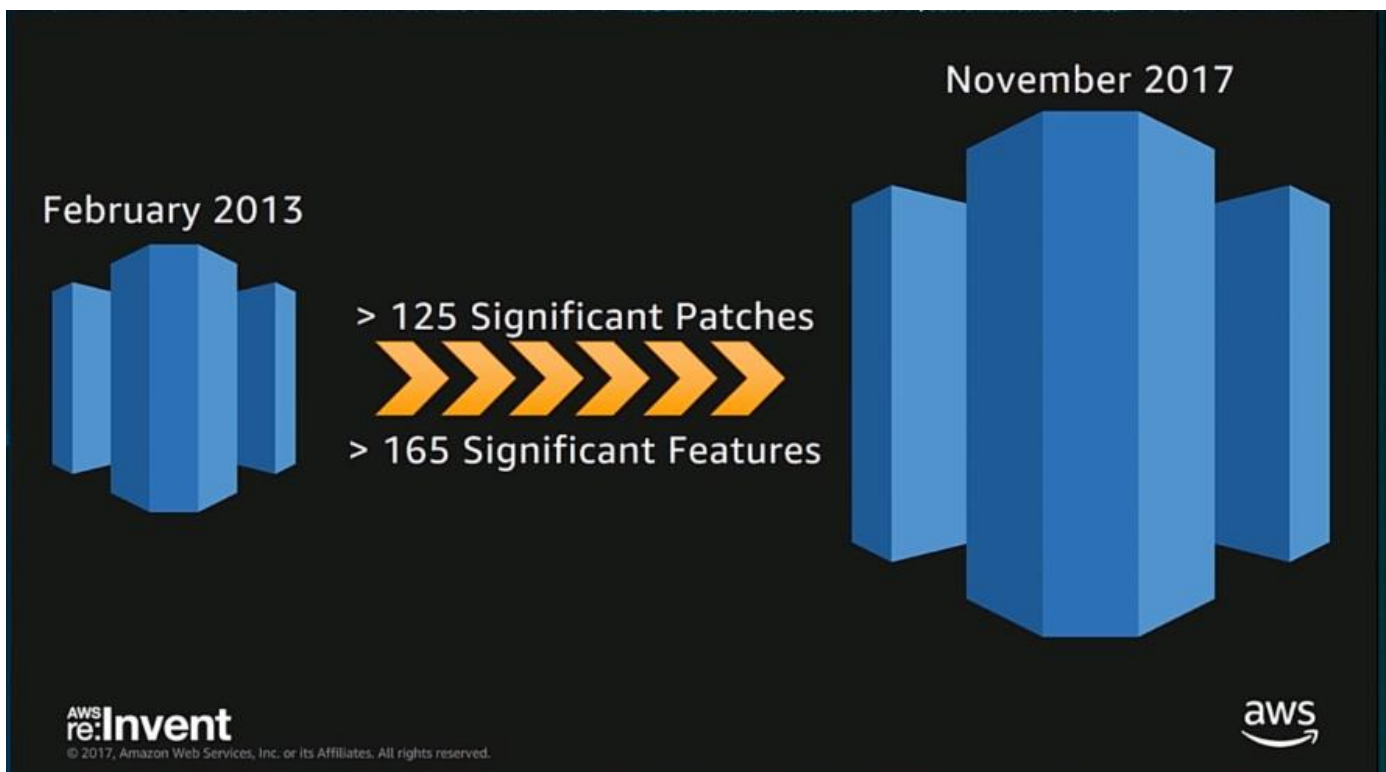
AWS has rewritten the entire Postgres storage engine to use a columnar storage engine, and made the system MPP which is why you can scale it horizontally up to 128 nodes. Then we added a lot of analytics functions, percentiles, and more that are typically used in an analytics data warehouse.



We then wrapped it up with other AWS services like S3 for backup and restore and loading, KMS for encryption, IAM for authentication and connecting to other AWS services



All these components make up Redshift today



Concepts and Table Design

Amazon Redshift Architecture

Massively parallel, shared nothing columnar architecture

Leader node

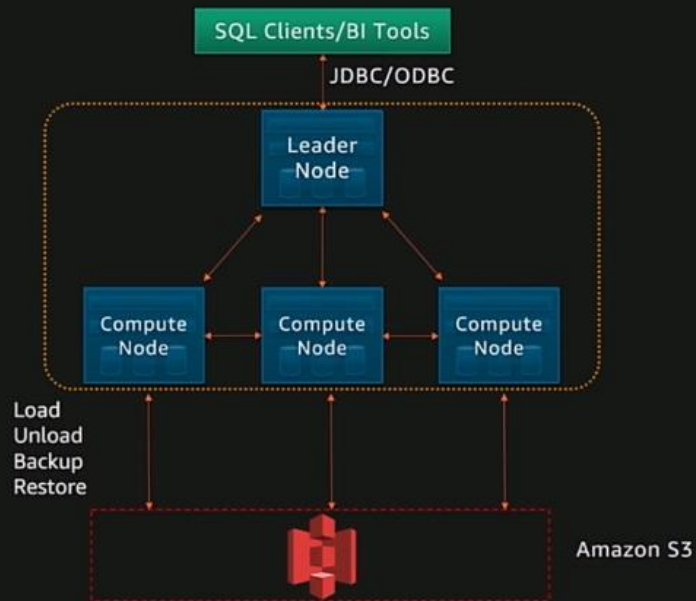
- SQL endpoint
- Stores metadata
- Coordinates parallel SQL processing

Compute nodes

- Local, columnar storage
- Executes queries in parallel
- Load, unload, backup, restore

Amazon Redshift Spectrum nodes

- Execute queries directly against Amazon Simple Storage Service (Amazon S3)



© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

You usually connect to your **Redshift cluster** using the **SQL Clients and BI Tools**, Redshift supports and supplies **JDBC and ODBC drivers** for connection uses but also supports **Postgres drivers** as well if you want to connect using **Postgres Python's driver** to connect with Redshift. The connection is to the **leader node that does all the query parsing, query coordination and also stores the PG Catalog**. Behind the leader node can sit up to **2-128 Compute Nodes** where the actual data resides in Redshift, also the compute nodes are responsible for all the processing and heavy lifting of the data. The **compute nodes do everything in parallel, every compute nodes work on the same query all the time in a Massively Parallel Processing architecture (MPP)**. The **compute nodes also typically talk to S3 for loading data in and out of Redshift**, the backup and restore of data also go directly from the Compute Nodes to S3.

Amazon Redshift Architecture

Massively parallel, shared nothing columnar architecture

Leader node

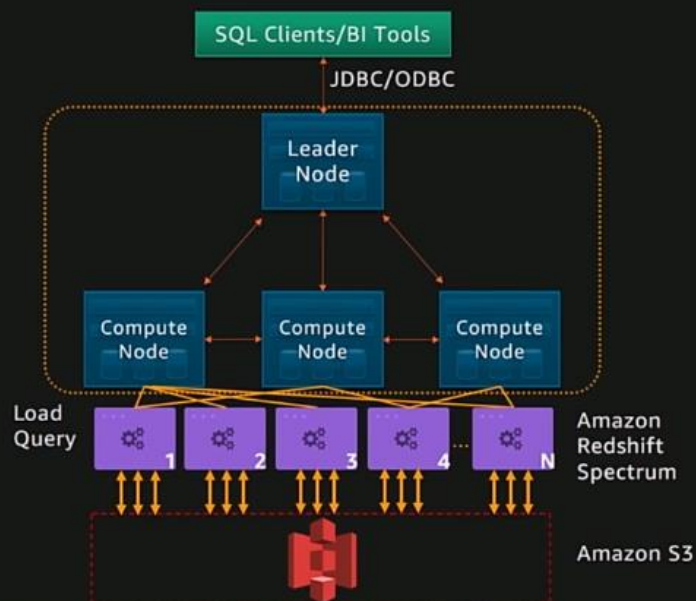
- SQL endpoint
- Stores metadata
- Coordinates parallel SQL processing

Compute nodes

- Local, columnar storage
- Executes queries in parallel
- Load, unload, backup, restore

Amazon Redshift Spectrum nodes

- Execute queries directly against Amazon Simple Storage Service (Amazon S3)



© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Amazon Redshift Spectrum was released earlier this year, this gives Redshift the ability to query S3 directly by provisioning an elastic layer of compute that does the S3 querying and passing the data back up to your Redshift compute node, to the leader node and then to you. You can also load data through the Spectrum layer.

Terminology and Concepts: Columnar

Amazon Redshift uses a columnar architecture for storing data on disk

Goal: reduce I/O for analytics queries

Physically store data on disk by column rather than row

Only read the column data that is required

Redshift is a columnar data warehouse, this means that we store data on disk column by column rather than row by row. This is because the queries that are typically run against your data warehouse like sums, averages, and other aggregate functions typically only operate on a subset of the columns. A columnar architecture greatly reduces I/O on those operations.

Columnar Architecture: Example

```
CREATE TABLE deep_dive (  
  aid INT      --audience_id  
  ,loc CHAR(3) --location  
  ,dt  DATE    --date  
);
```

aid	loc	dt
1	SFO	2017-10-20
2	JFK	2017-10-20
3	SFO	2017-04-01
4	JFK	2017-05-14

aid	loc	dt

```
SELECT min(dt) FROM deep_dive;
```

Row-based storage

- Need to read everything
- Unnecessary I/O

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

In a row-based storage, we would have to read all the rows in the database to find the minimum date **dt**

Columnar Architecture: Example

```
CREATE TABLE deep_dive (  
  aid INT      --audience_id  
, loc CHAR(3) --location  
, dt DATE     --date  
);
```

aid	loc	dt
1	SFO	2017-10-20
2	JFK	2017-10-20
3	SFO	2017-04-01
4	JFK	2017-05-14

aid	loc	dt

```
SELECT min(dt) FROM deep_dive;
```

Column-based storage

- Only scan blocks for relevant column

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

But in a columnar-based storage, we will only need to read data for that date column only to find the minimum data **dt** and thus reduce I/O

Terminology and Concepts: Compression

Goals:

- Allow more data to be stored within an Amazon Redshift cluster
- Improve query performance by decreasing I/O

Impact:

- Allows two to four times more data to be stored within the cluster

By default, COPY automatically analyzes and compresses data on first load into an empty table

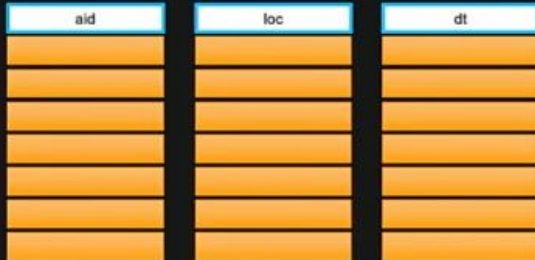
ANALYZE COMPRESSION is a built-in command that will find the optimal compression for each column on an existing table

Compression in Redshift allows you to store significantly more than 4x data in your cluster and also **improves the performance** of Redshift by **reducing I/O** by being able to put more data into each data block. Redshift figures out the best compression to use when you load data into it the first time. The ANALYZE_COMPRESSION utility/command also allows you to find the optimal compression for your data

Compression: Example

```
CREATE TABLE deep_dive (  
  aid INT      --audience_id  
  ,loc CHAR(3) --location  
  ,dt  DATE    --date  
);
```

aid	loc	dt
1	SFO	2017-10-20
2	JFK	2017-10-20
3	SFO	2017-04-01
4	JFK	2017-05-14



Add 1 of 11 different encodings to each column

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Compression: Example

```
CREATE TABLE deep_dive (  
  aid INT      ENCODE ZSTD  
  ,loc CHAR(3) ENCODE BYTEDICT  
  ,dt  DATE    ENCODE RUNLENGTH  
);
```

aid	loc	dt
1	SFO	2017-10-20
2	JFK	2017-10-20
3	SFO	2017-04-01
4	JFK	2017-05-14



More efficient compression is due to storing the same data type in the columnar architecture

Columns grow and shrink independently
Reduces storage requirements
Reduces I/O

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

For the same query, if we modify the DDL with the encode statements above, that is how compression is applied to Redshift.

Best Practices: Compression

Apply compression to all tables

Use ANALYZE COMPRESSION command to find optimal compression

- RAW (no compression) for sparse columns and small tables

Changing column encoding requires a table rebuild

<https://github.com/awslabs/amazon-redshift-utils/tree/master/src/ColumnEncodingUtility>

Verifying columns are compressed: `SELECT "column", type, encoding FROM pg_table_def WHERE tablename = 'deep_dive';`

column	type	encoding
aid	integer	zstd
loc	character(3)	bytedict
dt	date	runlength

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Terminology and Concepts: Blocks

Column data is persisted to 1 MB immutable blocks

Blocks are individually encoded with 1 of 11 encodings

A full block can contain millions of values

Terminology and Concepts: Zone Maps

Goal: eliminates unnecessary I/O

In-memory block metadata

- Contains per-block min and max values
- All blocks automatically have zone maps
- Effectively prunes blocks which cannot contain data for a given query

Zone Maps are metadata about the Blocks.

Terminology and Concepts: Data Sorting

Goal: make queries run faster by increasing the effectiveness of zone maps and reducing I/O

Impact: enables range-restricted scans to prune blocks by leveraging zone maps

Achieved with the table property SORTKEY defined on one or more columns

Optimal sort key is dependent on:

- Query patterns
- Business requirements
- Data profile

Data Sorting is physically sorting data on disk in Redshift, picking one or more columns and sorting them. Sort Keys are mostly put on the columns that you are filtering by in your SQL queries, preferable using the WHERE clause.

Sort Key: Example

```
CREATE TABLE deep_dive (  
  aid INT      --audience_id  
  ,loc CHAR(3) --location  
  ,dt  DATE    --date  
);
```

Add a sort key to one or more columns to physically sort the data on disk

deep_dive		
aid	loc	dt
1	SFO	2017-10-20
2	JFK	2017-10-20
3	SFO	2017-04-01
4	JFK	2017-05-14

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

To add a sort key to your query as above,

Sort Key: Example

```
CREATE TABLE deep_dive (  
    aid INT      --audience_id  
    ,loc CHAR(3) --location  
    ,dt  DATE    --date  
) SORT KEY (dt, loc);
```

Add a sort key to one or more columns to physically sort the data on disk

deep_dive		
aid	loc	dt
1	SFO	2017-10-20
2	JFK	2017-10-20
3	SFO	2017-04-01
4	JFK	2017-05-14

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

We simply modify the DDL by adding a sort key to first the **dt** column and then the **loc** column.

Sort Key: Example

```
CREATE TABLE deep_dive (  
    aid INT      --audience_id  
    ,loc CHAR(3) --location  
    ,dt  DATE    --date  
) SORT KEY (dt, loc);
```

Add a sort key to one or more columns to physically sort the data on disk

deep_dive		
aid	loc	dt
1	SFO	2017-10-20
2	JFK	2017-10-20
3	SFO	2017-04-01
4	JFK	2017-05-14

deep_dive (sorted)		
aid	loc	dt
3	SFO	2017-04-01
4	JFK	2017-05-14
2	JFK	2017-10-20
1	SFO	2017-10-20





© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

The data will end up as above, sorted first by the date **dt** column, then by the location **loc** column. This is how we can make the zone maps more effective

Zone Maps and Sorting: Example

```
SELECT count(*) FROM deep_dive WHERE dt = '06-09-2017';
```

Unsorted table

	MIN: 01-JUNE-2017 MAX: 20-JUNE-2017
	MIN: 08-JUNE-2017 MAX: 30-JUNE-2017
	MIN: 12-JUNE-2017 MAX: 20-JUNE-2017
	MIN: 02-JUNE-2017 MAX: 25-JUNE-2017

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Supposed we have a zone map (metadata held in-memory in Redshift) with 4 data blocks as above, then we have a SQL query to count the number of records on a certain day.

Zone Maps and Sorting: Example

```
SELECT count(*) FROM deep_dive WHERE dt = '06-09-2017';
```

Unsorted table

	MIN: 01-JUNE-2017 MAX: 20-JUNE-2017
	MIN: 08-JUNE-2017 MAX: 30-JUNE-2017
	MIN: 12-JUNE-2017 MAX: 20-JUNE-2017
	MIN: 02-JUNE-2017 MAX: 25-JUNE-2017

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Redshift is going to check the zone maps first and realize it only needs to read 3 data blocks for reduced I/O,

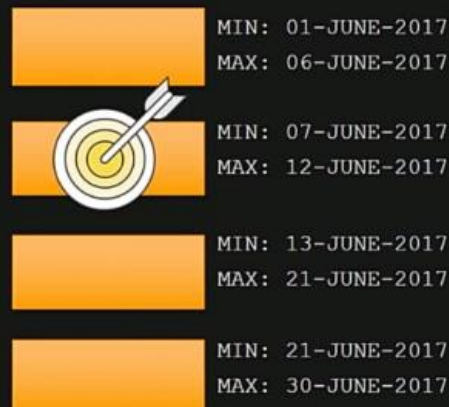
Zone Maps and Sorting: Example

```
SELECT count(*) FROM deep_dive WHERE dt = '06-09-2017';
```

Unsorted table



Sorted by date



© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

If we take the same table and sort it by the date, the zone maps get to be in a more optimal condition for this SQL query and we can reduce I/O further

Best Practices: Sort Keys

Place the sort key on columns that are frequently filtered on placing the lowest cardinality columns first

- On most fact tables, the first sort key column should be a temporal column
- Columns added to a sort key after a high-cardinality column are not effective

With an established workload, use the following scripts to help find sort key suggestions:

https://github.com/awslabs/amazon-redshift-utils/blob/master/src/AdminScripts/filter_used.sql

https://github.com/awslabs/amazon-redshift-utils/blob/master/src/AdminScripts/predicate_columns.sql

Design considerations:

- Sort keys are less beneficial on small tables
- Define four or less sort key columns—more will result in marginal gains and increased ingestion overhead

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

The main point is that **sort keys are primarily making the zone maps more effective**. Sort keys are mostly put on timestamps.

Terminology and Concepts: Slices

A *slice* can be thought of like a virtual compute node

- Unit of data partitioning
- Parallel query processing

Facts about slices:

- Each compute node has either 2, 16, or 32 slices
- Table rows are distributed to slices
- A slice processes only its own data

Slice is how we get parallelism within a single node in a Redshift cluster, data is stored physically per slice

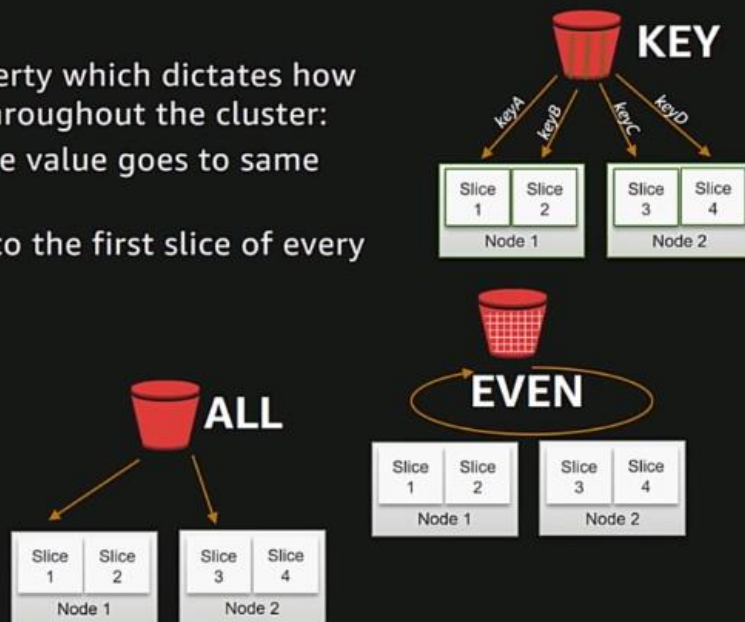
Data Distribution

Distribution style is a table property which dictates how that table's data is distributed throughout the cluster:

- **KEY:** value is hashed, same value goes to same location (slice)
- **ALL:** full table data goes to the first slice of every node
- **EVEN:** round robin

Goals:

- Distribute data evenly for parallel processing
- Minimize data movement during query processing



Data Distribution: Example

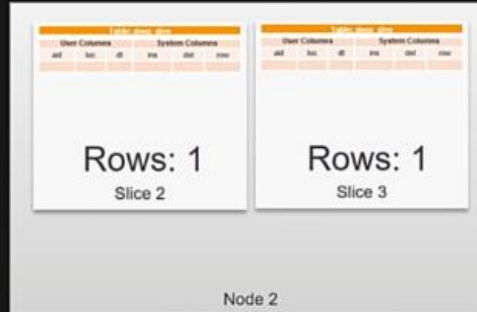
```
CREATE TABLE deep_dive (  
  aid INT      --audience_id  
  ,loc CHAR(3) --location  
  ,dt DATE     --date  
) DISTSTYLE (EVEN|KEY|ALL);
```



Data Distribution: **EVEN** Example

```
CREATE TABLE deep_dive (  
  aid INT      --audience_id  
  ,loc CHAR(3) --location  
  ,dt DATE     --date  
) DISTSTYLE EVEN;
```

```
INSERT INTO deep_dive VALUES  
(1, 'SFO', '2016-09-01'),  
(2, 'JFK', '2016-09-14'),  
(3, 'SFO', '2017-04-01'),  
(4, 'JFK', '2017-05-14');
```



Data Distribution: KEY Example #1

```
CREATE TABLE deep_dive (  
  aid INT      --audience_id  
  ,loc CHAR(3) --location  
  ,dt DATE     --date  
) DISTSTYLE KEY DISTKEY (loc);
```

```
INSERT INTO deep_dive VALUES  
(1, 'SFO', '2016-09-01'),  
(2, 'JFK', '2016-09-14'),  
(3, 'SFO', '2017-04-01'),  
(4, 'JFK', '2017-05-14');
```

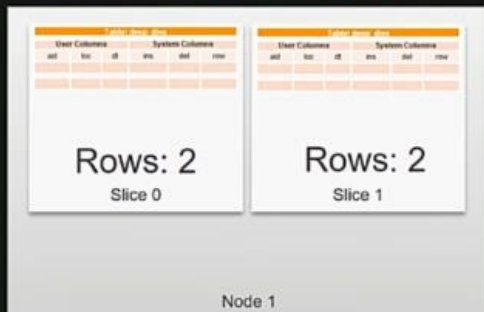


We are picking the location **loc** column to distribute the data to nodes by. The Node 0 Slices get the SFO and JFK keys data on the first run above and also on the 2nd run below

Data Distribution: KEY Example #1

```
CREATE TABLE deep_dive (  
  aid INT      --audience_id  
  ,loc CHAR(3) --location  
  ,dt DATE     --date  
) DISTSTYLE KEY DISTKEY (loc);
```

```
INSERT INTO deep_dive VALUES  
(1, 'SFO', '2016-09-01'),  
(2, 'JFK', '2016-09-14'),  
(3, 'SFO', '2017-04-01'),  
(4, 'JFK', '2017-05-14');
```



This is a bad distribution in Redshift because if you execute a query against data in your Redshift cluster, your second compute node does not have data in it to work on and this defeats the purpose of MPP and you have 1 node doing all the work.

Data Distribution: KEY Example #2

```
CREATE TABLE deep_dive (  
  aid INT      --audience_id  
  ,loc CHAR(3) --location  
  ,dt DATE     --date  
) DISTSTYLE KEY DISTKEY (aid);
```

```
INSERT INTO deep_dive VALUES  
(1, 'SFO', '2016-09-01'),  
(2, 'JFK', '2016-09-14'),  
(3, 'SFO', '2017-04-01'),  
(4, 'JFK', '2017-05-14');
```

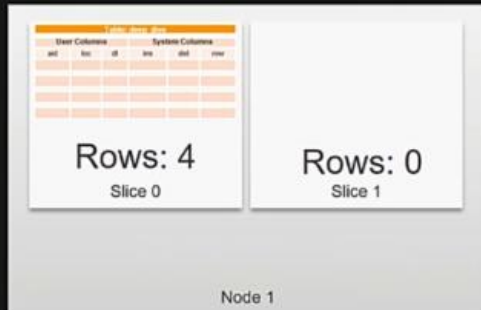


Let us now pick another column to distribute the data by like the aid key above. The hashing of the aid key will end up as above

Data Distribution: ALL Example

```
CREATE TABLE deep_dive (  
  aid INT      --audience_id  
  ,loc CHAR(3) --location  
  ,dt DATE     --date  
) DISTSTYLE ALL;
```

```
INSERT INTO deep_dive VALUES  
(1, 'SFO', '2016-09-01'),  
(2, 'JFK', '2016-09-14'),  
(3, 'SFO', '2017-04-01'),  
(4, 'JFK', '2017-05-14');
```



Best Practices: Data Distribution

DISTSTYLE **KEY**

- Goals
 - Optimize **JOIN** performance between large tables
 - Optimize **INSERT INTO SELECT** performance
 - Optimize **GROUP BY** performance
- The column that is being distributed on should have a high cardinality and not cause row skew:

```
SELECT diststyle, skew_rows FROM svv_table_info WHERE "table" = 'deep_dive';
diststyle | skew_rows
-----+-----
KEY(aid)  |      1.07
```

 ← Ratio between the slice with the most and least number of rows

DISTSTYLE **ALL**

- Goals
 - Optimize **Join** performance with dimension tables
 - Reduces disk usage on small tables
 - Small and medium size dimension tables (< 3M rows)

DISTSTYLE **EVEN**

- If neither KEY or ALL apply (or you are unsure)

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Best Practices: Table Design Summary

Materialize often filtered columns from dimension tables into fact tables

Materialize often calculated values into tables

Avoid distribution keys on temporal columns

Keep data types as wide as necessary (but no longer than necessary)

- VARCHAR, CHAR, and NUMERIC

Add compression to columns

- Optimal compression can be found using ANALYZE COMPRESSION

Add sort keys on the primary columns that are filtered on

Data Storage, Ingestion, and ELT

Terminology and Concepts: Disks

Amazon Redshift utilizes locally attached storage devices

- Compute nodes have 2½ to 3 times the advertised storage capacity

Each disk is split into two partitions

- Local data storage, accessed by local CN
- Mirrored data, accessed by remote CN

Partitions are raw devices

- Local storage devices are ephemeral in nature
- Tolerant to multiple disk failures on a single node

Terminology and Concepts: Redundancy

Global **commit** ensures all permanent tables have written blocks to another node in the cluster to ensure data redundancy

Asynchronously backup blocks to Amazon S3—always consistent snapshot

- Every 5 GB of changed data or eight hours
- User on-demand manual snapshots

Temporary tables

- Blocks are not mirrored to the remote partition—two-times faster write performance
- Do not trigger a full commit or backups

Disable backups at the table level:

```
CREATE TABLE example(id int) BACKUP NO;
```

Terminology and Concepts: Transactions

Amazon Redshift is a fully transactional, ACID compliant data warehouse

- Isolation level is *serializable*
- Two phase commits (local and global commit phases)

Cluster commit statistics:

https://github.com/awslabs/amazon-redshift-utils/blob/master/src/AdminScripts/commit_stats.sql

Design consideration:

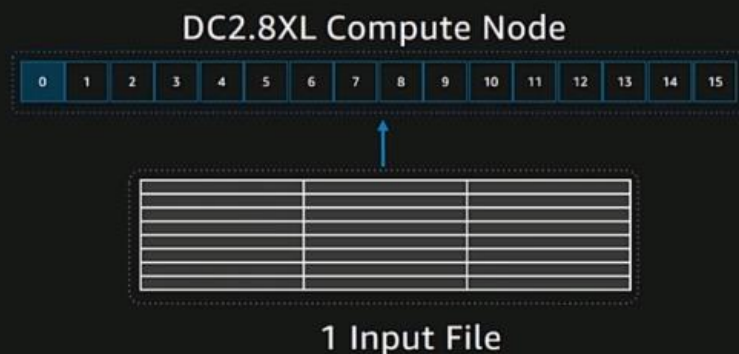
- Because of the expense of commit overhead, limit commits by explicitly creating transactions

Data Ingestion: COPY Statement

Ingestion throughput:

- Each slice's query processors can load one file at a time:
 - Streaming decompression
 - Parse
 - Distribute
 - Write

Realizing only partial node usage as 6.25% of slices are active



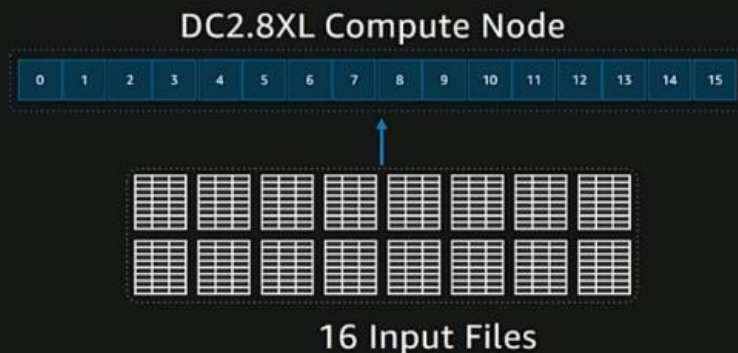
The COPY statement is the primary way to get your data into Redshift, it works mostly against loading data from S3.

Data Ingestion: COPY Statement

Number of input files should be a multiple of the number of slices

Splitting the single file into 16 input files, all slices are working to maximize ingestion performance

COPY continues to scale linearly as you add nodes



Recommendation is to use delimited files—1 MB to 1 GB after gzip compression

This is going to run 16 times faster.

Best Practices: COPY Ingestion

Delimited files are recommend

- Pick a simple delimiter '|' or ',' or tabs
- Pick a simple NULL character (\N)
- Use double quotes and an escape character (' \ ') for varchars
- UTF-8 varchar columns take four bytes per char

Split files so there is a multiple of the number of slices

Files sizes should be 1 MB–1 GB after gzip compression

Useful COPY options

- MAXERRORS
- ACCEPTINVCHARS
- NULL AS

Data Ingestion: Amazon Redshift Spectrum

Use INSERT INTO SELECT against external Amazon S3 tables

- Ingest additional file formats: Parquet, ORC, Grok
- Aggregate incoming data
- Select subset of columns and/or rows
- Manipulate incoming column data with SQL

Best practices:

- Save cluster resources for querying and reporting rather than on ELT
- Filtering/aggregating incoming data can improve performance over COPY

Design considerations:

- Repeated reads against Amazon S3 are not transactional
- \$5/TB of (compressed) data scanned

Spectrum allows you to create an external table against S3 so that you can query that external table. You can also do INSERT INTO SELECT statement against those external table and load those to Redshift.

Design Considerations: Data Ingestion

Designed for large writes

- Batch processing system, optimized for processing massive amounts of data
- 1 MB size plus immutable blocks means that we clone blocks on write so as not to introduce fragmentation
- Small write (~1-10 rows) has similar cost to a larger write (~100K rows)

UPDATE and DELETE

- Immutable blocks means that we only logically delete rows on UPDATE or DELETE
- Must VACUUM or DEEP COPY to remove ghost rows from table

Data Ingestion: Deduplication/UPSERT

Table: deep_dive

aid	loc	dt
1	SFO	2016-09-01
2	JFK	2016-09-14
3	SFO	2017-04-01
4	JFK	2017-05-14

Data Ingestion: Deduplication/UPSERT

Table: deep_dive

aid	loc	dt
1	SFO	2017-10-20
2	JFK	2017-10-20
3	SFO	2017-04-01
4	JFK	2017-05-14
5	SJC	2017-10-10
6	SEA	2017-11-29

s3://bucket/dd.csv

aid	loc	dt
1	SFO	2017-10-20
2	JFK	2017-10-20
5	SJC	2017-10-10
6	SEA	2017-11-29

How do we do this in Redshift?

Data Ingestion: Deduplication/UPSERT

Steps:

1. Load CSV data into a staging table
2. Delete duplicate data from the production table
3. Insert (or append) data from the staging into the production table

Data Ingestion: Deduplication/UPSERT

BEGIN;

We first create a transaction

Data Ingestion: Deduplication/UPSERT

BEGIN;

```
CREATE TEMP TABLE staging(LIKE deep_dive);
```

Then create the staging table as a temporary table, then use LIKE to make a copy of the deep_dive distribution table into the staging table as well as get the compression chosen in the prod distribution table

Data Ingestion: Deduplication/UPSERT

BEGIN;

```
CREATE TEMP TABLE staging(LIKE deep_dive);
```

```
COPY staging FROM 's3://bucket/dd.csv'  
: 'creds' COMPUPDATE OFF;
```

We then load the data in from the S3 file

Data Ingestion: Deduplication/UPSERT

BEGIN;

```
CREATE TEMP TABLE staging(LIKE deep_dive);
```

```
COPY staging FROM 's3://bucket/dd.csv'  
: 'creds' COMPUPDATE OFF;
```

```
DELETE deep_dive d
```

```
USING staging s WHERE d.aid = s.aid;
```

```
INSERT INTO deep_dive SELECT * FROM staging;
```

```
DROP TABLE staging;
```

COMMIT;

This is the best way to do an UPSERT

Best Practices: ELT

Wrap workflow/statements in an explicit transaction

Consider using DROP TABLE or TRUNCATE instead of DELETE

Staging Tables:

- Use temporary table or permanent table with the "BACKUP NO" option
- If possible use DISTSTYLE KEY on both the staging table and production table to speed up the INSERT INTO SELECT statement
- Turn off automatic compression—COMPUPDATE OFF
- Copy compression settings from production table or use ANALYZE COMPRESSION statement
 - Use CREATE TABLE LIKE or write encodings into the DDL
- For copying a large number of rows (> hundreds of millions) consider using ALTER TABLE APPEND instead of INSERT INTO SELECT

Vacuum and Analyze

VACUUM will globally sort the table and remove rows that are marked as deleted

- For tables with a sort key, ingestion operations will locally sort new data and write it into the unsorted region

ANALYZE collects table statistics for optimal query planning

Best Practices:

- VACUUM should be run only as necessary
 - Typically nightly or weekly
 - Consider **Deep Copy** (recreating and copying data) for larger or wide tables
- ANALYZE can be run periodically after ingestion on just the columns that WHERE predicates are filtered on
- Utility to VACUUM and ANALYZE all the tables in the cluster:
<https://github.com/aws-labs/amazon-redshift-utils/tree/master/src/AnalyzeVacuumUtility>

Node Types and Cluster Sizing

Terminology and Concepts: Node Types

Dense Compute—DC2

- Solid state disks

Dense Storage—DS2

- Magnetic disks

Instance Type	Disk Type	Size	Memory	CPUs
DC2 large	NVMe SSD	160 GB	16 GB	2
DC2 8xlarge	NVMe SSD	2.56 TB	244 GB	32
DS2 xlarge	Magnetic	2 TB	32 GB	4
DS2 8xlarge	Magnetic	16 TB	244 GB	36

Best Practices: Cluster Sizing

Use at least two compute nodes (multi-node cluster) in production for data mirroring

- Leader node is given for no additional cost

Amazon Redshift is significantly faster in a VPC compared to EC2 Classic

Maintain at least 20% free space or three times the size of the largest table

- Scratch space for usage, rewriting tables
- Free space is required for vacuum to re-sort table
- Temporary tables used for intermediate query results

The maximum number of available Amazon Redshift Spectrum nodes is a function of the number of slices in the Amazon Redshift cluster

If you're using DC1 instances, upgrade to the DC2 instance type

- Same price as DC1, significantly faster
- Reserved Instances do not automatically transfer over

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

You can run a single-node cluster in Redshift for things like PoC, playing around, Dev or QA work, etc. Use a multi-node cluster for your production workloads, you get an extra node for the Leader node in this case.

Additional Resources

AWS Labs on GitHub—Amazon Redshift

<https://github.com/awslabs/amazon-redshift-utils>

<https://github.com/awslabs/amazon-redshift-monitoring>

<https://github.com/awslabs/amazon-redshift-udfs>

Admin Scripts

Collection of utilities for running diagnostics on your cluster

Admin Views

Collection of utilities for managing your cluster, generating schema DDL, and so on

Analyze Vacuum Utility

Utility that can be scheduled to vacuum and analyze the tables within your Amazon Redshift cluster

Column Encoding Utility

Utility that will apply optimal column encoding to an established schema with data already loaded

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

AWS Big Data Blog—Amazon Redshift

Amazon Redshift Engineering's Advanced Table Design Playbook

<https://aws.amazon.com/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-preamble-prerequisites-and-prioritization/>

- Zach Christopherson

Top 10 Performance Tuning Techniques for Amazon Redshift

<https://aws.amazon.com/blogs/big-data/top-10-performance-tuning-techniques-for-amazon-redshift/>

- Ian Meyers and Zach Christopherson

10 Best Practices for Amazon Redshift Spectrum

<https://aws.amazon.com/blogs/big-data/10-best-practices-for-amazon-redshift-spectrum/>

- Po Hong and Peter Dalton



AWS
re:Invent

Thank you!

Tony Gibbs

AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

