Best Practices with Amazon Redshift

AWS Webinar

Tony Gibbs, Sr. Data Warehouse Solutions Architect

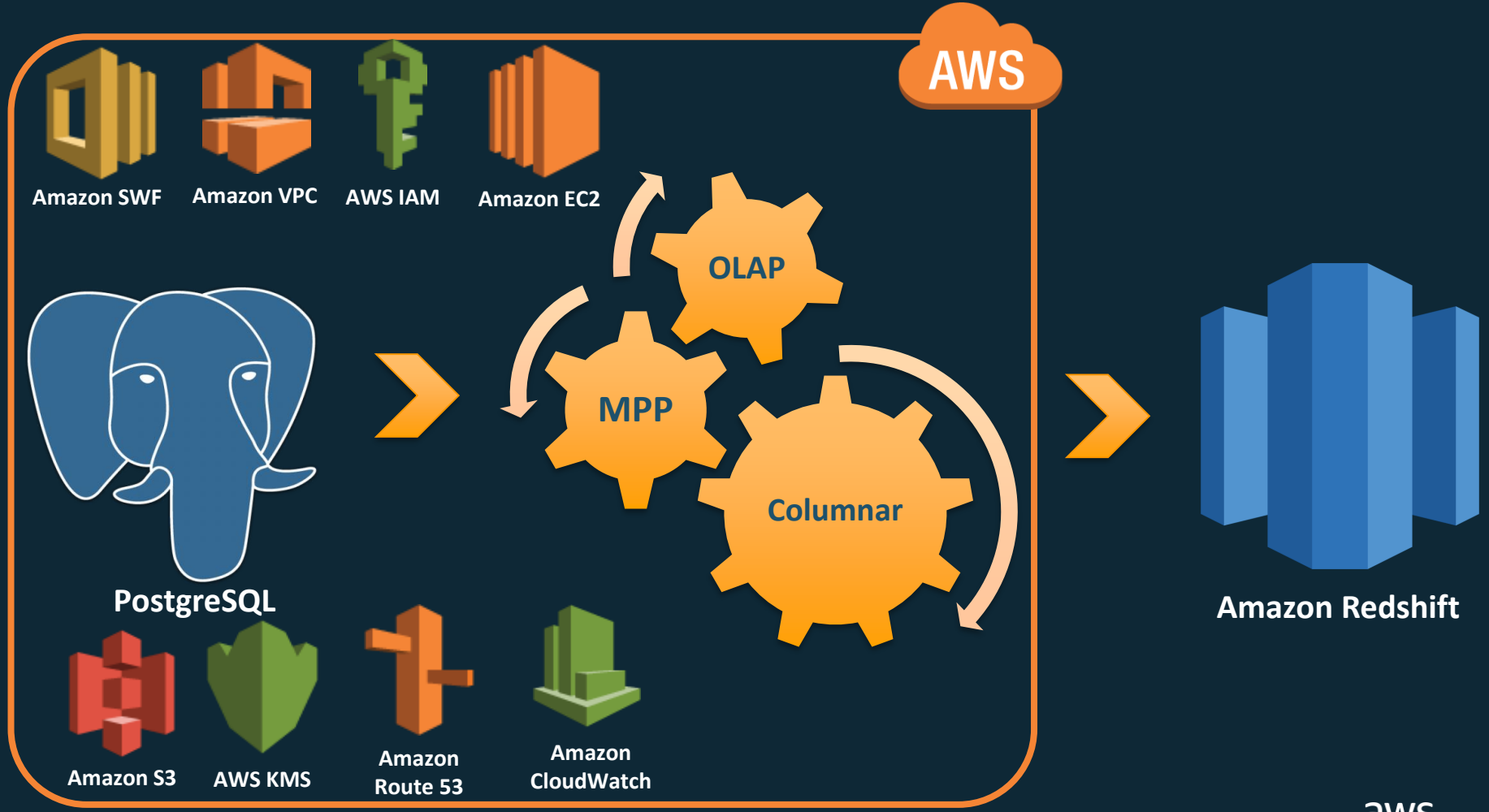
August 23, 2018

Amazon Redshift Best Practices Overview

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

aws

History and Development



February 2013

> 130 Significant Patches

> 180 Significant Features

August 2018

Concepts and Table Design

Amazon Redshift Architecture

Massively parallel, shared nothing columnar architecture

Leader node

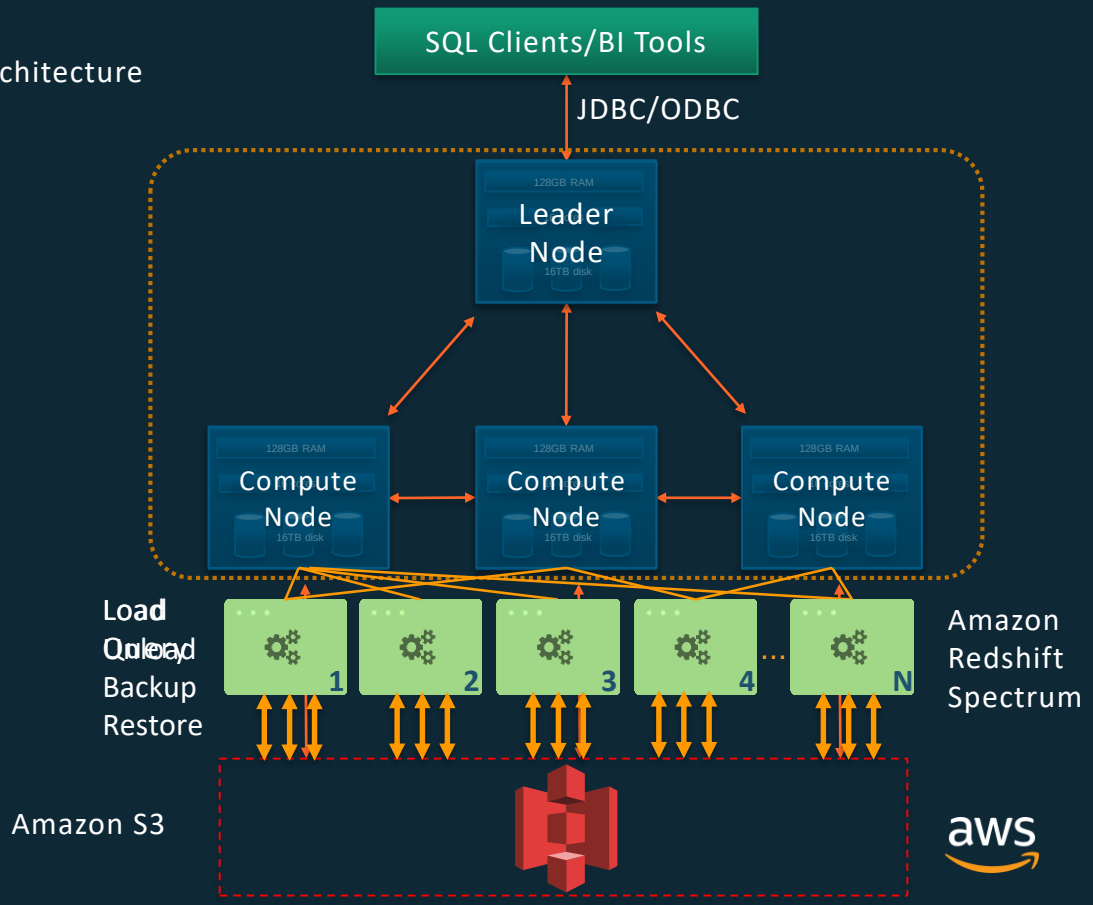
- SQL endpoint
- Stores metadata
- Coordinates parallel SQL processing

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)



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

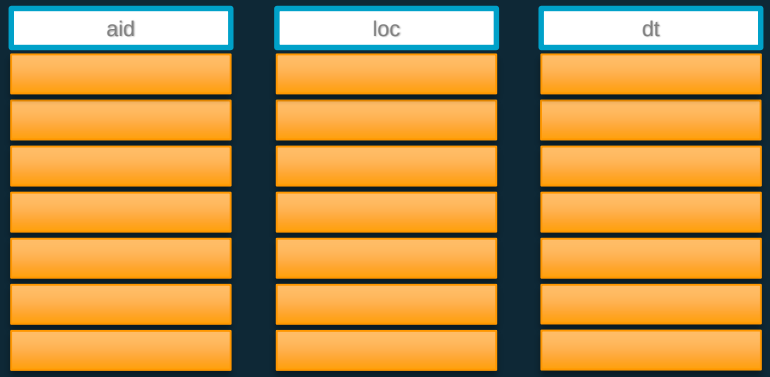
Only read the column data that is required

aws

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




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

Row-based storage

- Need to read everything
- Unnecessary I/O

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


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

Column-based storage

- Only scan blocks for relevant column

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

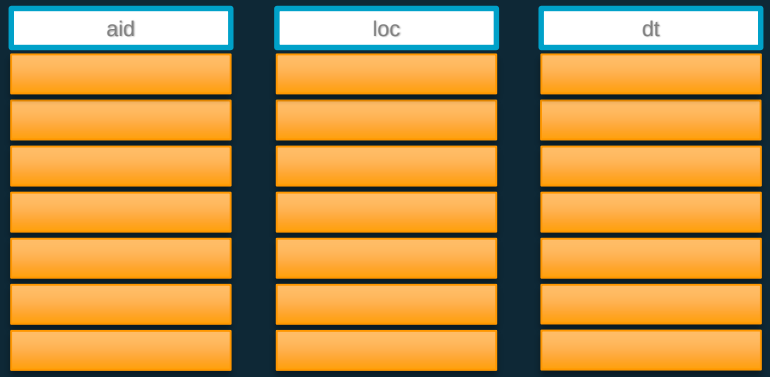
By default, COPY automatically analyzes and compresses data on first load into an empty table

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

Compression: 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

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

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
```

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

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 a SORTKEY defined on one or more columns

Optimal sort key is dependent on:

- Query patterns
- Business requirements
- Data profile

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

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

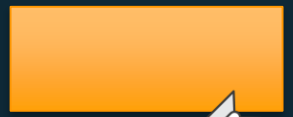
Zone Maps and Sorting: Example

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

Unsorted table

MIN: 01-JUNE-2017
MAX: 20-JUNE-2017

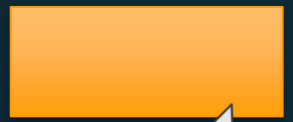
MIN: 08-JUNE-2017
MAX: 30-JUNE-2017



MIN: 12-JUNE-2017
MAX: 20-JUNE-2017

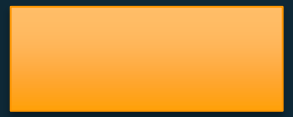
MIN: 02-JUNE-2017
MAX: 25-JUNE-2017

Sorted by date

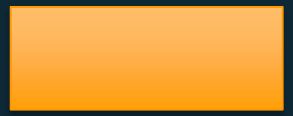


MIN: 01-JUNE-2017
MAX: 06-JUNE-2017

MIN: 07-JUNE-2017
MAX: 12-JUNE-2017



MIN: 13-JUNE-2017
MAX: 21-JUNE-2017



MIN: 21-JUNE-2017
MAX: 30-JUNE-2017

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

aws

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:

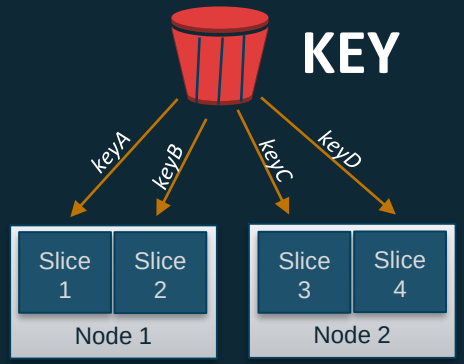
- Each compute node has either 2, 16, or 32 slices
- Table rows are distributed to slices
- A slice processes only its own data

aws

Data Distribution

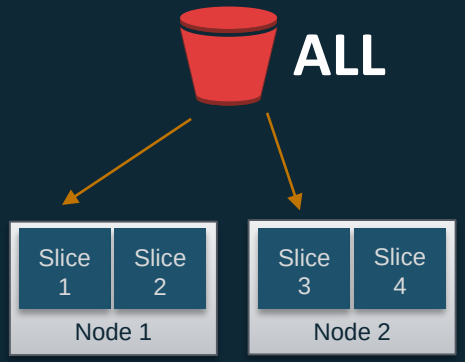
Distribution style is a table property which dictates how that table's data is distributed throughout the cluster:

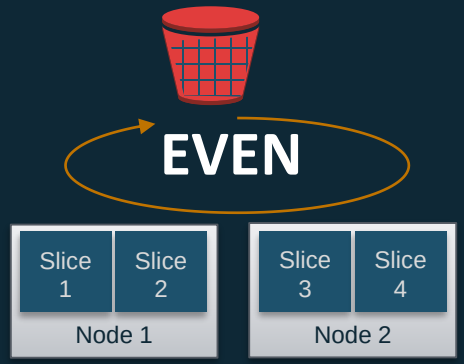
- KEY: value is hashed, same value goes to same location (slice)
- ALL: full table data goes to the first slice of every node
- EVEN: round robin



Goals:

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





Data Distribution: Example

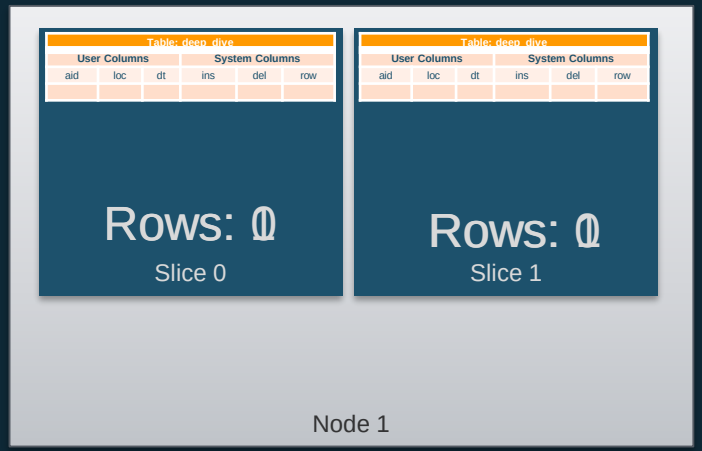
```
CREATE TABLE deep_dive (
    aid   INT       --audience_id
    ,loc  CHAR(3)   --location
    ,dt   DATE      --date
) DISTSTYLE EVEN|KEY|ALL;
```

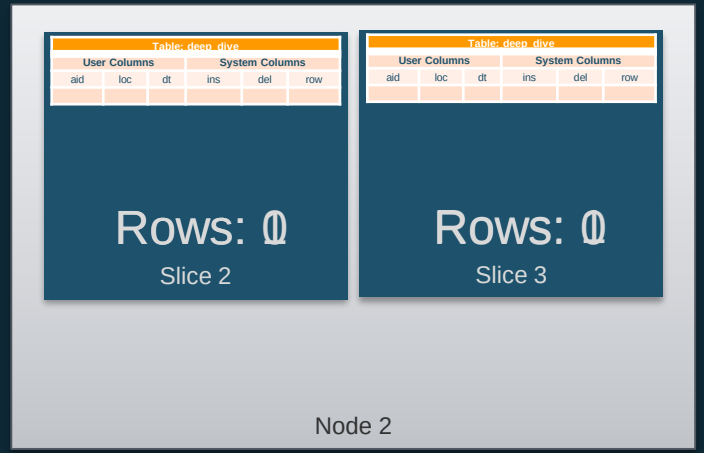
Table: deep_dive					
User Columns			System Columns		
aid	loc	dt	ins	del	row

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');
```

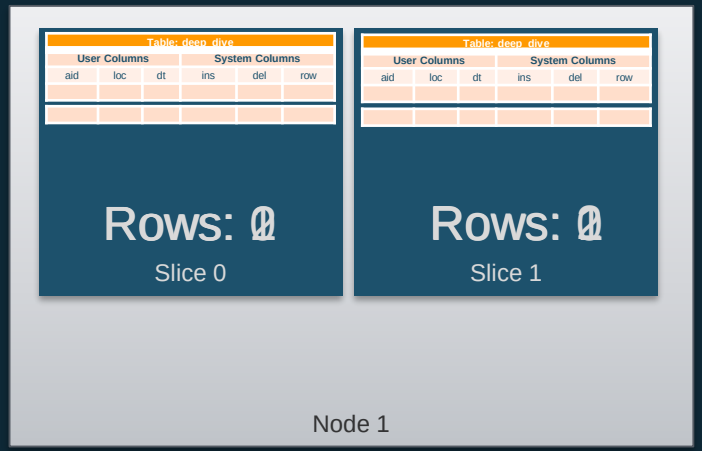


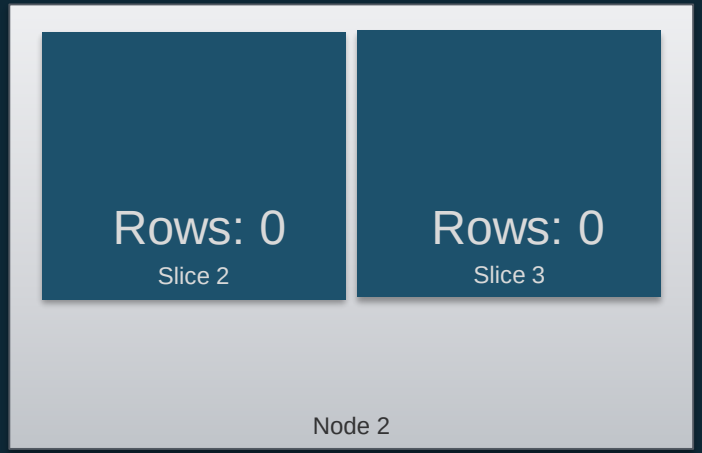


Data Distribution: KEY Example #1

```
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');
```

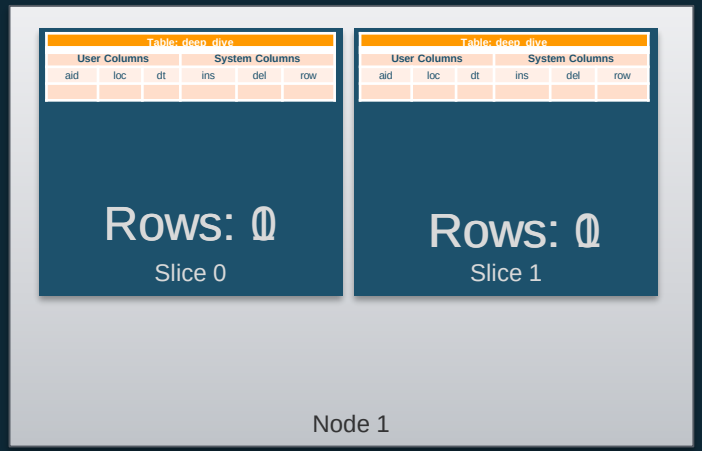


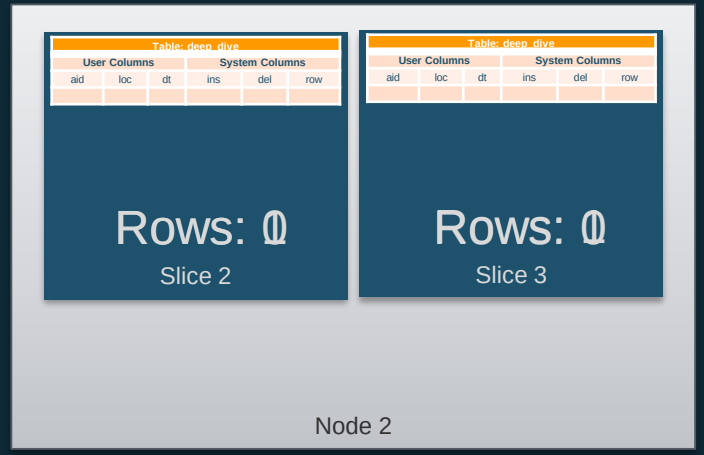


Data Distribution: KEY Example #2

```
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');
```

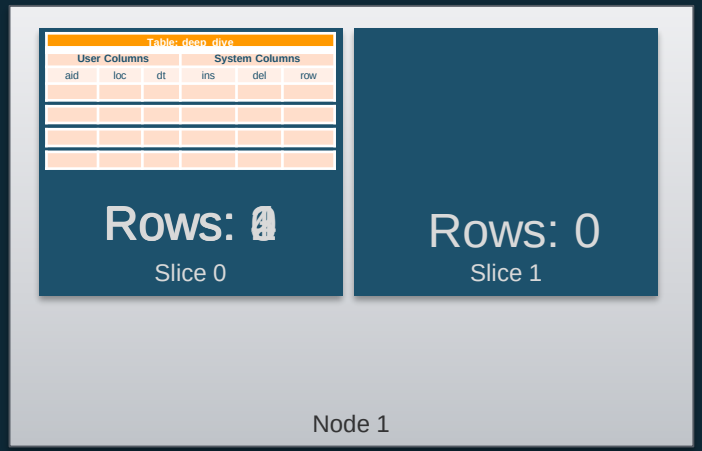


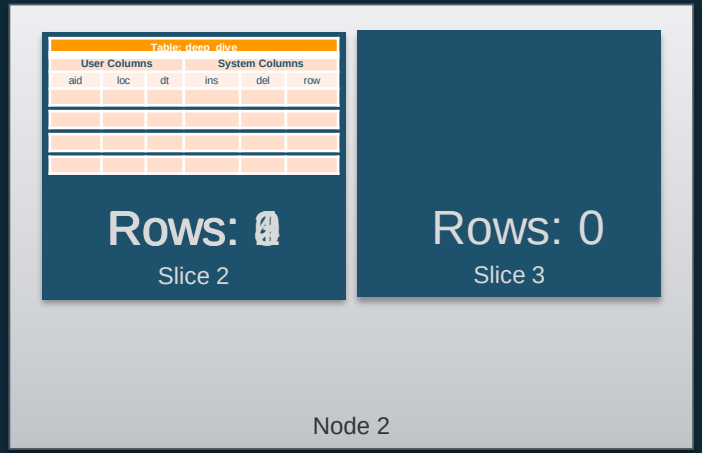


Data Distribution: ALL Example

```
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)

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

aws

Data Storage, Ingestion, and ELT

Terminology and Concepts: Disks

Amazon Redshift utilizes locally attached storage devices

- Compute nodes have 2 ½ 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:

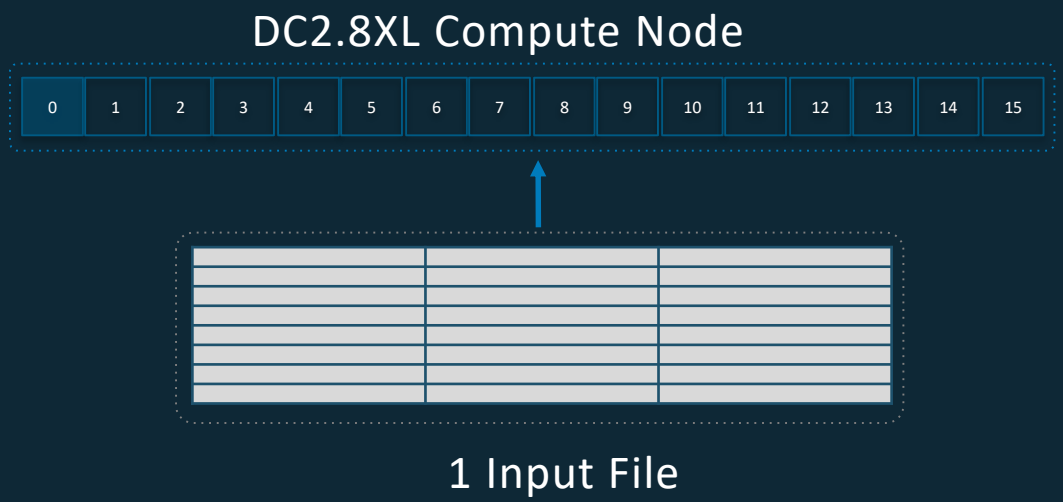
- Because of the expense of commit overhead, limit commits by explicitly creating transactions

aws

Data Ingestion: COPY Statement

Ingestion throughput:

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



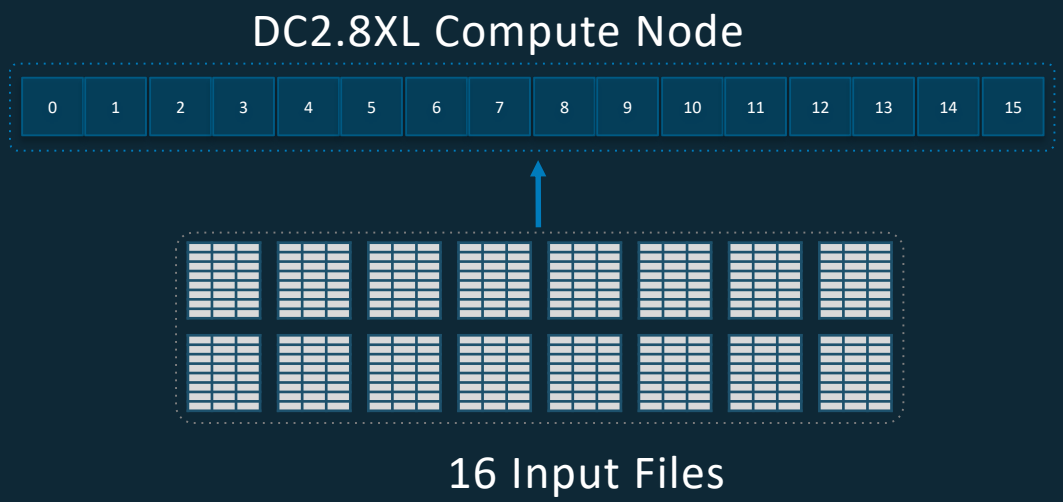
Realizing only partial node usage as 6.25% of slices are active

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

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

aws

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

aws

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

aws

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

aws

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

aws

Data Ingestion: Deduplication/UPSERT

```
BEGIN;
CREATE TEMP TABLE staging(LIKE deep_dive);
COPY staging FROM 's3://bucket/dd.csv'
:'creds' COMPUPDATE OFF;
DELETE FROM deep_dive d
USING staging s WHERE d.aid = s.aid;
INSERT INTO deep_dive SELECT * FROM staging;
DROP TABLE staging;
COMMIT;
```

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

aws

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/awslabs/amazon-redshift-utils/tree/master/src/AnalyzeVacuumUtility

Workload Management and Query Monitoring Rules

Workload Management (WLM)

Allows for the separation of different query workloads

Goals

- Prioritize important queries
- Throttle/abort less important queries

Control concurrent number of executing of queries

Divide cluster memory

Set query timeouts to abort long running queries

aws

Terminology and Concepts: WLM

Queues

- Assigned a percentage of cluster memory
- SQL queries execute in queue based on
  - User group: which groups the user belongs to
  - Query group session level variable

Slots

- Division of memory within a WLM Queue, correlated with the number of simultaneous running queries
- WLM_QUERY_SLOT_COUNT is a session level variable
  - Useful to increase for memory intensive operations

aws

Workload Management: Example

Use case:

- Light ingestion/ELT on a continuous cadence of 10 minutes
- Peak reporting workload during business hours (7 a.m.–7 p.m.)
- Heavy ingestion/ELT nightly (11 p.m.–3 a.m.)

User types:

- Business reporting and dashboards
- Analysts and data science teams
- Database administrators

Workload Management: Example

Create a queue for each workload type:

Queue Name	Memory	Slots/Concurrency	Timeout (seconds)
Ingestion	20%	2	None
Dashboard	50%	10	120
Default (Analysts)	25%	3	None

- Enable: Short Query Acceleration (SQA)
- Unallocated memory goes into a general pool that can be used by any queue
- Hidden superuser queue can be used by admins, manually switched into:
  SET query_group TO 'superuser'
- The superuser queue has a single slot, the equivalent of 7-9% memory allocation, and no timeout

aws

Query Monitoring Rules (QMR)

Extension of workload management (WLM)

Allow the automatic handling of runaway (poorly written) queries

Rules applied to a WLM queue allow queries to be:

- LOGGED
- ABORTED
- HOPPED

Goals

- Protect against wasteful use of the cluster
- Log resource-intensive queries

aws

Query Monitoring Rules (QMR)

Metrics with operators and values (e.g. return_row_count > 10000000) create a predicate

Multiple predicates can be AND-ed together to create a rule

Multiple rules can be defined for a queue in WLM. These rules are OR-ed together

If *{ rule }* then *[action]*

{ rule : metric operator value } e.g.: rows_scanned > 1000000

Metric: cpu_time, query_blocks_read, rows scanned, query execution time, cpu & io skew per slice, join_row_count, etc.

Operator: <, >, ==

Value: integer

[action]: hop, log, abort

Best Practices: WLM and QMR

Keep the number of WLM queues to a minimum, typically just three queues to avoid having unused queues

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

Use WLM to limit ingestion/ELT concurrency to two to three

To maximize query throughput, use WLM to throttle the number of concurrent queries to 15 or less and enable SQA

Use QMR rather than WLM to set query timeouts

Use QMR to log long running queries

Save the superuser queue for administration tasks and canceling queries

aws

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 computes 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

aws

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

aws

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

Thank You!

Tony Gibbs