Deep Dive on Amazon Redshift

Storage Subsystem and Query Life Cycle

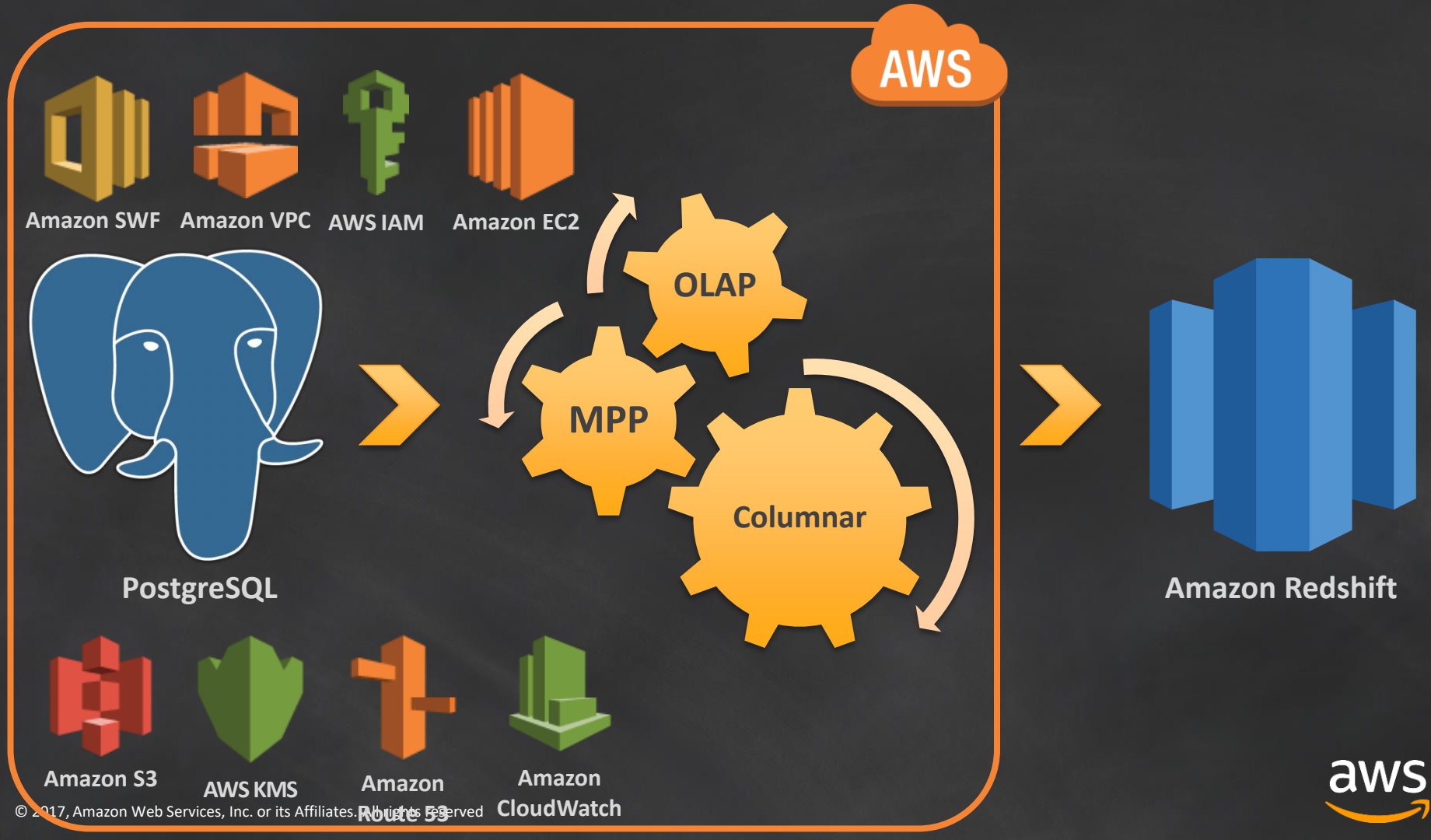
Darin Briskman
Databases, Analytics, and AI
briskman@amazon.com

Pop-up Loft

October 2017

Deep Dive Overview

- Amazon Redshift History and Development
- Cluster Architecture
- Concepts and Terminology
- Storage Deep Dive
- Query Life Cycle
- Data Ingestion Best Practices
- Recently Released Features
- Additional Resources
- Open Q&A



February 2013

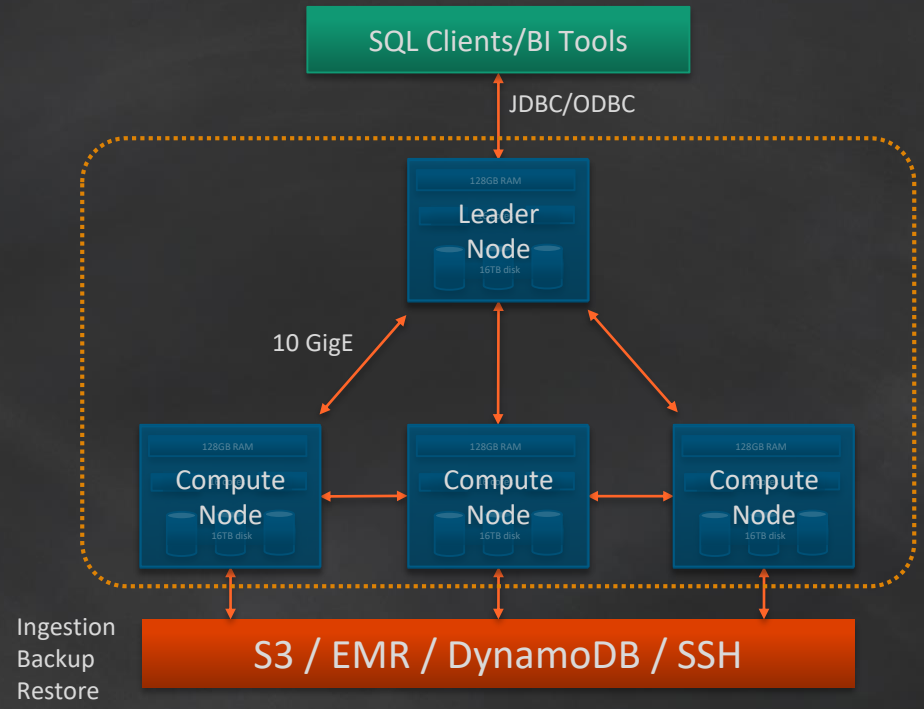
> 100 Significant Patches

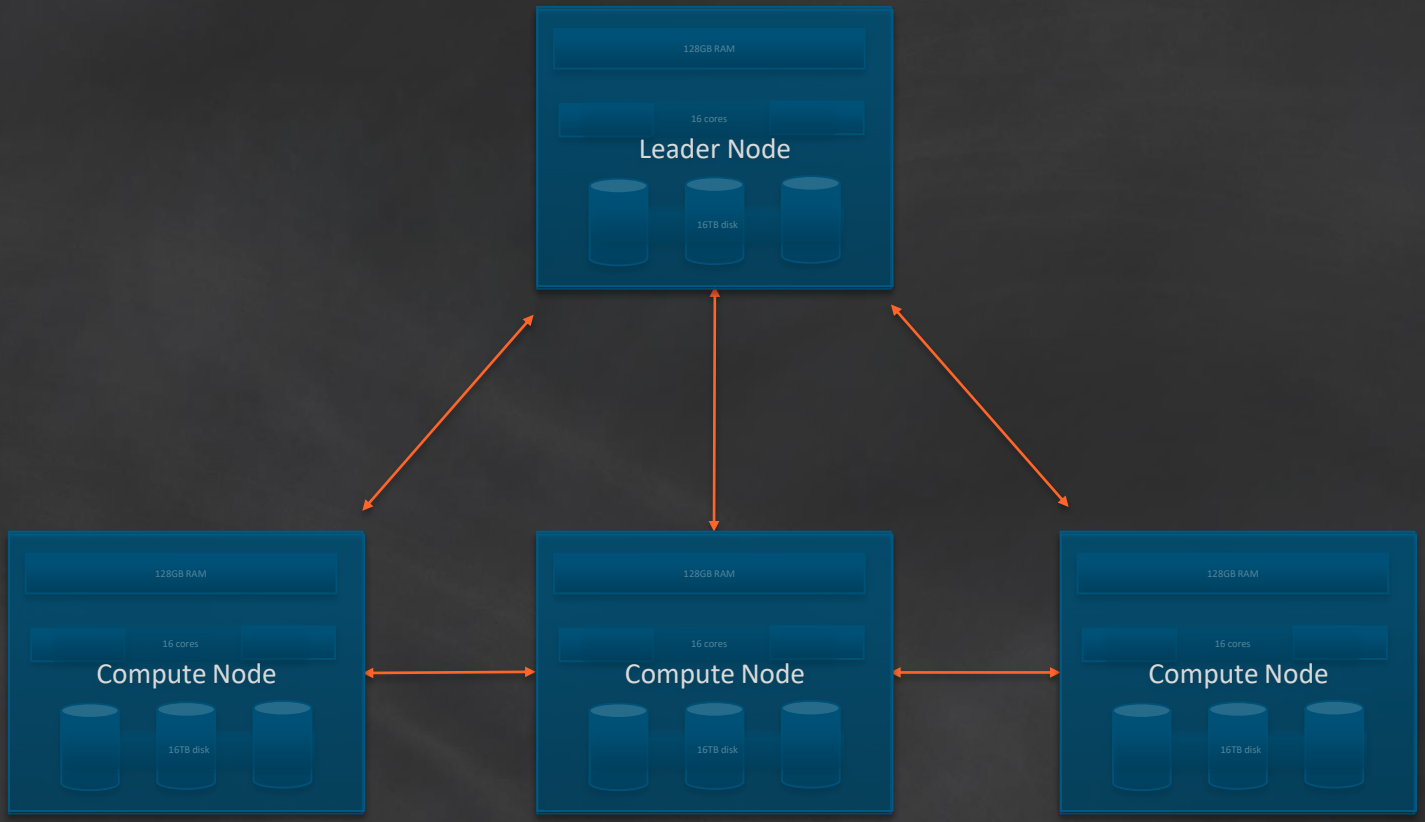
> 150 Significant Features

October 2017

Redshift Cluster Architecture

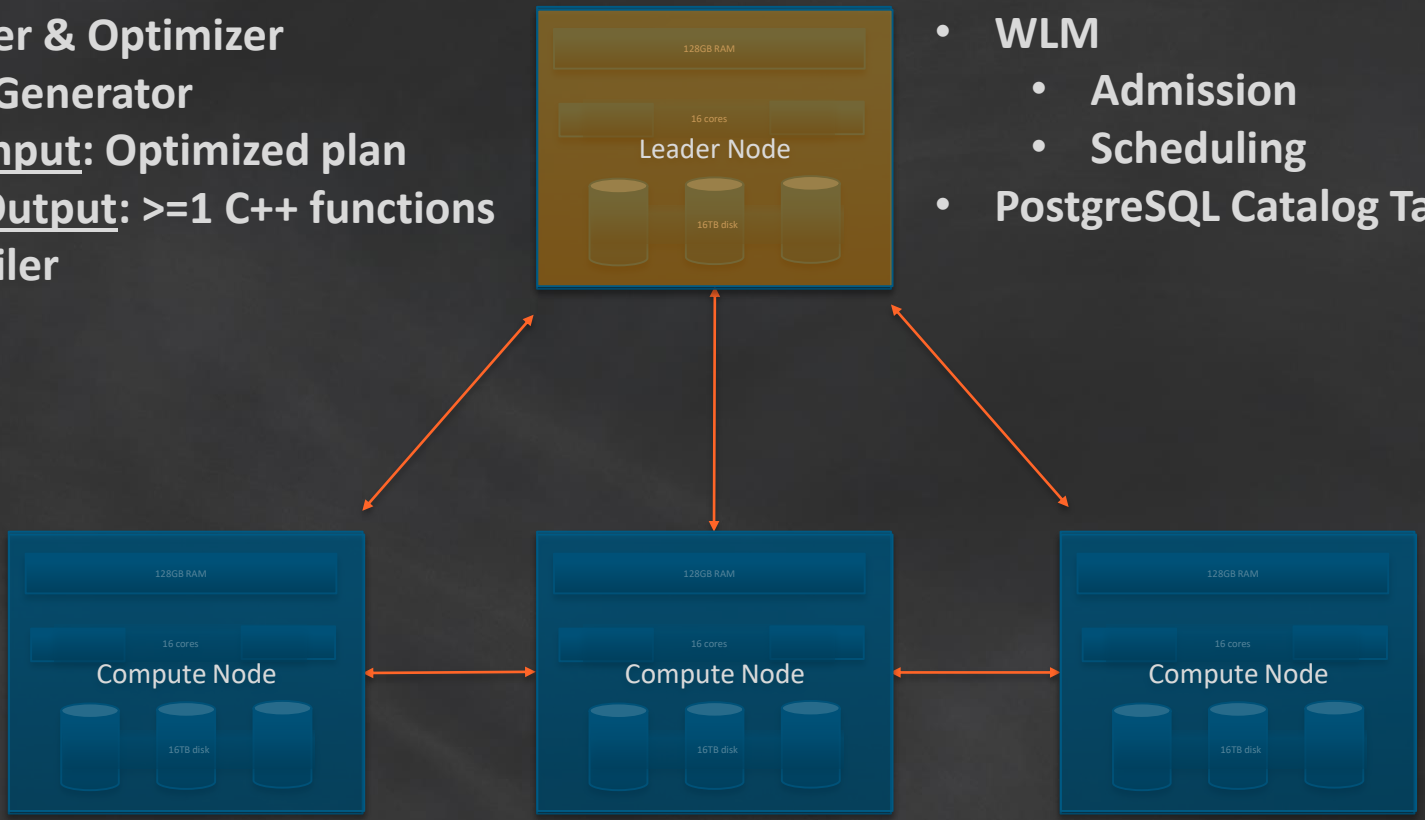
- **Massively parallel, shared nothing**
- Leader node
  - SQL endpoint
  - Stores metadata
  - Coordinates parallel SQL processing
- Compute nodes
  - Local, columnar storage
  - Executes queries in parallel
  - Load, backup, restore



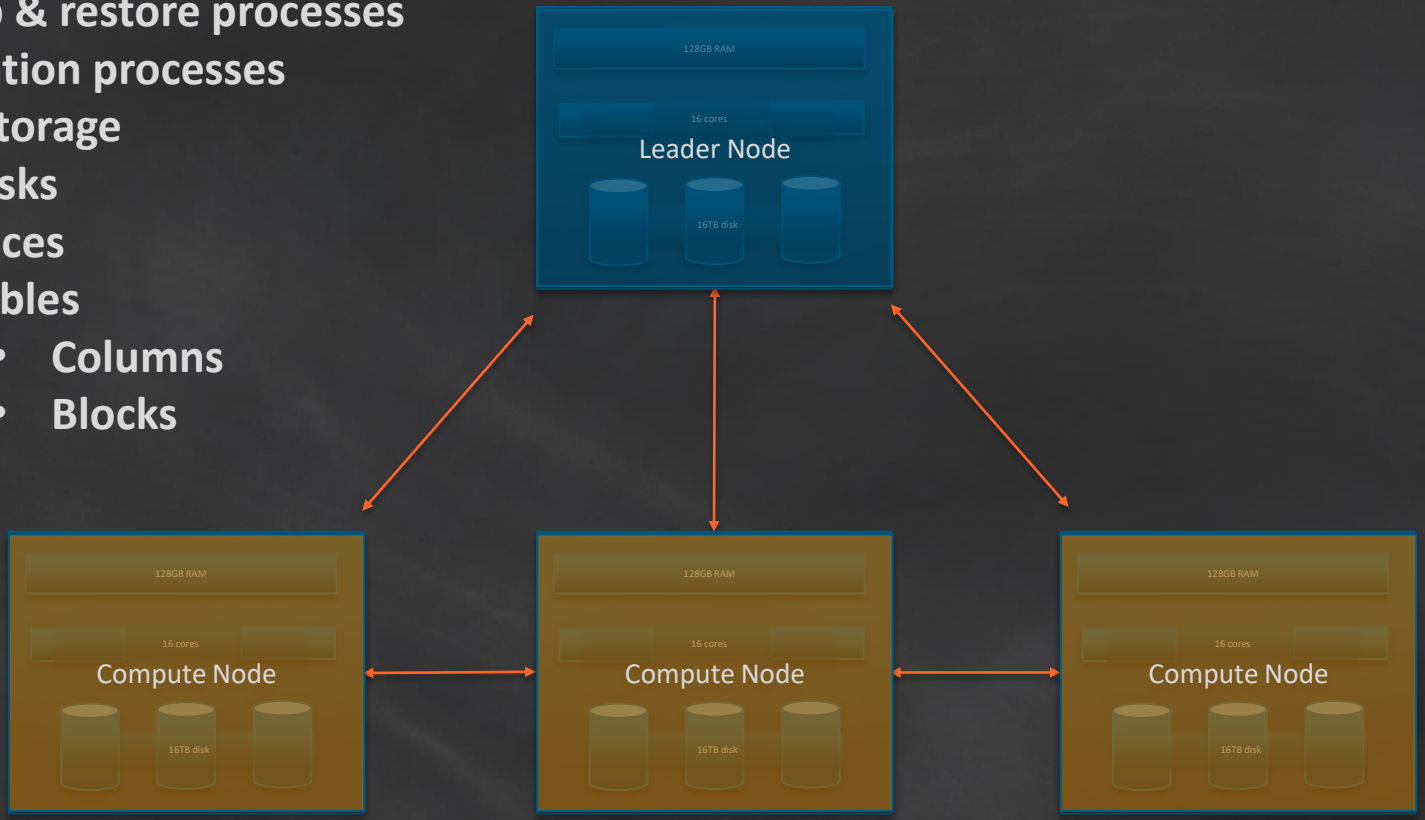


- Parser & Rewriter
- Planner & Optimizer
- Code Generator
  - Input: Optimized plan
  - Output: >=1 C++ functions
- Compiler

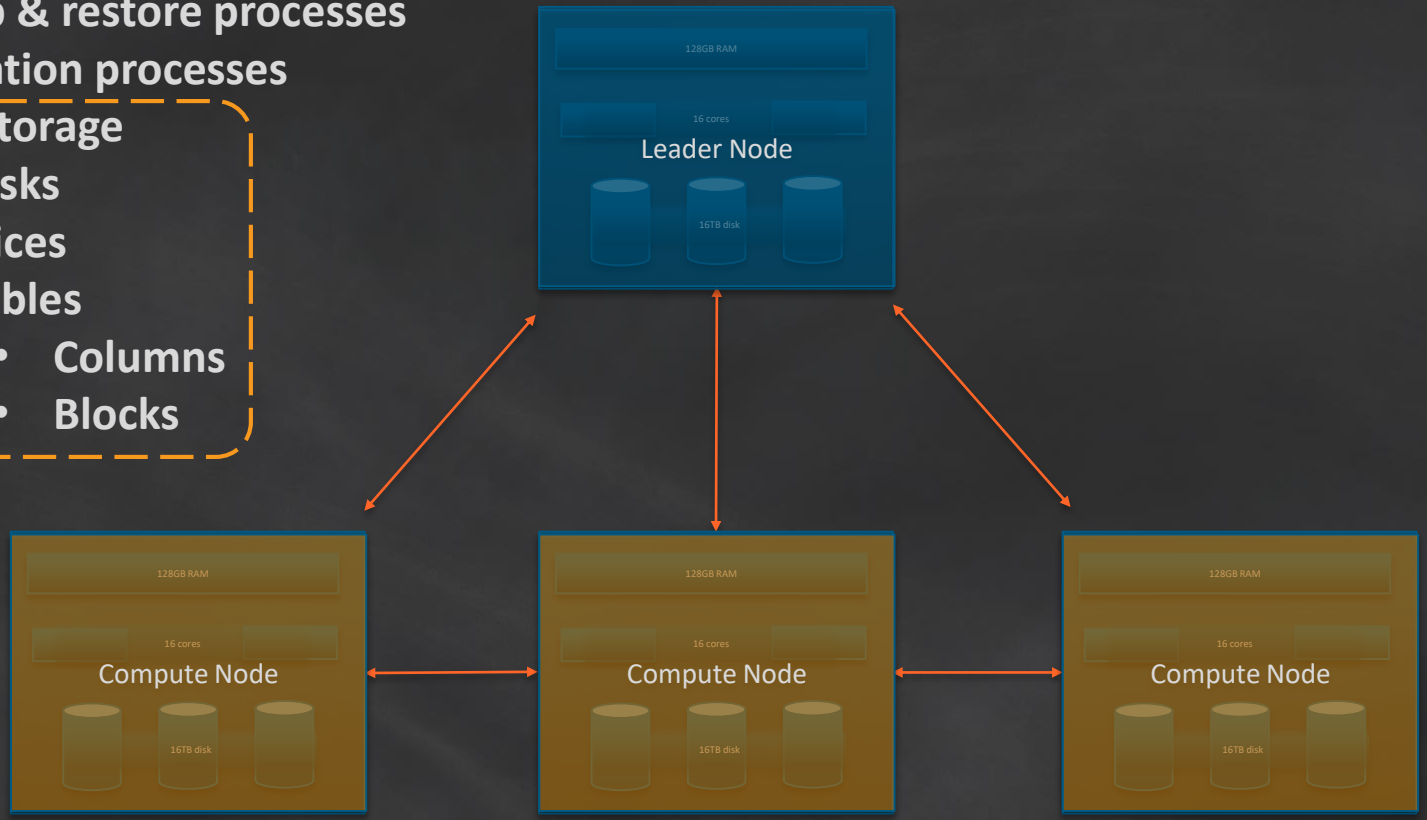
- Task Scheduler
- WLM
  - Admission
  - Scheduling
- PostgreSQL Catalog Tables



- Query execution processes
- Backup & restore processes
- Replication processes
- Local Storage
  - Disks
  - Slices
  - Tables
    - Columns
    - Blocks



- Query execution processes
- Backup & restore processes
- Replication processes
- Local Storage
  - Disks
  - Slices
  - Tables
    - Columns
    - Blocks



Designed for I/O Reduction

- Columnar storage
- Data compression
- Zone maps


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

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

- Accessing dt with row storage:
  - Need to read everything
  - Unnecessary I/O

Designed for I/O Reduction

- Columnar storage
- Data compression
- Zone maps


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

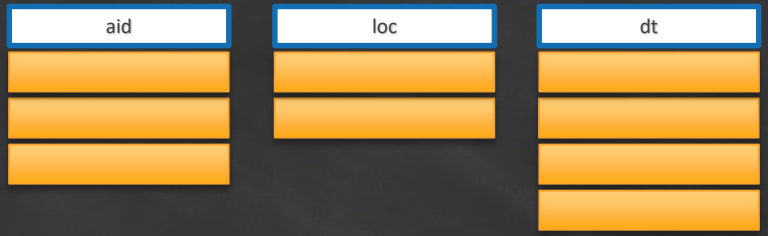
aid	loc	dt
1	SFO	2016-09-01
2	JFK	2016-09-14
3	SFO	2017-04-01
4	JFK	2017-05-14

- Accessing dt with columnar storage:
  - Only scan blocks for relevant column

aws

Designed for I/O Reduction

- Columnar storage
- Data compression
- Zone maps



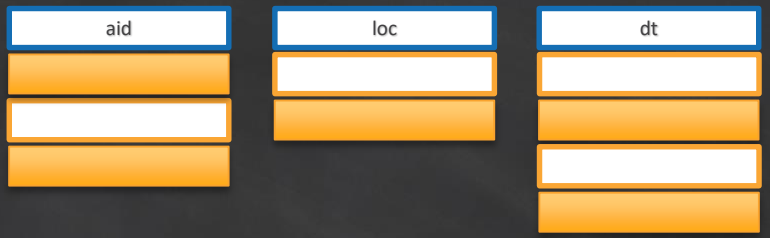

```
CREATE TABLE deep_dive (
    aid    INT         ENCODE LZO
    ,loc   CHAR(3)     ENCODE BYTEDICT
    ,dt    DATE        ENCODE RUNLENGTH
);
```

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

- Columns grow and shrink independently
- Effective compression ratios due to like data
- Reduces storage requirements
- Reduces I/O

Designed for I/O Reduction

- Columnar storage
- Data compression
- Zone maps




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

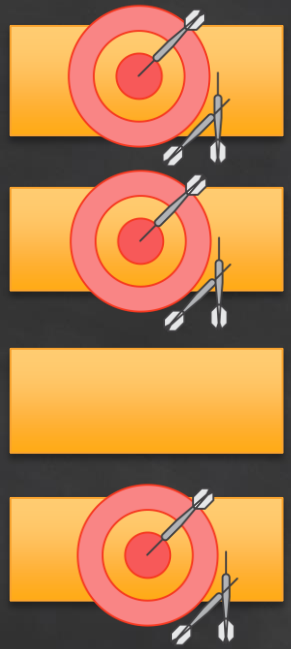
aid	loc	dt
1	SFO	2016-09-01
2	JFK	2016-09-14
3	SFO	2017-04-01
4	JFK	2017-05-14

- In-memory block metadata
- Contains per-block MIN and MAX value
- Effectively prunes blocks which cannot contain data for a given query
- Eliminates unnecessary I/O

Zone Maps and Sorting

```
SELECT COUNT(*) FROM deep_dive WHERE dt = '06-09-2013'
```

Unsorted Table



MIN: 01-JUNE-2013
MAX: 20-JUNE-2013

MIN: 08-JUNE-2013
MAX: 30-JUNE-2013

MIN: 12-JUNE-2013
MAX: 20-JUNE-2013

MIN: 02-JUNE-2013
MAX: 25-JUNE-2013

Sorted By Date

MIN: 01-JUNE-2013
MAX: 06-JUNE-2013

MIN: 07-JUNE-2013
MAX: 12-JUNE-2013

MIN: 13-JUNE-2013
MAX: 18-JUNE-2013

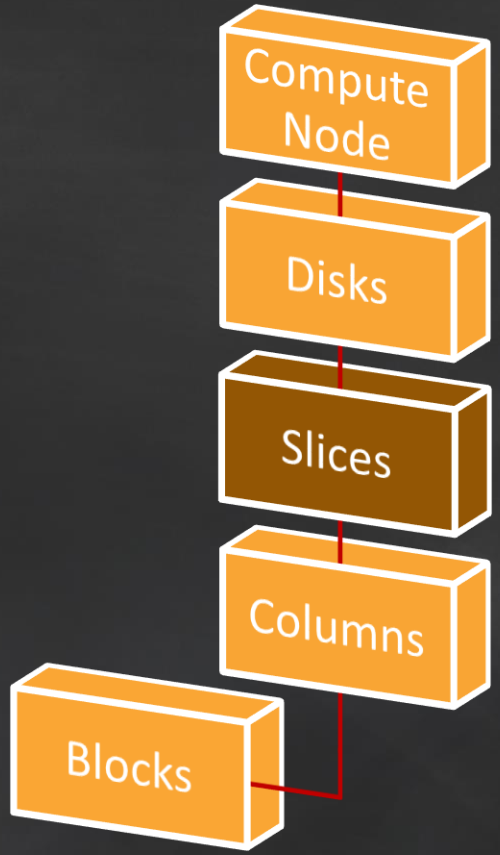
MIN: 19-JUNE-2013
MAX: 24-JUNE-2013

Terminology and Concepts: Data Sorting

- Goal:
  - Make queries run faster by optimizing the effectiveness of zone maps
    - Typically on the columns that are filtered on (where clause predicates)
- Impact:
  - Enables range restricted scans to prune blocks by leveraging zone maps
  - Overall reduction in block I/O
- Achieved with the table property SORTKEY defined on one or more columns
- Optimal SORTKEY is dependent on:
  - Query patterns
  - Data profile
  - Business requirements

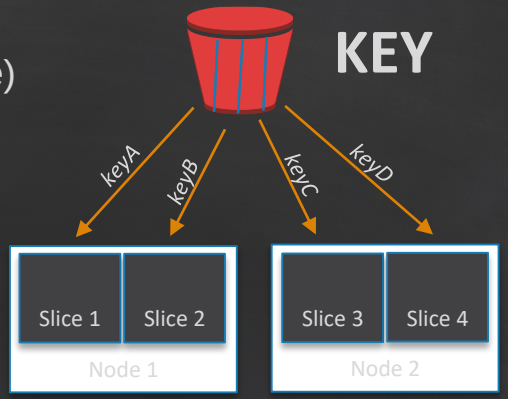
Terminology and Concepts: Slices

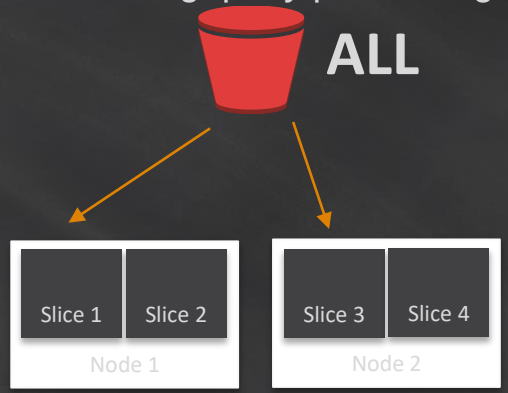
- A **slice** can be thought of like a “virtual compute node”
  - Unit of data partitioning
  - Parallel query processing
- Facts about slices:
  - Each compute node has either 2, 16, or 32 slices
  - Table rows are distributed to slices
  - A slice processes only its own data



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 first slice of every node
  - EVEN: Round robin
- Goals:
  - Distribute data evenly for parallel processing
  - Minimize data movement during query processing

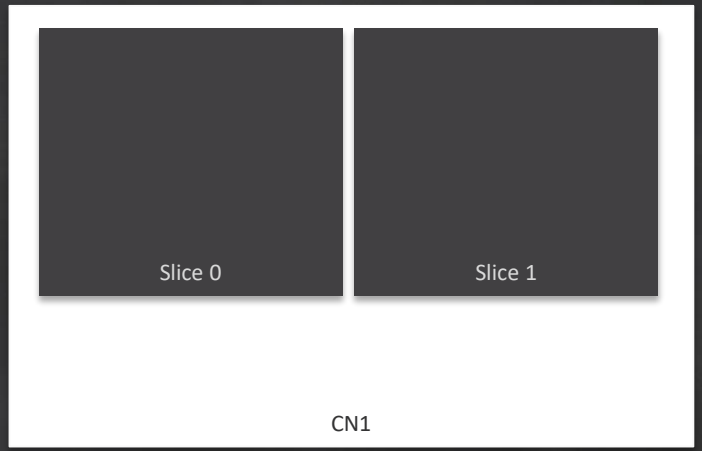


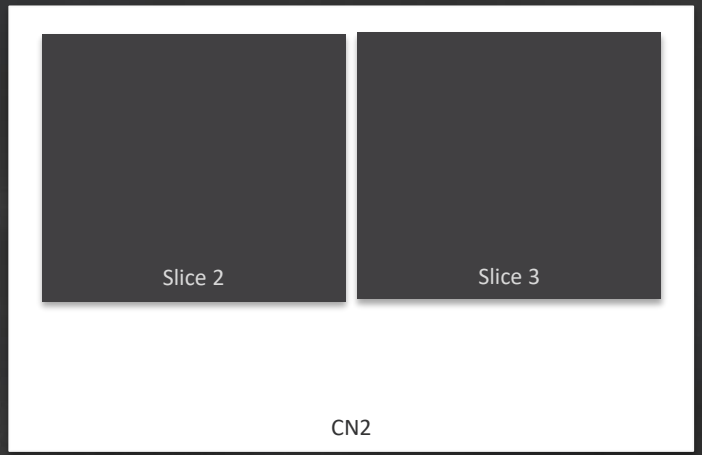


Data Distribution: Example

```
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

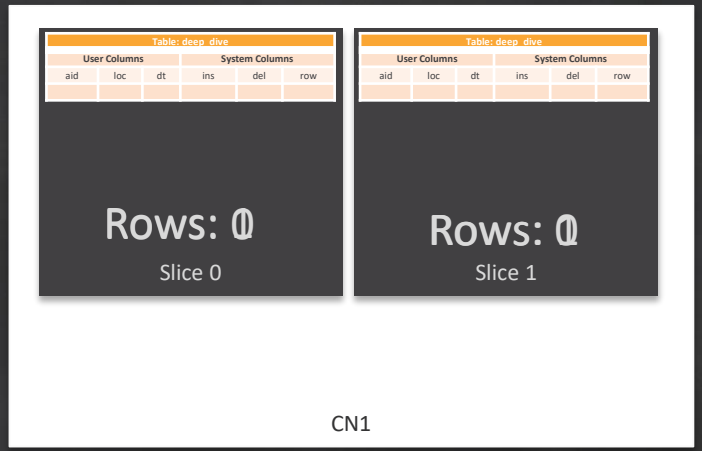


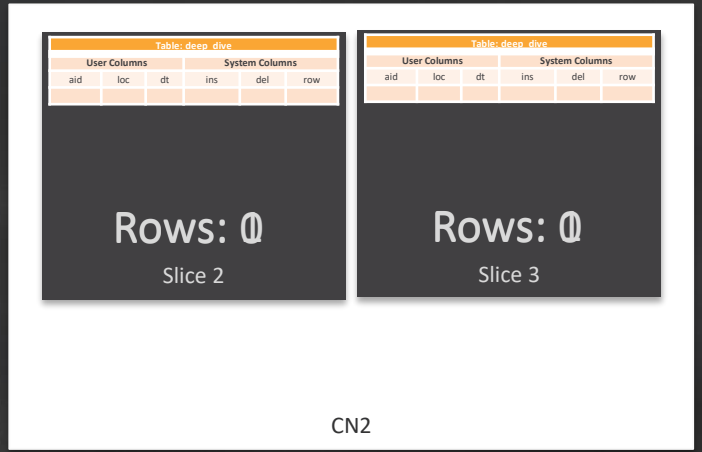


Data Distribution: EVEN Example

```
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');
```





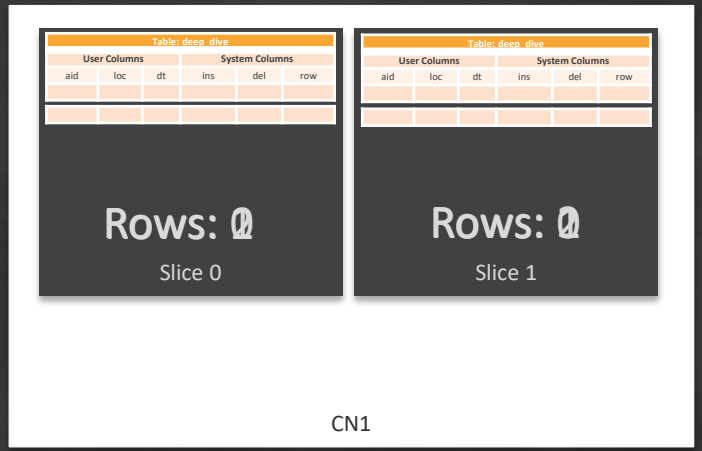
(3 User Columns + 3 System Columns) x (4 slices) = 24 Blocks (24MB)

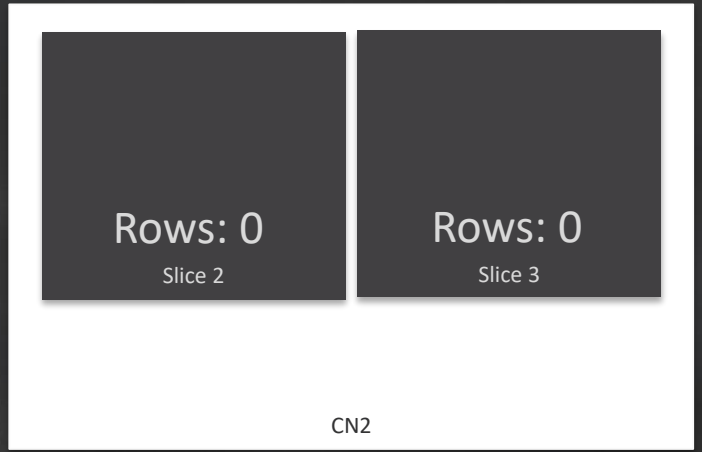
aws

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');
```





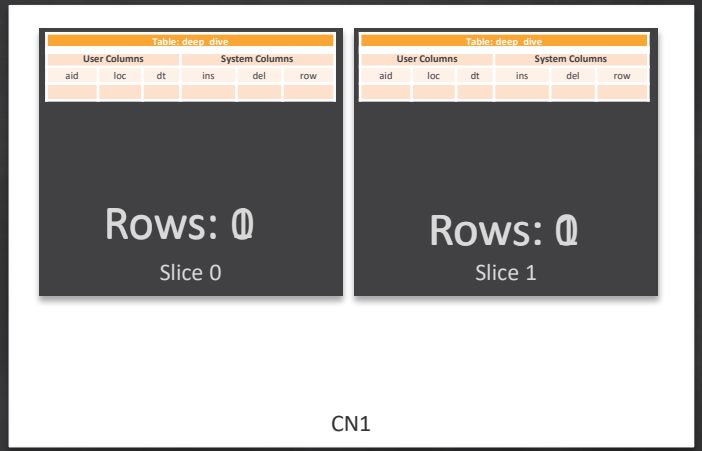
(3 User Columns + 3 System Columns) x (2 slices) = 12 Blocks (12 MB)

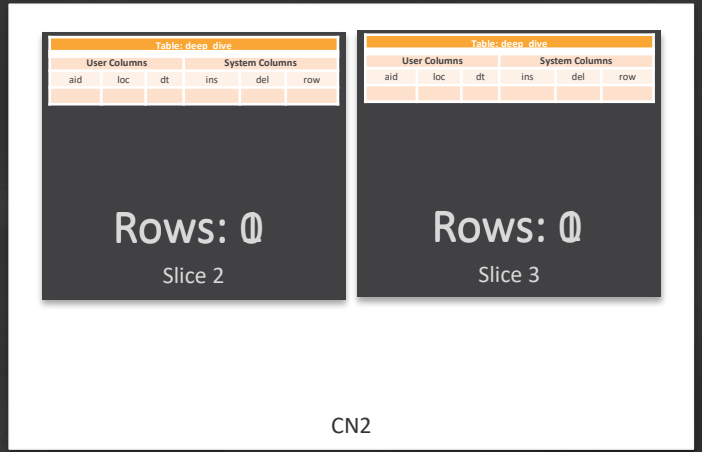
aws

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');
```



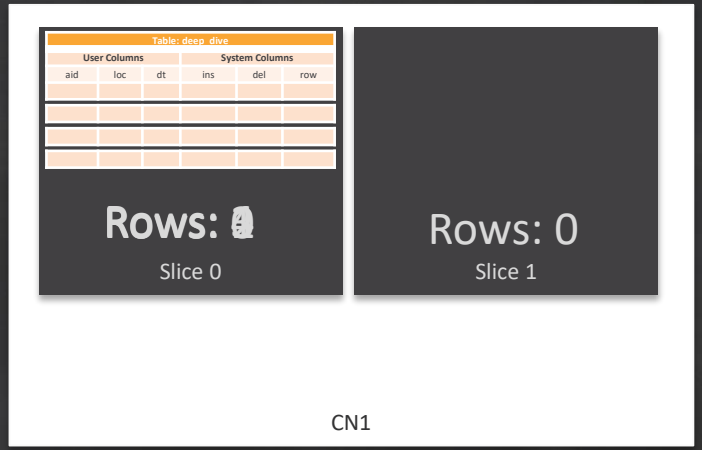


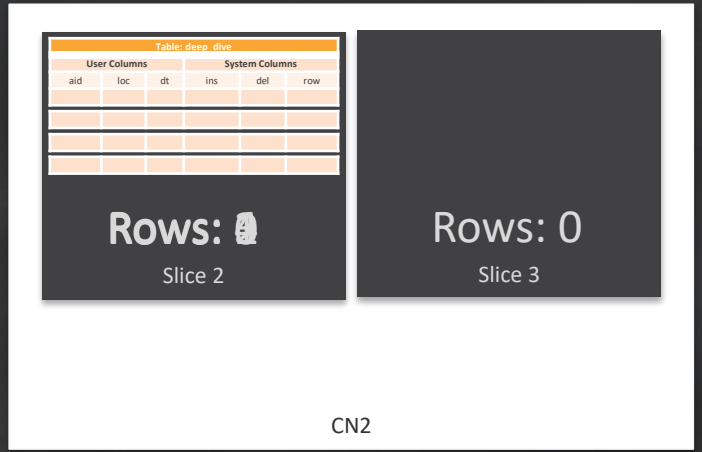
(3 User Columns + 3 System Columns) x (4 slices) = 24 Blocks (24 MB)

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');
```





(3 User Columns + 3 System Columns) x (**2 slice**) = 12 Blocks (12MB)

Terminology and Concepts: Data Distribution

- KEY
  - The key creates an even distribution of data
  - Joins are performed between large fact/dimension tables
  - Optimizing joins and group by
- ALL
  - Small and medium size dimension tables (< 2-3M)
- EVEN
  - When key cannot produce an even distribution

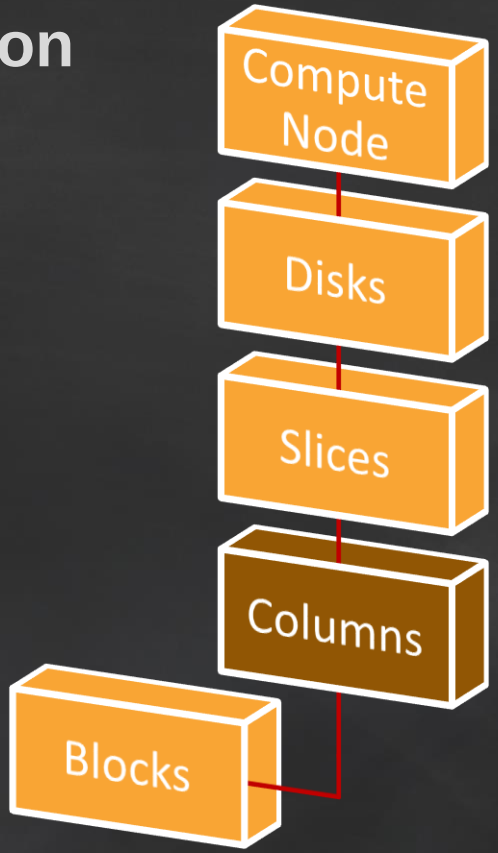


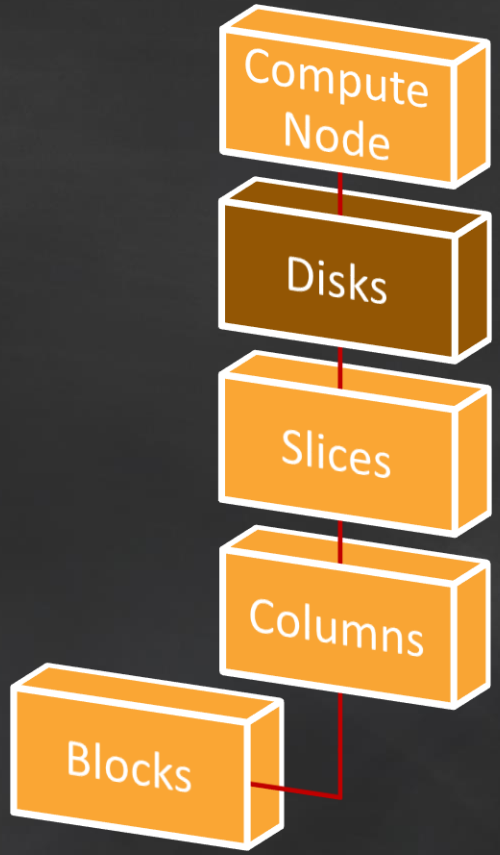
Table Design Considerations

- Denormalize/materialize heavily filtered columns into the fact table(s)
  - Timestamp/date should be in fact tables rather than using time dimension tables
- Add SORT KEYS to medium and large sized tables on the primary columns that are filtered on
  - Not necessary to add them to small tables
- Keep variable length data types as long as required
  - VARCHAR, CHAR and NUMERIC
- Add compression to tables
  - Optimal compression can be found using ANALYZE COMPRESSION

aws

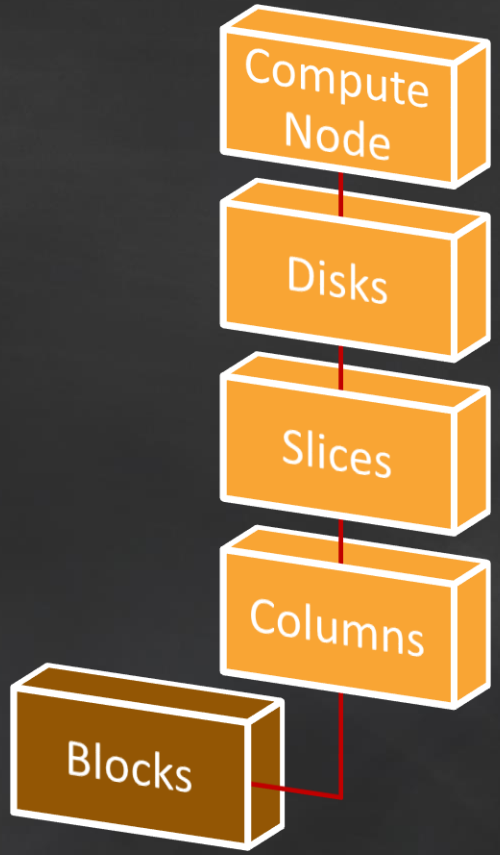
Storage Deep Dive: Disks

- Redshift utilizes locally attached storage devices
- Compute nodes have **2.5-3x** the advertised storage capacity
- 1, 3, 8, or 24 disks depending on node type
- 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



Storage Deep Dive: Blocks

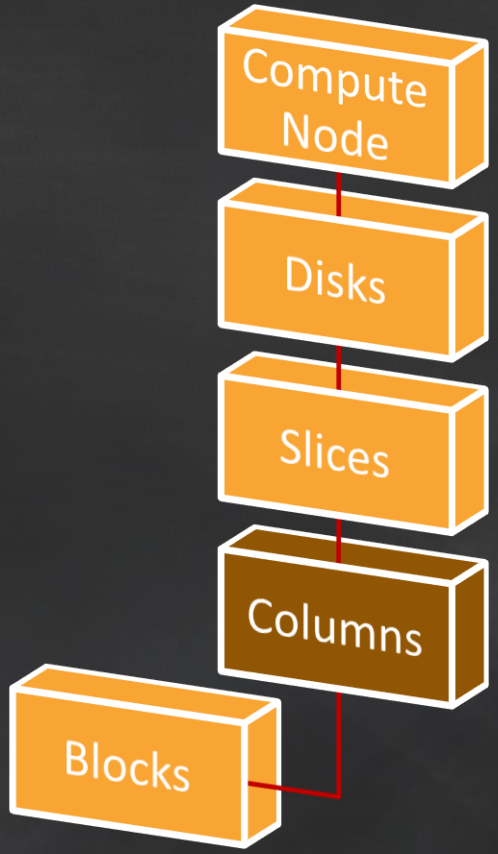
- Column data is persisted to **1MB immutable blocks**
- Each block contains in-memory metadata:
  - Zone Maps (MIN/MAX value)
  - Location of previous/next block
- Blocks are individually compressed with 1 of 10 encodings
- A full block contains between 16 and 8.4 million values



Storage Deep Dive: Columns

- **Column**: Logical structure accessible via SQL
- Physical structure is a doubly linked list of blocks
- These **blockchains** exist on **each slice** for each column
- All sorted & unsorted blockchains compose a column

- Column properties include:
  - Distribution Key
  - Sort Key
  - Compression Encoding
- Columns shrink and grow independently, 1 block at a time
- Three system columns per table-per slice for MVCC



Block Properties: Design Considerations

- Small writes:
  - Batch processing system, optimized for processing massive amounts of data
  - 1MB size + 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 (~100 K 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

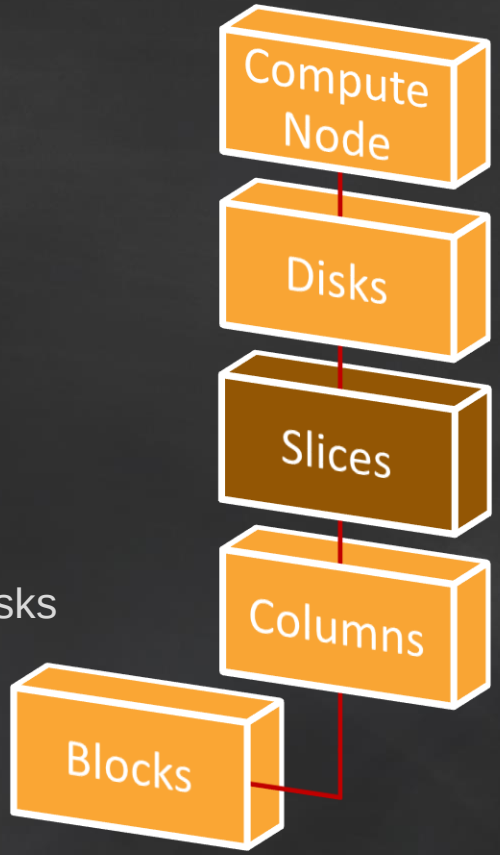
aws

Column Properties: Design Considerations

- Compression:
  - COPY automatically analyzes and compresses data when loading into empty tables
  - ANALYZE COMPRESSION checks existing tables and proposes optimal compression algorithms for each column
  - Changing column encoding requires a table rebuild
- DISTKEY and SORTKEY significantly influence performance (orders of magnitude)
- Distribution Keys:
  - A poor DISTKEY can introduce data skew and an unbalanced workload
  - A query completes only as fast as the slowest slice completes
- Sort Keys:
  - A sortkey is only effective as the data profile allows it to be
  - Selectivity needs to be considered

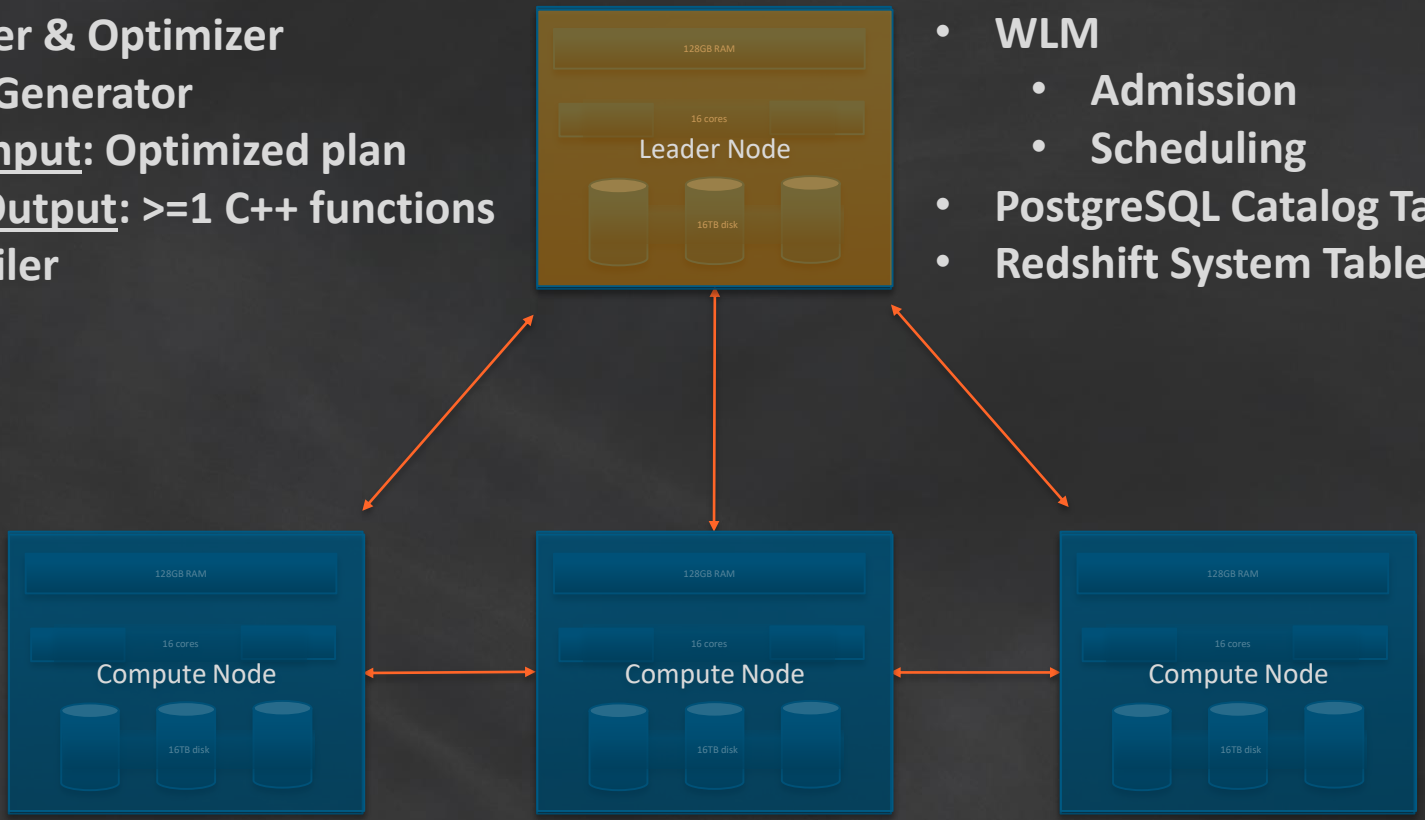
Storage Deep Dive: Slices

- Each compute node has either 2, 16, or 32 slices
- A **slice** can be thought of like a “virtual compute node”
  - Unit of data partitioning
  - Parallel query processing
- Facts about slices:
  - Table rows are distributed to slices
  - A slice processes only its own data
  - Within a compute node all slices read from and write to all disks

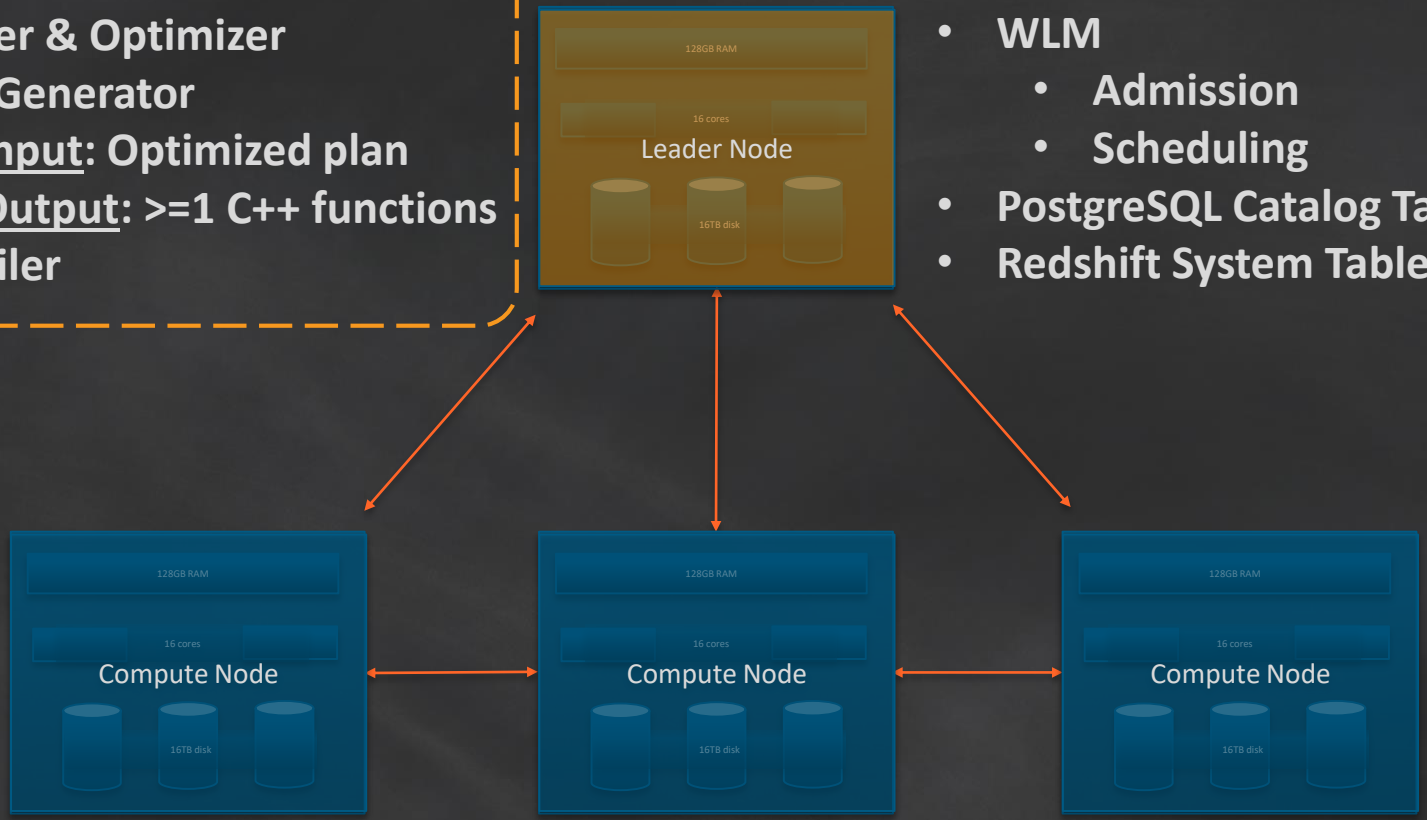


- Parser & Rewriter
- Planner & Optimizer
- Code Generator
  - Input: Optimized plan
  - Output: >=1 C++ functions
- Compiler

- Task Scheduler
- WLM
  - Admission
  - Scheduling
- PostgreSQL Catalog Tables
- Redshift System Tables (STV)



- Parser & Rewriter
- Planner & Optimizer
- Code Generator
  - Input: Optimized plan
  - Output: >=1 C++ functions
- Compiler



- Task Scheduler
- WLM
  - Admission
  - Scheduling
- PostgreSQL Catalog Tables
- Redshift System Tables (STV)

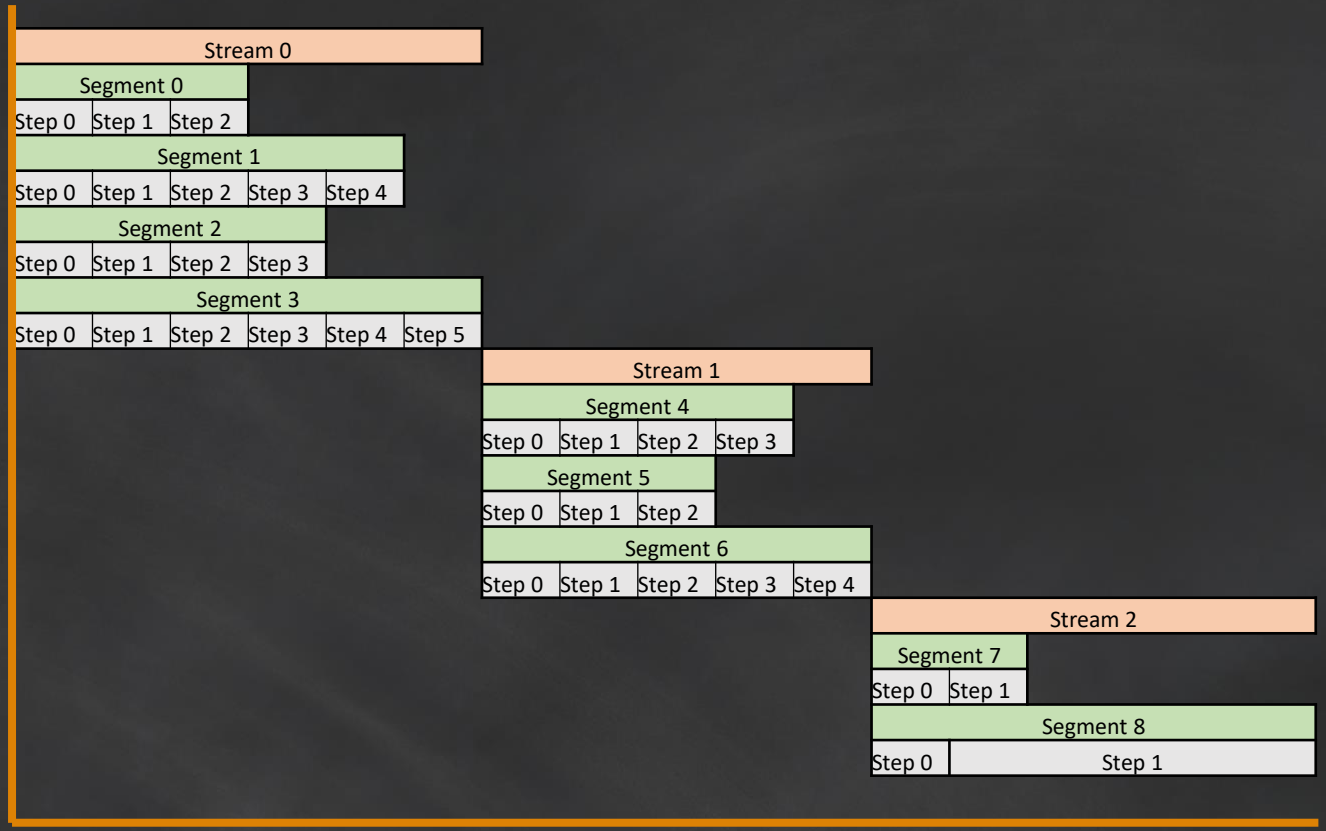
Query Execution Terminology

- **Step:** An individual operation needed during query execution. Steps need to be combined to allow compute nodes to perform a join. Examples: scan, sort, hash, aggr

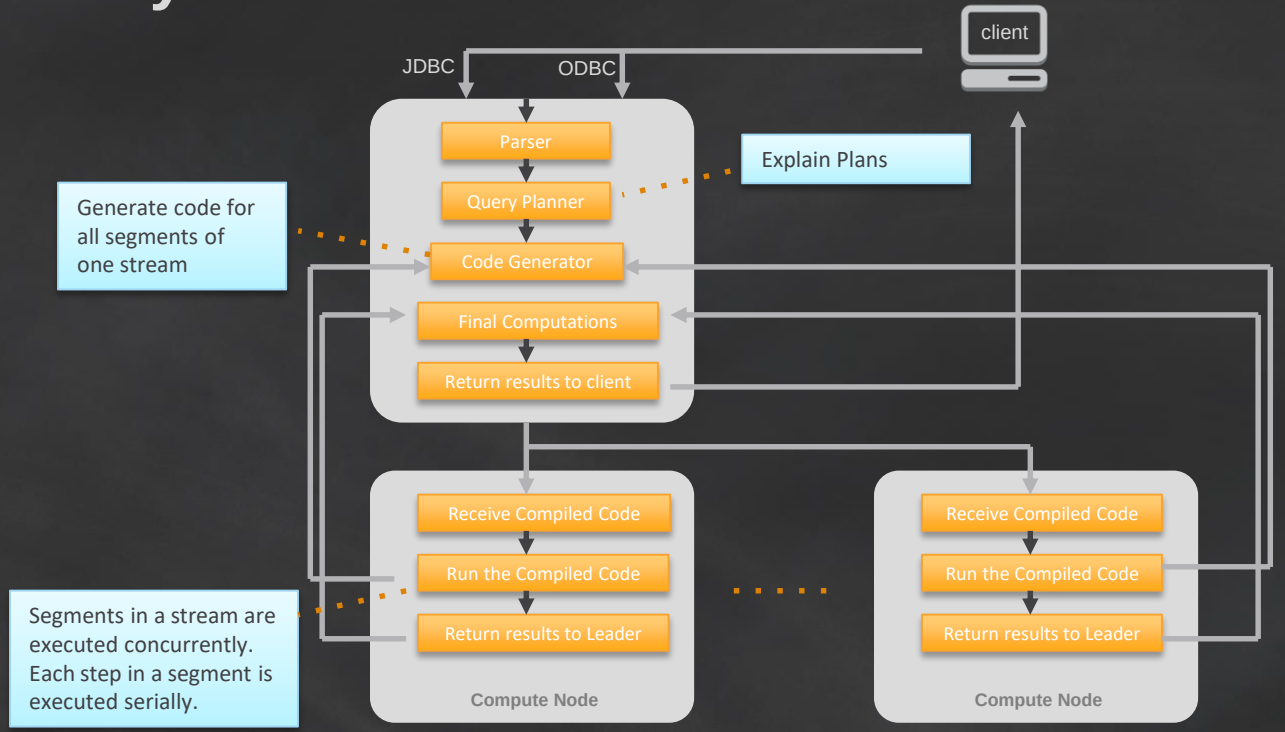
- **Segment:** A combination of several steps that can be done by a single process. The smallest compilation unit executable by a slice. Segments within a stream run in parallel.

- **Stream:** A collection of combined segments which output to the next stream or SQL client.

Visualizing Streams, Segments, and Steps



Query Lifecycle



Query Execution Deep Dive: Leader Node

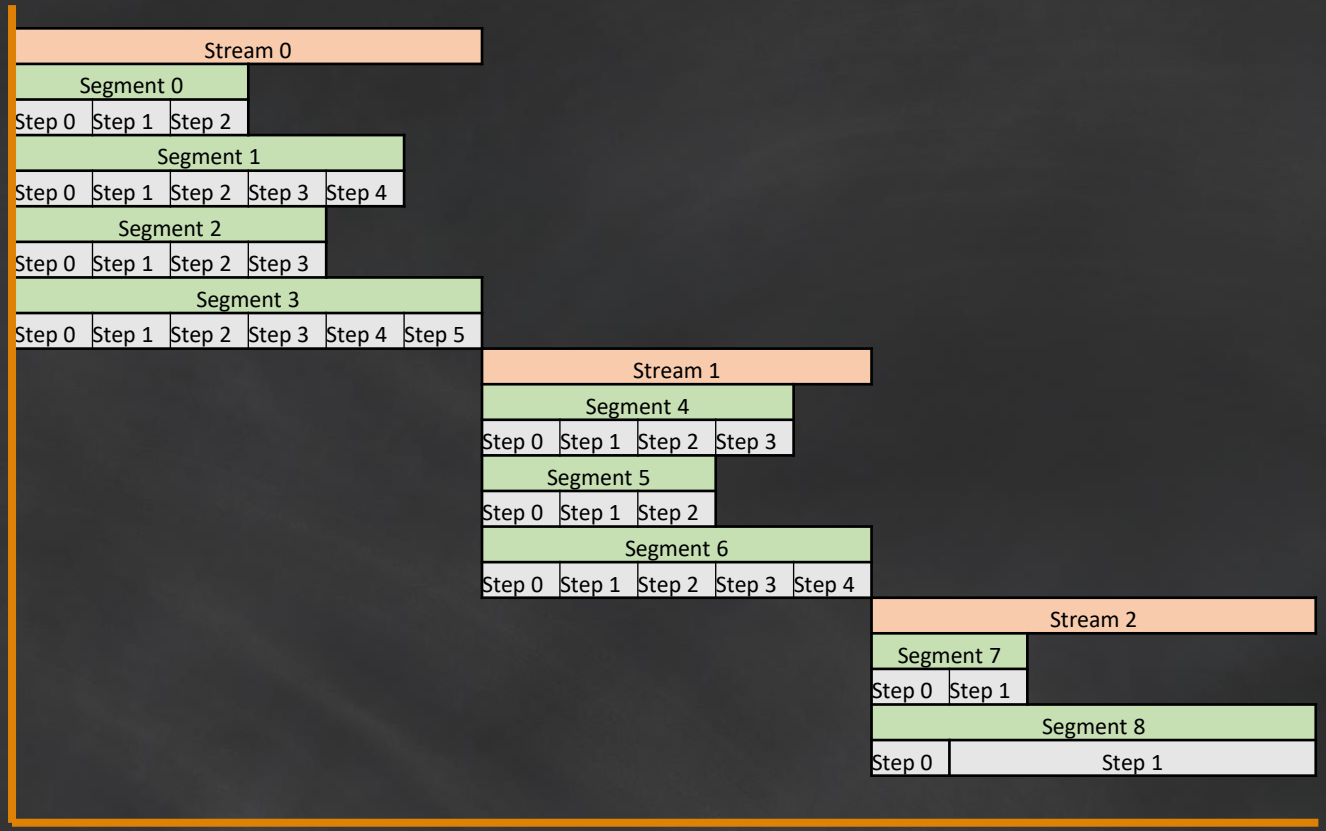
1. The leader node receives the query and parses the SQL.
2. The parser produces a logical representation of the original query.
3. This query tree is input into the query optimizer (volt).
4. Volt rewrites the query to maximize its efficiency. Sometimes a single query will be rewritten as several dependent statements in the background.
5. The rewritten query is sent to the planner which generates >= 1 query plans for the execution with the best estimated performance.
6. The query plan is sent to the execution engine, where it's translated into **steps**, **segments**, and **streams**.
7. This translated plan is sent to the code generator, which generates a C++ function for each **segment**.
8. This generated C++ is compiled with gcc to a .o file and distributed to the compute nodes.

Query Execution Deep Dive: Compute Nodes

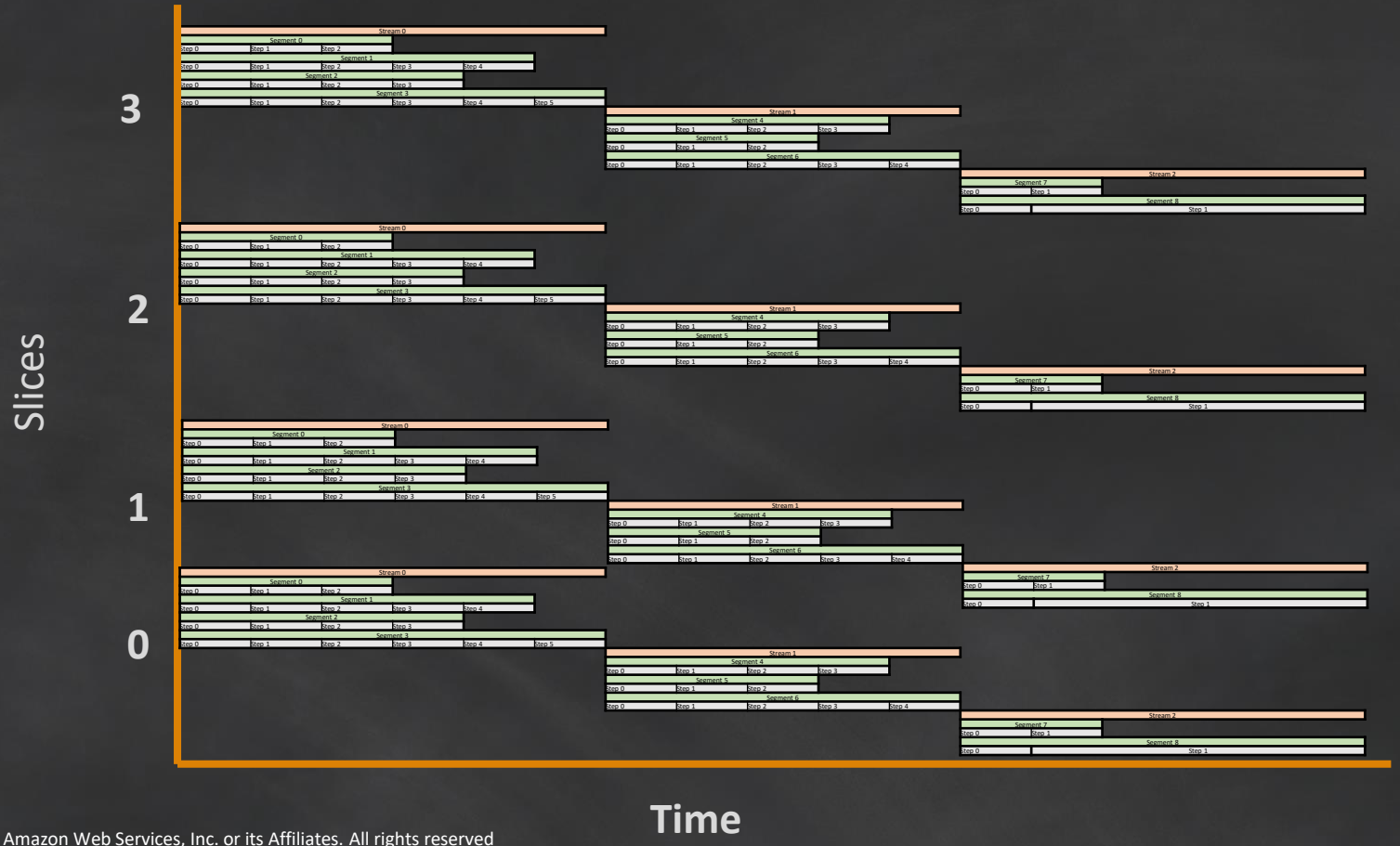
- Slices execute the query segments in parallel.
- Executable segments are created for one stream at a time. When the segments of that stream are complete, the engine generates the segments for the next stream.
- When the compute nodes are done, they return the query results to the leader node for final processing.
- The leader node merges the data into a single result set and addresses any needed sorting or aggregation.
- The leader node then returns the results to the client.

aws

Visualizing Streams, Segments, and Steps

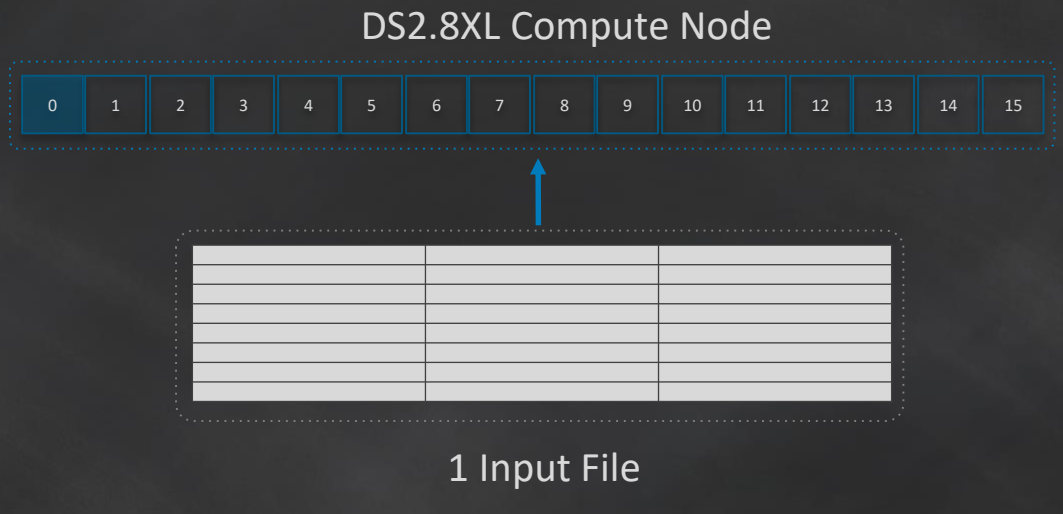


Query Execution



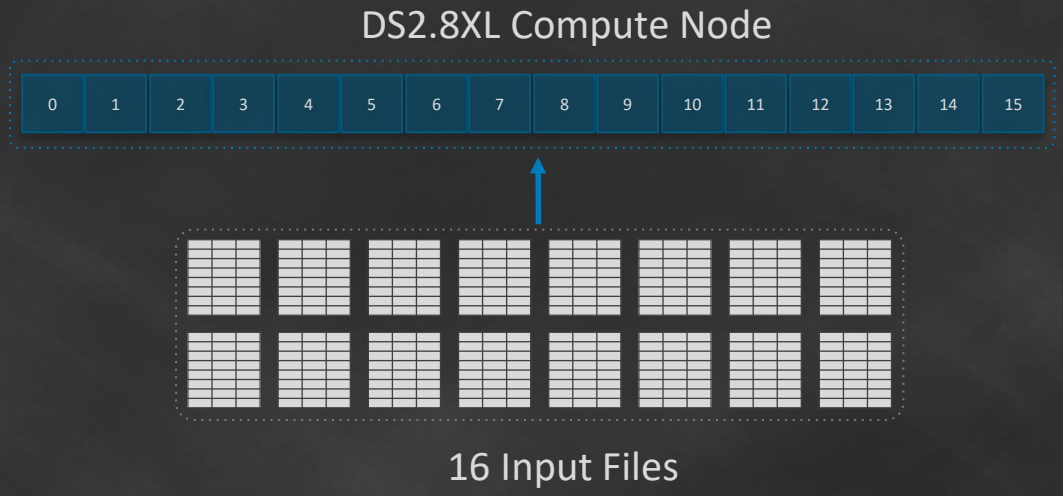
Data Ingestion: COPY Statement

- 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 – 1MB to 1GB after GZIP compression

Best Practices: Data Ingestion

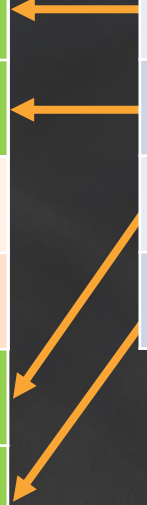
- Export Data from Source System
  - Delimited Files are recommend
    - Pick a simple delimiter '|' or ',' or tabs
    - Pick a simple NULL character (\N)
    - Use double quotes and an escape character (' \ ') for varchars
    - UTF-8 varchar columns take 4 bytes per char
  - Split Files so there is a multiple of the number of slices
  - Files sizes should be 1MB – 1GB after gzip compression

- Useful COPY Options
  - MAXERRORS
  - ACCEPTINVCHARS
  - NULL AS

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

Data Ingestion: Deduplication / UPSERT

```
BEGIN;
CREATE TEMP TABLE staging(LIKE deep_dive);
COPY INTO staging :'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;
```

Best Practices: Deduplication / UPSERT

- Wrap workflow/statements in an explicit transaction

- With Staging Tables:
  - Use temporary tables or “NO BACKUP” option
  - If possible use DISTSTYLE KEY on both the staging table and production table to speed up the INSERT/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

aws

Query Monitoring Rules (QMR)

- Goal - Automatic handling of run-away (poorly written) queries

- Extension to WLM
- Rules applied onto a WLM Queue
  - Queries can be LOGGED, CANCELED or HOPPED

Query Monitoring Rules (QMR)

- Metrics with operators and values (e.g. query_cpu_time > 1000) 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

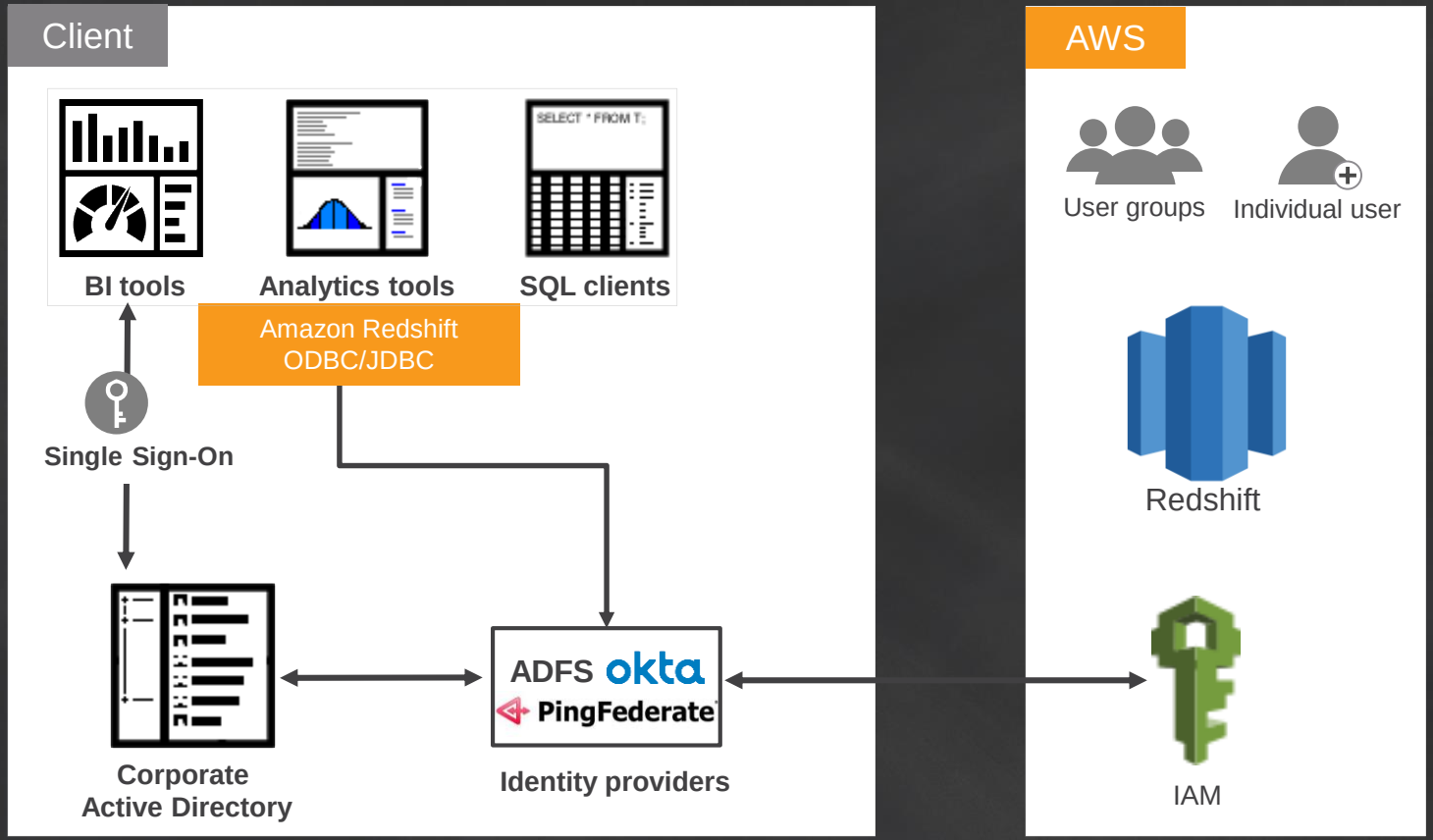
If { *rule* } then [*action*]

{ rule : metric operator value } e.g.: rows_scanned > 100000

- 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

IAM Authentication



New Redshift ODBC/JDBC drivers. Grab the ticket (userid) and get a SAML assertion.

Amazon Redshift Spectrum

Run SQL queries directly against data in S3 using thousands of nodes

Fast @ exabyte scale

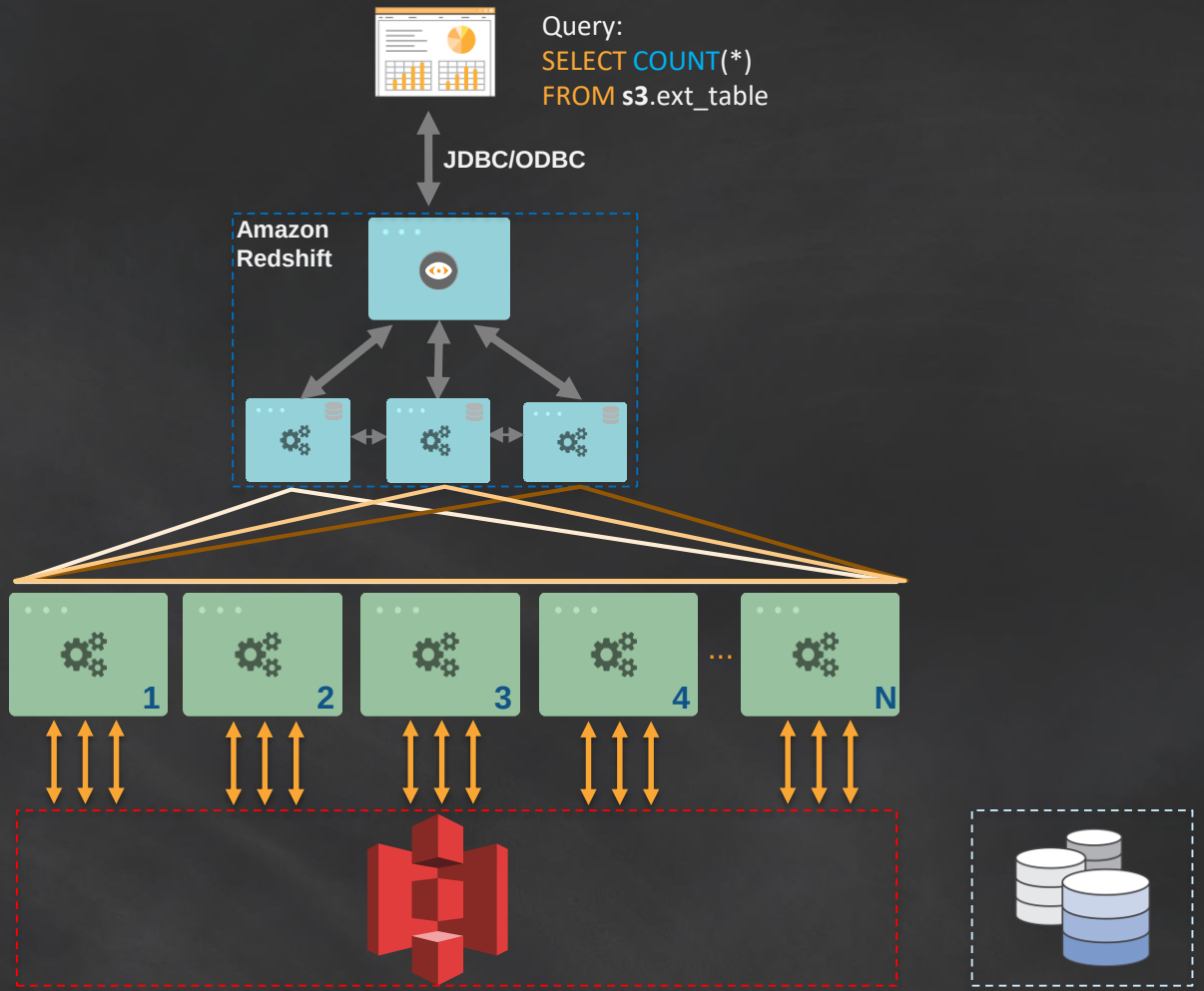
Elastic & highly available

On-demand, pay-per-query

High concurrency: Multiple clusters access same data

No ETL: Query data in-place using open file formats

Full Amazon Redshift SQL support



Amazon S3

Exabyte-scale object storage

Data Catalog

Apache Hive Metastore

An Exabyte Query: Harry Potter

```
demo=# SELECT
demo-#   P.ASIN,
demo-#   P.TITLE,
demo-#   R.POSTAL_CODE,
demo-#   P.RELEASE_DATE,
demo-#   SUM(D.QUANTITY * D.OUR_PRICE) AS SALES_sum
demo-# FROM s3.d_customer_order_item_details D, asin_attributes A, products P, regions r
demo-# WHERE D.ASIN = P.ASIN AND
demo-#       P.ASIN = A.ASIN AND
demo-#       D.REGION_ID = R.REGION_ID AND
demo-#       A.EDITION LIKE '%FIRST%' AND
demo-#       P.TITLE LIKE '%Potter%' AND
demo-#       P.AUTHOR = 'J. K. Rowling' AND
demo-#       R.COUNTRY_CODE = 'US' AND
demo-#       R.CITY = 'Seattle' AND
demo-#       R.STATE = 'WA' AND
demo-#       D.ORDER_DAY :: DATE >= P.RELEASE_DATE AND
demo-#       D.ORDER_DAY :: DATE < dateadd(day, 3, P.RELEASE_DATE)
demo-# GROUP BY P.ASIN, P.TITLE, R.POSTAL_CODE, P.RELEASE_DATE
demo-# ORDER BY sales_sum DESC
demo-# LIMIT 20;
```

Roughly 140 TB of customer item order detail records for each day over past 20 years.

190 million files across 15,000 partitions in S3. One partition per day for USA and rest of world.

Need a **billion-fold** reduction in data processed.

Running this query using a 1000 node Hive cluster would take over 5 years.*

- Compression ……..…….5X
- Columnar file format……..……10X

- Scanning with 2500 nodes…….2500X

- Static partition elimination……..…..2X
- Dynamic partition elimination……350X
- Redshift's query optimizer…….40X

Total reduction……..……**3.5B X**

* Estimated using 20 node Hive cluster & 1.4TB, assume linear
* Query used a 20 node DC1.8XLarge Amazon Redshift cluster
* Not actual sales data - generated for this demo based on data format used by Amazon Retail.

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, etc.
- **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 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!

Darin Briskman
briskman@amazon.com
aws.amazon.com/activate