Amazon Redshift – Performance Tuning and Optimization

Dario Rivera – AWS Solutions Architect

Agenda

- Service overview
- Top 10 Redshift Performance Optimizations
- What's new?
- Q & A (~15 minutes)

Service overview

Amazon Redshift

a lot faster
a lot simpler
a lot cheaper

Relational data warehouse

Massively parallel; petabyte scale

Fully managed

HDD and SSD platforms

$1,000/TB/year; starts at $0.25/hour

Selected Amazon Redshift customers

Amazon Redshift system architecture

Leader node

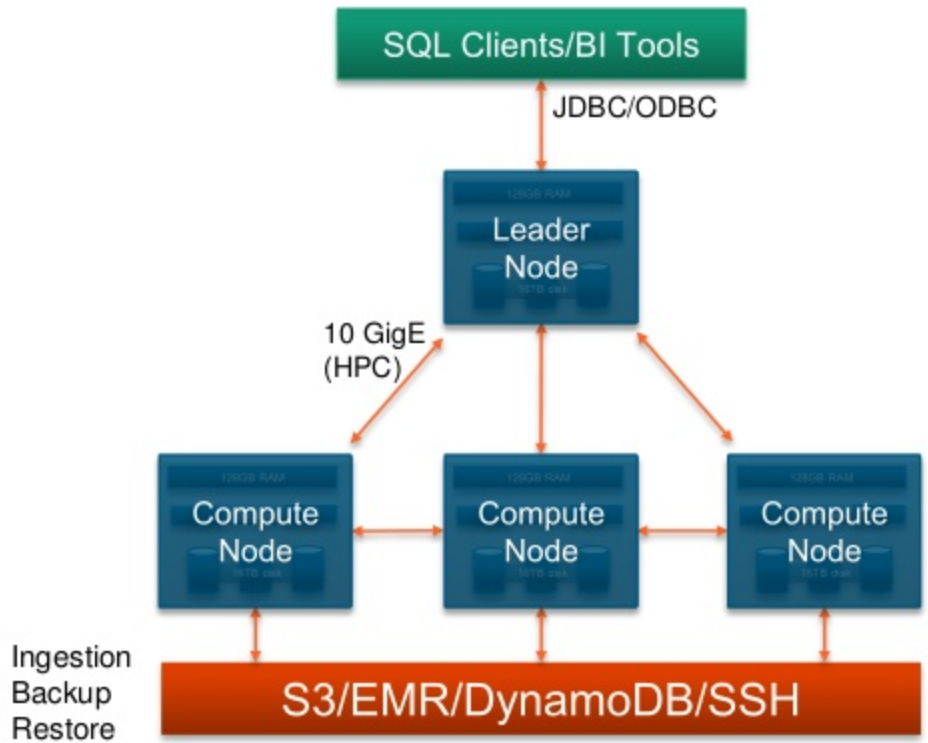
- SQL endpoint
- Stores metadata
- Coordinates query execution

Compute nodes

- Local, columnar storage
- Executes queries in parallel
- Load, backup, restore via Amazon S3; load from Amazon DynamoDB, Amazon EMR, or SSH

Two hardware platforms

- Optimized for data processing
- DS2: HDD; scale from 2 TB to 2 PB
- DC1: SSD; scale from 160 GB to 326 TB



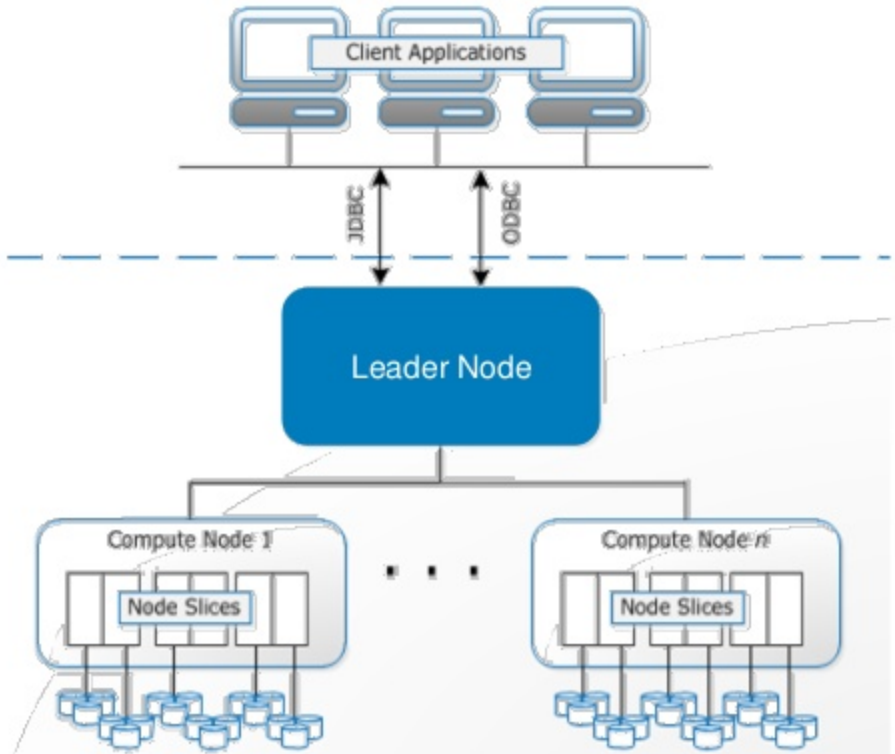
A deeper look at compute node architecture

Each node contains multiple slices

- DS2 – 2 slices on XL, 16 on 8 XL
- DC1 – 2 slices on L, 32 on 8 XL

Each slice is allocated CPU and table data

Each slice processes a piece of the workload in parallel



Let’s get to It!! –
Top 10 Best Practices

Issue #1: Incorrect column encoding

- Amazon Redshift is a column-oriented database
- well suited to analytics queries on tables with a large number of columns
- Only able to access those blocks on disk that are for columns included in the SELECT or WHERE clause, and doesn’t have to read all table data to evaluate a query
- Data stored by column should also be **encoded**
- Clusters without column encoding is not considered a best practice

Solution #1: the v_extended_table_info view

- To determine if you are deviating from this best practice, you can use the **v_extended_table_info** view from the Amazon Redshift Utils GitHub repository
- Once view is created, run the following view:
    - SELECT database, tablename, columns FROM admin.v_extended_table_info ORDER BY database;

Solution #1: the v_extended_table_info view

- Afterward, review the tables and columns which aren't encoded by running the following query:

```
SELECT trim(n.nspname || '.' || c.relname) AS "table",
  trim(a.attname) AS "column",
  format_type(a.atttypid, a.atttypmod) AS "type",
  format_encoding(a.attencodingtype::integer) AS "encoding",
  a.attsortkeyord AS "sortkey"
FROM pg_namespace n, pg_class c, pg_attribute a
WHERE n.oid = c.relnamespace
AND c.oid = a.attrelid
AND a.attnum > 0
AND NOT a.attisdropped and n.nspname NOT IN ('information_schema','pg_catalog','pg_toast')
AND format_encoding(a.attencodingtype::integer) = 'none'
AND c.relkind='r'
AND a.attsortkeyord != 1
ORDER BY n.nspname, c.relname, a.attnum;
```

Solution #1: Redshift Column Encoding Utility

- If you find that you have tables without optimal column encoding, then use the Amazon Redshift Column Encoding Utility on AWS Labs GitHub to apply encoding.
- This utility uses the ANALYZE COMPRESSION command on each table.
- If encoding is required, it generates a SQL script which creates a new table with the correct encoding, copies all the data into the new table, and then transactionally renames the new table to the old name while retaining the original data.

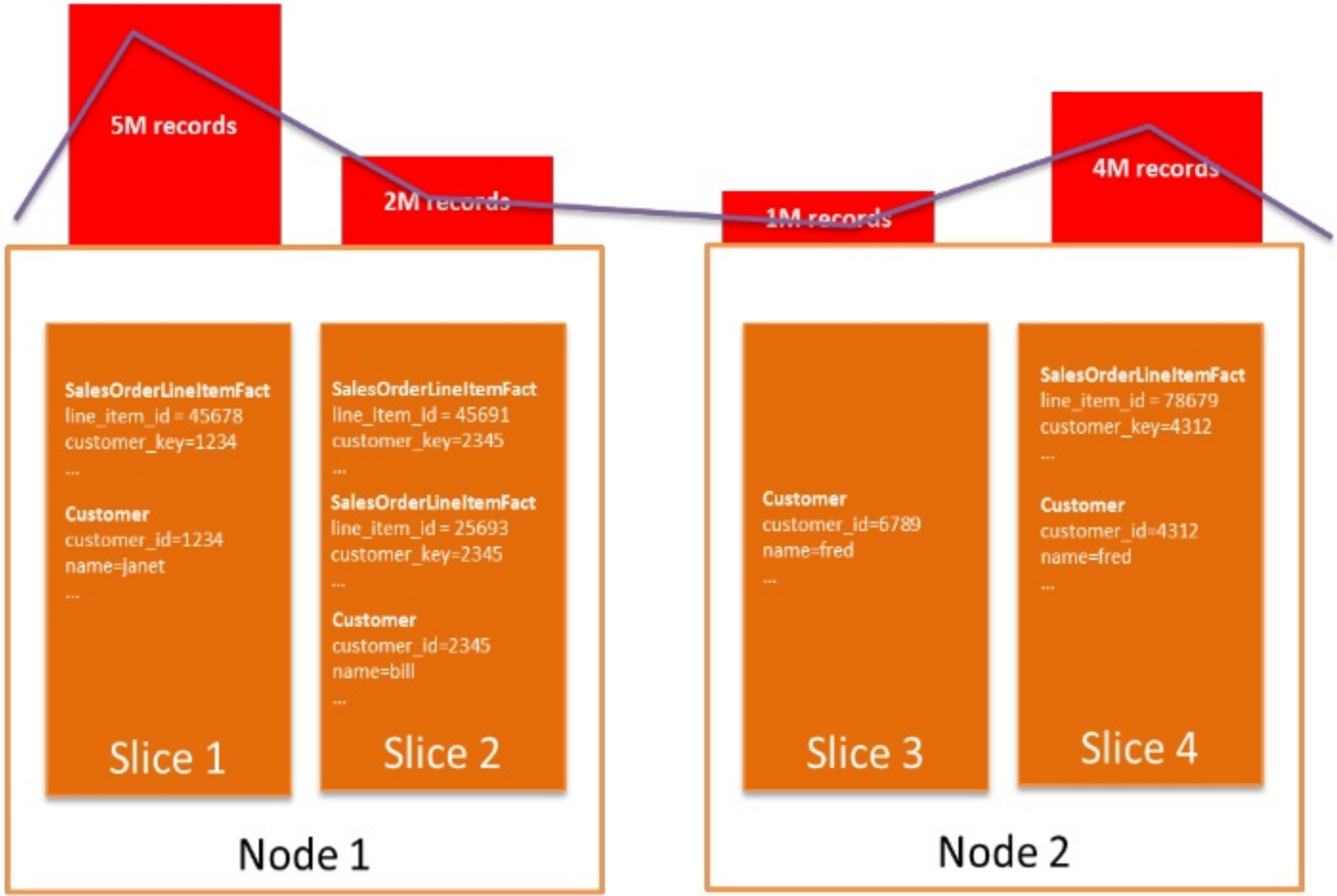
Issue #2 – Skewed table data

- Redshift nodes are managed by the number of slices (CPUs) per node
- When a table is created, you decide whether to spread the data evenly among slices (default), or assign data to specific slices on the basis of one of the columns.
- By choosing columns for distribution that are commonly joined together, you can minimize the amount of data transferred over the network during the join.
- This can **significantly** increase performance on these types of queries.

Issue #2 – Skewed table data

- The selection of a good distribution key is the topic of many AWS articles, including Choose the Best Distribution Style; see a definitive guide to distribution and sorting of star schemas in the Optimizing for Star Schemas and Interleaved Sorting on Amazon Redshift blog post.
- A **skewed distribution key** results in slices not working equally hard as each other during query execution, requiring unbalanced CPU or memory, and ultimately only running as fast as the slowest slice:

Issue #2 – Skewed table data



If skewing is an issue:

- Use one of the admin scripts in the Amazon Redshift Utils GitHub repository, such as table_inspector.sql, to see how data blocks in a distribution key map to the slices and nodes in the cluster.

Solution #2: Use Correct Distribution Key

- If you find that you have tables with skewed distribution keys, then consider changing the distribution key to a column that exhibits **high cardinality** and **uniform distribution**.
- Evaluate a candidate column as a distribution key by creating a new table using CTAS:

```
CREATE TABLE my_test_table DISTKEY (<column name>) AS SELECT <column name> FROM <table name>;
```

- Run the table_inspector.sql script against the table again to analyze data skew.

Solution #2: Use Correct Distribution Key

- If there is no good distribution key in any of your records, you may find that moving to EVEN distribution works better.
- For small tables (for example, dimension tables with a couple of million rows), you can also use DISTSTYLE ALL to place table data onto every node in the cluster.

Issue #3 – Queries not benefiting from sort keys

- Sort keys acts like an index in other databases, but which does not incur a storage cost as with other platforms. (for more information, see Choosing Sort Keys)
- To determine which tables don't have sort keys, run the following query against the v_extended_table_info view from the Amazon Redshift Utils repository:

```
SELECT * FROM admin.v_extended_table_info WHERE sortkey IS null
```

Solution #3: Use Sort Key on Columns used in Where Clause

- A sort key should be created on those columns which are most commonly used in WHERE clauses.
  - If you have a known query pattern, then COMPOUND sort keys give the best performance;
  - If using compound sort keys, review your queries to ensure that their WHERE clauses specify the sort columns in the same order they were defined in the compound key.
  - if end users query different columns equally, then use an INTERLEAVED sort key.

Solution #3: Run Sort Key Recommendation Query

- You can run a tutorial that walks you through how to address unsorted tables in the *Amazon Redshift Developer Guide*.
- You can also run the following query to generate a list of recommended sort keys based on query activity:

  http://tinyurl.com/redshift-sortkey-recommender

- Queries evaluated against a sort key column must not apply a SQL function to the sort key; instead, ensure that you apply the functions to the compared values so that the sort key is used. This is commonly found on TIMESTAMP columns that are used as sort keys.

Will use sort key	Won't use sort key
`SELECT count(*) FROM flight_data WHERE fl_date BETWEEN '2010-12-31 00:00:00' AND '2010-12-31 23:59:00'`	`SELECT count(*) FROM flight_data WHERE cast (fl_date AS date) = '2010-12-31'`

Issue #4 – Tables without statistics or which need vacuum

- Bad Stats: Amazon Redshift, like other databases, requires statistics about tables and the composition of data blocks being stored in order to make good decisions when planning a query (for more information, see Analyzing Tables).

- Stale Data: When rows are DELETED or UPDATED, they are simply logically deleted (flagged for deletion) but not physically removed from disk. Updates result in a new block being written with new data appended. As a result, table storage space is increased and performance degraded

Solution #4: missing_table_stats.sql admin script

- The ANALYZE Command History topic in the *Amazon Redshift Developer Guide* supplies queries to help you address missing or stale statistics
- Or, you can also simply run the missing_table_stats.sql admin script to determine which tables are missing stats, or the statement below to determine tables that have stale statistics:

```
SELECT database, schema || '.' || "table" AS "table", stats_off
FROM svv_table_info
WHERE stats_off > 5
ORDER BY 2;
```

Solution #4: perf_alert.sql admin script

- You can use the perf_alert.sql admin script to identify tables for which alerts about scanning a large number of deleted rows have been raised in the last seven days.
- To address issues with tables with missing or stale statistics or where vacuum is required, run another AWS Labs utility, Analyze & Vacuum Schema. This ensures that you always keep up-to-date statistics, and only vacuum tables that actually need re-organisation.

Issue #5 – Tables with very large VARCHAR columns

During processing of complex queries, intermediate query results can be stored in temporary uncompressed blocks, consuming excessive memory and temporary disk space, affecting query performance. For more information, see Use the Smallest Possible Column Size.

Solution #5 – Inspect and Deep Copy to Optimized Table

Use the following query to generate a list of tables that should have their maximum column widths reviewed:

```
SELECT database, schema || '.' || "table" AS "table", max_varchar
FROM svv_table_info
WHERE max_varchar > 150
ORDER BY 2
```

After you have a list of tables, identify which table columns have wide varchar columns and then determine the true maximum width for each wide column, using the following query:

```
SELECT max(len(rtrim(column_name)))
FROM table_name;
```

If you find that the table has columns that are wider than necessary, then you need to re-create a version of the table with appropriate column widths by performing a deep copy.

Solution #5 – Working with JSON Columns

- In some cases, you may have large VARCHAR type columns because you are storing JSON fragments in the table, which you then query with JSON functions.
- Pay special attention to SELECT * queries which include the JSON fragment column, from the top_queries.sql admin script.
    - If end users query these large JSON columns but don't execute JSON functions against them, consider moving them into another table that only contains the primary key column of the original table and the JSON column.

Issue #6 - Queries waiting on queue slots

Amazon Redshift runs queries using a queuing system known as workload management (WLM), allowing up to 8 separate workloads.

In some cases, the queue to which a user or query has been assigned is completely busy and a user's query must wait for a slot to be open. During this time, the system is not executing the query at all, which is a sign that you may need to increase concurrency.

Solution #6 – Assess Queuing History and configure WLM

- determine if any queries are queuing, using the queuing_queries.sql admin script.
- Review the maximum concurrency that your cluster has needed in the past with wlm_apex.sql,
    - down to an hour-by-hour historical analysis with wlm_apex_hourly.sql.
- Modify WLM to optimal queuing configuration based on history stats

Solution #6 – Configure WLM

-Note: while increasing concurrency allows more queries to run, each query will get a smaller share of the memory allocated to its queue (unless you increase it).

You may find that by increasing concurrency, some queries must use temporary disk storage to complete, which is also sub-optimal

Issue #7 - Queries that are disk-based

- If a query isn't able to completely execute in memory, it may need to use disk-based temporary storage for parts of an explain plan.
- The additional disk I/O slows down the query, and can be addressed by increasing the amount of memory allocated to a session (for more information, see WLM Dynamic Memory Allocation).

Solution #7 – Disk Write Check and Configure

- To determine if any queries have been writing to disk, use the following query: **http://tinyurl.com/redshift-diskquery**
- Based on the user or the queue assignment rules, you can increase the amount of memory given to the selected queue to prevent queries needing to spill to disk to complete.
- You can also increase the WLM_QUERY_SLOT_COUNT …use with care!

Issue #8 – Commit Queue Waits

- Amazon Redshift is designed for **analytics queries**, rather than **transaction processing**.

- *The cost of COMMIT is relatively high*, and excessive use of COMMIT can result in queries waiting for access to a commit queue.

Solution 8: Analyze Commit Workloads

- If you are committing too often on your database, you will start to see waits on the commit queue increase,
- Use the commit_stats.sql admin script to show the largest queue length and queue time for queries run in the past two days.
- If you have queries that are waiting on the commit queue, then look for sessions that are **committing multiple times** per session, such as ETL jobs that are logging progress or inefficient data loads.

Issue #9 - Inefficient data loads

- Anti-Pattern: Insert data directly into Amazon Redshift, with single record inserts or the use of a multi-value INSERT statement,
- These INSERTs allow up to a 16 MB ingest of data at one time.
- These are leader node–based operations, and can create significant performance bottlenecks by maxing out the leader node network as data is distributed by the leader to the compute nodes.

Solution #9 – Use COPY cmd, and Compression

- Amazon Redshift best practices suggest the use of the COPY command to perform data loads. This API operation uses all compute nodes in the cluster to load data in parallel.
- Compress the files to be loaded whenever possible; Amazon Redshift supports both GZIP and LZO compression.

Solution #9 – Use COPY cmd, and Compression

- It is more efficient to load a large number of small files than one large one, and the ideal file count is a multiple of the slice count.
- The number of slices per node depends on the node size of the cluster.
- By ensuring you have an equal number of files per slice, you can know that COPY execution will evenly use cluster resources and complete as quickly as possible.

Solution #9 – Table Load Statistics Scripts

The following query calculates statistics for each load:

http://preview.tinyurl.com/redshift-ingest-load-stats

The following query shows the time taken to load a table, and the time taken to update the table statistics, both in seconds and as a percentage of the overall load process:

http://tinyurl.com/redshift-tableload-stat-time

Issue #10 - Inefficient use of Temporary Tables

Overview of Temp Tables:

- Amazon Redshift provides temporary tables, which are like normal tables except that they are only visible within a single session.
- When the user disconnects the session, the tables are automatically deleted.
- Temporary tables can be created using the CREATE TEMPORARY TABLE syntax, or by issuing a SELECT … INTO #TEMP_TABLE query.
- The CREATE TABLE statement gives you complete control over the definition of the temporary table, while the SELECT … INTO and C(T)TAS commands use the input data to determine column names, sizes and data types, and use default storage properties.

Issue #10 - Inefficient use of Temporary Tables

- These default storage properties may cause issues if not carefully considered.
- Amazon Redshift's default table structure is to use EVEN distribution with no column encoding.
- This is a sub-optimal data structure for many types of queries, and if you are using SELECT…INTO syntax you cannot set the column encoding or distribution and sort keys.
- If you use the CREATE TABLE AS (CTAS) syntax, you can specify a distribution style and sort keys, but you still can't set the column encodings.

Solution #10 – Apply Source Table configurations to Target Temp Table

If you are creating temporary tables, it is highly recommended that you convert all SELECT…INTO syntax to use the CREATE statement. This ensures that your temporary tables have column encodings and are distributed in a fashion that is sympathetic to the other entities that are part of the workflow.

Solution #10 – Apply Source Table configurations to Target Temp Table

If you typically do this:

SELECT column_a, column_b INTO #my_temp_table FROM my_table;

You would analyze the temporary table for optimal column encoding:

```
master=# analyze compression #my_temp_table;
     Table      | Column   | Encoding
----------------+---------+----------
 #my_temp_table | column_a | lzo
 #my_temp_table | column_b | bytedict
(2 rows)
```

Solution #10 – Apply Source Table configurations to Target Temp Table

And then convert the select/into statement to:

```
BEGIN;
CREATE TEMPORARY TABLE my_temp_table(
column_a varchar(128) encode lzo,
column_b char(4) encode bytedict)
distkey (column_a) -- Assuming you intend to join this table on column_a
sortkey (column_b); -- Assuming you are sorting or grouping by column_b

INSERT INTO my_temp_table SELECT column_a, column_b FROM my_table;
COMMIT;
```

You may also wish to analyze statistics on the temporary table, if it is used as a join table for subsequent queries:

```
ANALYZE my_temp_table;
```

Solution #11 - Using explain plan alerts

- The last tip is to use diagnostic information from the cluster during query execution. This is stored in an extremely useful view called STL_ALERT_EVENT_LOG.

- Use the perf_alert.sql admin script to diagnose issues that the cluster has encountered over the last seven days. This is an invaluable resource in understanding how your cluster develops over time.

Open source tools

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.

ColumnEncodingUtility

Gives you the ability to apply optimal column encoding to an established schema with data already loaded

Remember to complete your evaluations!

Thank you!

amazon web services