



BEST PRACTICES FOR

# HIGH-PERFORMANCE ETL TO REDSHIFT



## Table of Contents

Amazon Redshift - Overview	1
Take Advantage Of Columnar Data Storage	2
Compression	3
Zone Maps And Data Sorting	4
Take Advantage Of Data Distribution	6
Disk Consideration While Designing ETL	11
Perform Multiple Steps In A Single Transaction	12
Take Advantage Of Copy Statement Features	13
Perform Vacuum and Analyze	14
Avoid Creating Unnecessary Varchar Columns	15
Take Advantage Of Massively Parallel Processing (MPP)	15
Use Workload Management To Improve ETL Runtimes	16
Redshift Spectrum For Ad Hoc Query Processing	18
Bonus - There is An Easier Way To Perform ETL!	19

## Amazon Redshift - Overview



Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the cloud. You can start with just a few hundred gigabytes of data and scale to a petabyte or more.

To design robust ETL platform and deliver results in a timely manner it is very much important to leverage Amazon Redshift Architecture. As a prerequisite for this ebook you would need to understand the same. If you are new to this space, refer this article about [Amazon Redshift](#) to get a head start.

There are a lot of cool features in Amazon Redshift which ETL designers need to take into account while delivering their solution. This ebook will cover the best practices for High-Performance ETL processing using Amazon Redshift.

## Take Advantage Of Columnar Data Storage

Instead of storing data as a series of rows, Amazon Redshift organizes the data by columns. Since only the column involved in the queries are processed and columnar data is stored sequentially on the storage media, column-based systems require far fewer I/Os, greatly improving query performance. So, make sure you select only the necessary columns in your select condition or filter condition while querying your tables.

Example:-

```
CREATE TABLE SAMPLE_TABLE2
AS SELECT ID, LOC, DT FROM SAMPLE_TABLE; → This will scan only relevant
column

CREATE TABLE SAMPLE_TABLE2
AS SELECT * FROM SAMPLE_TABLE; → This is a bad example as this query is doing
the full table scan and unnecessary I/O
```

Select only the relevant columns while designing your ETL.

## Compression

Compression is a column-level operation that reduces the size of data when it is stored. Compression conserves storage space and reduces the size of data that is read from storage, which reduces the amount of disk I/O and therefore improves query performance. There are various compression types provided by Amazon Redshift. Depending on your ETL logic, make sure you select the right compression technique on columns. AWS Redshift has done a great job in suggesting right compression type per column. ANALYZE COMPRESSION is a built-in command that will find the optimal compression for each column on an existing table.

Example:-

Analyze compression listing

Table	Column	Encoding	Est_reduction_pct
listing	listid	delta	75.00
listing	sellerid	delta32k	38.14

You can also verify whether columns are compressed by hitting pg\_table\_def table.

Example:-

```
SELECT "column", type, encoding FROM pg_table_def
WHERE tablename='listing';
```

Column	Type	Encoding
listid	integer	delta
sellerid	integer	delta32k

## Zone Maps And Data Sorting

AWS Redshift column data is persisted to 1 MB immutable blocks. A full block can contain millions of values for that column. A zone map exists for each 1 MB block and consists of in-memory metadata that tracks the minimum and maximum values within the block. All blocks automatically have zone maps, which helps to effectively prune blocks which cannot contain data for a given query. When you create a table, you can define one or more of its columns as sort keys. When data is initially loaded into the empty table, the rows are stored on disk in sorted order. Thus, data sorting makes zone maps more effective. However, while designing your table always make sure to provide your sort keys on the table where you are using a filter condition to one or more columns.

Example:-

```
CREATE TABLE sample_table (  
  aid integer,  
  loc varchar(3),  
  dt date  
) SORT KEY (dt, loc);  
SELECT count(*) from sample_table where dt= '01-01-2018';
```

However, keep a note that in your queries you should place the sort keys on columns that are frequently filtered by placing the lowest cardinality columns first, typically date columns.

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

Example of bad table design:-

```
CREATE TABLE sample_table (  
  aid integer,  
  loc varchar(3),  
  dt date  
) SORT KEY (aid, loc);  
SELECT COUNT(*) from sample_table where dt= '01-01-2018'; → This is bad  
example as table scans are not properly utilised during selection
```

Sort keys are less beneficial on small tables. Define four or fewer sort keys else it will increase ingestion overhead. If you are unsure of choosing the best sort-key, refer this article about [choosing right sort-key](#) to get a head start.

## Take Advantage Of Data Distribution

The disk storage for a compute node is divided into a number of slices. The number of slices per node depends on the node size of the cluster. For example, each DS1.XL compute node has two slices, and each DS1.8XL compute node has 16 slices. The nodes all participate in parallel query execution, working on data that is distributed as evenly as possible across the slices. When you load data into Amazon Redshift, you should aim to have each slice do an equal amount of work.

There are 3 distribution styles provided by AWS Redshift.

- a) Even distribution - The leader node distributes the rows across the slices in a round-robin fashion, regardless of the values in any particular column.
- b) Key distribution - The rows are distributed according to the values in one column. The leader node will attempt to place matching values on the same node slice.
- c) ALL distribution - A copy of the entire table is distributed to every node. Where EVEN distribution or KEY distribution place only a portion of a table's rows on each node.

When designing your ETL tables make sure you select right distribution keys else it will hamper the performance badly.



### Example 1:-

```
CREATE TABLE sample_table (
  id integer,
  loc varchar(3),
  dt date
) DISTSTYLE EVEN;
insert into sample_table values
(1, 'ABC', '01-01-2018'),
(2, 'DEF', '05-01-2018'),
(3, 'GHI', '01-01-2018'),
(4, 'JKL', '05-01-2018');
```

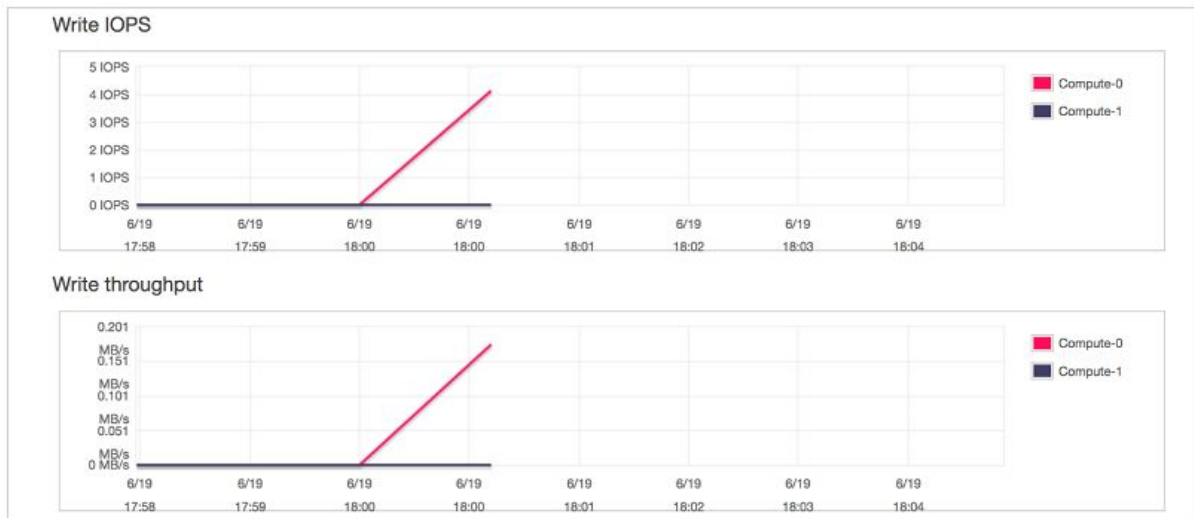
This insert will insert the data evenly into all the slices in each node.

### Example 2:-

```
CREATE TABLE sample_table (
  id integer,
  loc varchar(3),
  dt date
) DISTSTYLE KEY DISTKEY (dt)
INSERT INTO sample_table values
(1, 'ABC', '01-01-2018'),
(2, 'DEF', '05-01-2018'),
(3, 'GHI', '01-01-2018'),
(4, 'JKL', '05-01-2018');
```

This is a case of bad example as data will only go to the single node and other nodes (node) will not have any data resulting in data skew.

As you can see below node compute-0 is performing all Write IOPS and showing throughput in the node stats while node compute-1 is not used at all.



Example 3:-

```
CREATE TABLE sample_table (
  id integer,
  loc varchar(3),
  dt date
) DISTSTYLE KEY DISTKEY (id)
INSERT INTO sample_table values
(1, 'ABC', '01-01-2018'),
(2, 'DEF', '05-01-2018'),
(3, 'GHI', '01-01-2018'),
(4, 'JKL', '05-01-2018');
```

This is a case of good example as data will be evenly distributed to all slices and node, in addition to, taking advantage of MPP.

#### Example 4:-

```
CREATE TABLE sample_table (
  id integer,
  loc varchar(3),
  dt date
) DISTSTYLE ALL;
INSERT INTO sample_table values
(1, 'ABC', '01-01-2018'),
(2, 'DEF', '05-01-2018'),
(3, 'GHI', '01-01-2018'),
(4, 'JKL', '05-01-2018');
```

In the above example, the same set of data will go to each slice of multiple nodes.

Best Practices for choosing right data distribution:

- a) Choose DISTSTYLE KEY to optimize join performance between large tables.
- b) Choose DISTSTYLE KEY when copying data from one table to other.

Whenever you design your ETL tables, it is highly recommended to get the skew ratio for tables having DISTSTYLE KEY.

```
SELECT diststyle, skew_rows FROM svv_table_info WHERE "table"='sample_table';
```

column | type

-----+-----

key(id) | 1.07 → This is the ratio between the most and least number of rows.

Remember ideally this value should be close to 1.

- c) Choose DISTSTYLE ALL on smaller (dimension tables) or tables having rows < 3M. Choosing DISTSTYLE ALL will eat up a lot of disk space.
- d) If neither KEY nor ALL works, or if you are unsure, pick DISTSTYLE EVEN (which is default).

Temporal columns as distribution key should be avoided as distribution around will not happen properly which could lead to performance issues.

If you are inquisitive to know the various Redshift Distribution Styles, you can refer the article about [DISTSTYLE](#).

## Disk Consideration While Designing ETL

Whenever queries are written and as soon as the commit is performed AWS Redshift automatically duplicates the copy of data into another node. To avoid this step, use of Temporary tables are highly recommended as this step will be skipped. It saves disk space as well as improves performance. AWS also performs the backup of every 5 GB of changed data or every 8 hours whichever happens first to S3. So, to avoid this step it is recommended to use create the table with backup no option. As a rule of thumb, keep 3 times the free space of largest table in the cluster.

Example for backup no option:-

```
CREATE TABLE sample_table (id integer) backup no; → This step will skip the step of data copy into another node.
```

Example for creating a temp table and copy the data from another table including distkey:-

```
CREATE temp TABLE tmp_sample_table (like sample_table) → This will also take distkey and compression setting from the source table.
```

## Perform Multiple Steps In A Single Transaction

ETL transformation logic often spans multiple steps as commits in Amazon Redshift are expensive. If each ETL step performs a commit, multiple concurrent ETL processes can take a long time to execute. As a result of the expense of commit overhead, limit commits by explicitly creating transactions. Use drop table or truncate table instead of delete table to avoid ghost rows.

Example:-

```
BEGIN;  
CREATE temp TABLE tmp_sample_table (like sample_table);  
INSERT INTO tmp_sample_table values (1,2,3);  
DELETE FROM tmp_sample_table WHERE id=1;  
COMMIT;
```

## Take Advantage Of Copy Statement Features

Whenever copy statement is issued to load the data into a table, it is advisable to split up the file and take advantage of parallel load else only the single slice will take an entire load of processing. If the file is split up, it can easily take advantage of parallel processing for which AWS Redshift has already been designed. Use of delimited file is recommended. A file size of more than 1 GB could take more time.

When using copy statement it is recommended to use `COMPUPDATE OFF`. This will avoid Redshift figuring out compression over and over again. It is a great feature and highly recommended, as it will also help to improve Redshift CPU utilization.

Try to use `COPY` data from multiple, evenly sized files so that all the nodes are equally utilised.

Example:-

```
BEGIN;  
CREATE temp TABLE tmp_sample_table (like sample_table);  
copy tmp_sample_table from 's3://bucket/sample.csv' : 'creds' COMPUPDATE OFF;  
COMMIT;
```

When copying huge amount of data (> hundreds of million) consider using `ALTER TABLE APPEND` instead of `INSERT INTO SELECT`.

## Perform Vacuum and Analyze

To get the best performance from your Amazon Redshift database, you must ensure that database tables regularly are VACUUMED and ANALYZED. Whenever you add, delete, or modify a significant number of rows, you should run a VACUUM command and then an ANALYZE command. A vacuum recovers the space from deleted rows and restores the sort order. The ANALYZE command updates the statistics metadata, which enables the query optimizer to generate more accurate query plans.

However, it is highly recommended that:-

- a) Vacuum should be run typically nightly or weekly and not during business hours.
- b) When you are filtering on a specific column, analyze can be run just on that column alone.



## Avoid Creating Unnecessary Varchar Columns

Though Redshift handles large unused varchar datatypes. It is not recommended to use unnecessary large data size varchar datatype columns. This will consume excessive memory and temporary disk space.

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;
```

## Try To Avoid Running Redshift Cluster On EC2 Cluster

Instead, run your Redshift cluster on VPC. It runs much faster on VPC.

## Take Advantage Of Massively Parallel Processing (MPP)

Amazon Redshift automatically distributes data and query load across all nodes. Amazon Redshift makes it easy to add nodes to your data warehouse and enables you to maintain fast query performance as your data warehouse grows. Hence, keep an eye on the nodes and make sure nodes are not overloaded with data.

## Use Workload Management To Improve ETL Runtimes

Use Amazon Redshift's workload management (WLM) to define multiple queues dedicated to different workloads (for example, ETL versus Reporting) and to manage the runtimes of queries. As you migrate more workloads into Amazon Redshift, your ETL runtimes can become inconsistent if WLM is not appropriately set up. It is recommended to set overall concurrency of WLM across all queues to around 15 or less. For more details on WLM setup, you can refer [AWS documentation](#). When managing different workloads on your Amazon Redshift cluster, consider the following for the queue setup:

- a) Create a queue dedicated to your ETL processes. Configure this queue with a small number of slots (5 or fewer). 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 the commit queue. Hence, ETL is a commit-intensive process, having a separate queue with a small number of slots helps mitigate this issue.
- b) Claim extra memory available in a queue. When executing an ETL query, you can take advantage of the `wlm_query_slot_count` to claim the extra memory available in a particular queue. For example, a typical ETL process might involve COPYING raw data into a staging table so that downstream ETL jobs can run transformations that calculate daily, weekly, and monthly aggregates. To speed up the COPY process (so that the downstream tasks can start in parallel sooner), the `wlm_query_slot_count` can be increased for this step.
- c) Create a separate queue for reporting queries. Configure query monitoring rules on this queue to further manage long-running and expensive queries.

STV\_WLM\_CLASSIFICATION\_CONFIG, STV\_WLM\_SERVICE\_CLASS\_CONFIG, and STV\_WLM\_SERVICE\_CLASS\_STATE can be used to see queue status etc. through SQL client.

```
SELECT (config.service_class-5) AS queue
, TRIM (class.condition) AS description
, config.num_query_tasks AS slots
, config.query_working_mem AS mem
, config.max_execution_time AS max_time
, config.user_group_wild_card AS "user_*"
, config.query_group_wild_card AS "query_*"
, state.num_queued_queries queued
, state.num_executing_queries executing
, state.num_executed_queries executed
FROM
STV_WLM_CLASSIFICATION_CONFIG CLASS,
STV_WLM_SERVICE_CLASS_CONFIG config,
STV_WLM_SERVICE_CLASS_STATE state
WHERE
class.action_service_class = config.service_class
AND class.action_service_class = state.service_class
AND config.service_class > 4 -- 1 to 4 is reserved for AWS internal system
ORDER BY config.service_class;
```

queue	description	slots	mem	max_time	user_*	query_*	queued	executing	executed
0	(super user) and (query group: superuser)	1	357	0	false	false	0	0	0
1	(querytype: any)	5	836	0	false	false	0	1	160

## Redshift Spectrum For Ad Hoc Query Processing

Redshift Spectrum enables you to run queries against exabytes of data in Amazon S3. There is no loading or ETL required. Even if you don't store any of your data in Amazon Redshift, you can still use Redshift Spectrum to query datasets as large as an exabyte in Amazon S3. Try to leverage on Redshift Spectrum tables as time can be saved on skipping data load from S3 (In case, we are not getting performance benefit on normal AWS Redshift tables). AWS Spectrum data works in the form of an external table (using S3 read directly). You can join with normal Redshift table and get the performance.

Example:

```
CREATE EXTERNAL TABLE spectrum.sales(
salesid integer,
listid integer,
sellerid integer)
row format delimited
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION 's3://bucket/sales/';
```

To view external tables, query the SVV\_EXTERNAL\_TABLES system view.

```
SELECT top 10 spectrum.sales.eventid, sum(spectrum.sales.pricepaid)
FROM spectrum.sales, event
WHERE spectrum.sales.eventid = event.eventid
AND spectrum.sales.pricepaid > 30
GROUP BY spectrum.sales.eventid
ORDER BY 2 DESC;
```

If you are extracting data for use with Amazon Redshift Spectrum, you should make use of the MAXFILESIZE parameter to keep file size below 150 MB.

## There is An Easier Way To Perform ETL!


The detailed steps of ETL processing using Amazon Redshift mentioned involves multiple complex stages and can be a cumbersome experience.

If you want to load any data easily into Redshift without any hassle, you can try out [Hevo](#). Hevo automates the flow of data from various sources to Amazon Redshift in real time and at zero data loss. In addition to migrating data, you can also build aggregates and joins on Redshift to create materialized views that enable faster query processing.

---

### About Author:

Ankur Shrivastava is a AWS Solution Designer with hands-on experience on Data Warehousing, ETL, and Data Analytics. He is an AWS Certified Solution Architect Associate. In his free time, he enjoys all outdoor sports and practices.



Looking for a simple and reliable way to bring  
Data from Any Source to AWS Redshift?

# TRY HEVO

[SIGN UP FOR FREE TRIAL](#)