AWS Big Data Blog

Best Practices for Micro-Batch Loading on Amazon Redshift

by Ian Meyers | on 25 JUL 2014 | in Amazon Redshift* | Permalink | Comments | Share

https://aws.amazon.com/blogs/big-data/best-practices-for-micro-batch-loading-on-amazon-redshift/

Ian Meyers is a Solutions Architecture Senior Manager with AWS

Data analysts always want the newest data in their data warehouse. Historically, when transaction-optimized databases were used for warehousing analysts would “trickle load” (replicate data from production systems into the data warehouse) at the expense of read throughput. Analytics data warehouses are traditionally loaded nightly or several times per day, but often customers want to refresh data every hour, every minute, or even continuously. This post outlines best practices for using Amazon Redshift for micro-batch loading and is intended for data architects, data modelers, and DBAs.

Any data model can be loaded using micro-batching, but data organized as a time series is often ideal for this type of loading. As outlined in this best practices tutorial, we will load Amazon Redshift using the COPY command because it uses all compute nodes in the cluster and scales most effectively. (With small tables, the multi value insert might also help.) To make the COPY command as efficient as possible, ask it to do as little as possible. The following techniques help ensure an efficient COPY execution into a database that will be loaded every two to five minutes.

Balance input files

To get the best performance possible on load, ensure that every slice in the cluster (one virtual CPU, share of memory, and disk) does an equal amount of work. This provides the shortest runtime per slice and ensures that you can scale close to linearly. To do this, ensure that the number of input files is an even multiple of the slice count. The list below offers guidance for balancing input data by load source:

- If you are loading data from Amazon DynamoDB, this is handled automatically using the Parallel Scan feature. If you haven’t done this before, it’s easy to learn how.

- If you are loading data from Amazon S3, consider using the split command in Bash to do a line-wise split of your input files by slice count before uploading to Amazon S3. For example, to split bigfile.txt into 32 parts with prefix ‘part-’, you might use:
  ```
  split -l `wc -l bigfile.txt | awk '{print $1/32}'` -v bigfile.txt “part-“
  ```

- If you are loading data from HDFS, consider preprocessing your data with a MapReduce job that generates an even multiple of files relative to Amazon Redshift Slices. To do this, run a streaming map reduce job with Hadoop configuration `“mapred.reduce.tasks=”` and output data only from reducers.

Correctly balancing the number of input files can produce some of the largest performance gains for micro-batch loading. Our tests have shown performance improvement by several orders of magnitude over unoptimised load models.

Pre-configure column coding

One great feature of the COPY command is that it automatically applies optimal column encoding to a table when the table is empty. However, this takes time. We can significantly improve efficiency by skipping this step and explicitly listing column encoding on tables to be loaded. To do this, perform a COPY into the table once with a representative data set and review the column encodings listed in pg_table_def. Then drop and recreate the table to be loaded with explicit column encodings. When you perform a data load, add the option ‘COMPUPDATE OFF’ to the copy command to suppress automatic encoding.

For example:

```
master=# select "column", type, encoding from pg_table_def where tablename = 'nations';
```

Column	Type	Encoding
n_nationkey	integer	DELTA
n_name	character(25)	TEXT255

n_regionkey	integer	RUNLENGTH
n_comment	character varying(152)	LZO

Then drop and recreate the table to be loaded with explicit column encodings by adding the ENCODE keyword to the table creation DDL.

```
CREATE TABLE NATIONS (
    N_NATIONKEY INT ENCODE DELTA,
    N_NAME CHAR(25) ENCODE TEXT255,
    N_REGIONKEY INT ENCODE RUNLENGTH,
    N_COMMENT VARCHAR(152) ENCODE LZO
);
```

Reduce frequency of statistics calculation

The COPY command can also compute table statistics for the optimizer on each load. While useful, this also takes time. So, as with column encoding, compute statistics once for a fixed dataset and then suppress recalculation during COPY. To do this, again you need to load the table with a representative data set. Then analyze it with the ANALYZE command. You can stop automatic analysis by setting the COPY option 'STATUPDATE OFF.' Periodically run the ANALYZE on tables to ensure that they are up to date. If the nature of the data being loaded changes significantly over time, statistics and column encoding should be recalculated and stored against the table structure.

Load in sort key order

If the table being loaded has a sort key, you can load the data in this order and avoid the need for a VACUUM of the table. Amazon Redshift sorts the data as it is imported into the cluster, so for tables with date-based sort keys just ensure that the data loads are occurring

chronologically relative to the sort column. For example, load data for the period 14:00-15:00 before the period 15:00-16:00 as defined by the sort key. Always follow best practices for sort key order. Using this load model, you need to VACUUM only to reclaim free space from delete or update operations.

Use SSD Node Type

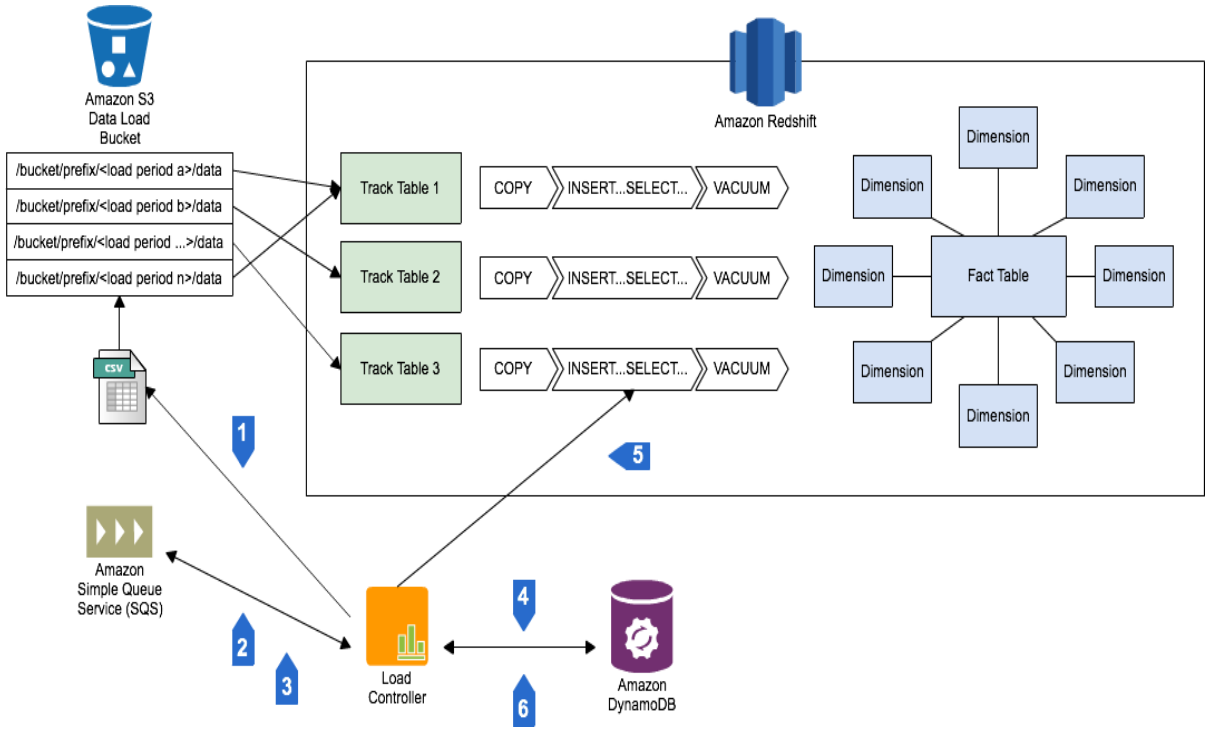
Amazon Redshift can be run using two different node types. The dw1 family uses magnetic spinning disk, and offers 2TB of usable storage on the dw1.xlarge node type, or 16TB of usable storage on the dw1.8xlarge node type. For significantly reduced IO latency, which is advantageous in a micro-batch load scenario, you can use the dw2 family, which offers 160GB of usable storage on the dw2.large node, and 2.56TB on the dw.8xlarge node. This node type offers a higher compute-to-storage ratio and therefore provides lower-latency COPY into track tables.

Loading tables using ‘load stream’ architecture

The load stream, or tracks architecture, discussed below builds on the architecture presented by HasOffers (slide 29). It uses multiple tables to be copied into rather than just one. These tables collectively represent all data that needs to be loaded, but gives us an optimal load path for COPY to follow. Configure your load such that a COPY is done only into an empty table by assigning the load an empty stream or track. This allocation is done as part of the data load orchestration, and the allocation of a free track can be done by updating a database table with a list of free tracks and setting the status to in use. This table does not contain SORT keys, and you use the COPY options listed above to prevent the need to ANALYZE or the computation of column encoding.

Once the table COPY is complete, load the stream into your analytics schema using SQL Commands or ETL tools, and then VACUUM after all data transformations are complete. Then put the stream or track back into the pool of available streams for loading by setting its status in the load state database to free. While this is happening, other COPY commands can be run concurrently into alternate streams or tracks without causing blocking and without the need to hold up processing to do a VACUUM.

The figure below illustrates the micro-batch loading process:



The steps below describe this micro-batch loading process:

1. The files are loaded into dedicated location in Amazon S3.
2. A message is sent to Amazon SQS indicating that a file is ready to be loaded.
3. The load message is received in the loader application.
4. The unique track is allocated in the backing database. Example:
   ```
   update load_tracks set track_state = <load id> where track_state = 'free' and rownum = 1)
   ```
5. The data load is performed:

1. COPY load prefix into track table.
2. Move the data into analytics schema using the tool of your choice.
3. Send an Amazon SQS messaging indicating that the load is completed.
4. Truncate the load track table. e. Initiate VACUUM to reclaim free space.

6. The track is marked as free in the backing database. Example:
   `update load_tracks set track_state = ‘free’ where track_state = <load id>)`

For loading at one-minute frequency, determine how long it takes for a minute of data to be loaded. For durations of less than one minute, you may require only five streams. However, if it takes longer than one minute to load a minute’s worth of data in a stream, allocate streams using the calculation (ceiling(* 2), which gives you adequate open streams to process many parallel micro-batch loads concurrently.

Summary

You can achieve micro-batch loading into Amazon Redshift, especially if you use some or all of the techniques outlined in this post. You can build on this by using a tracks architecture to move the VACUUM process out of the critical path for micro-batch loads, or eliminate the need for VACUUM entirely by loading in SORT KEY order. You can also use node types optimized for low latency. Many customers find that this architecture significantly increases their load frequency while limiting the CPU cost to the cluster.