Athena Best Practices

(30 minutes read)

(Yusuf Arif, Sr. Data Architect, Big Data - usuf@amazon.com)

When should I use Athena?

Athena helps you analyze unstructured, semi-structured, and structured data stored in Amazon S3. Examples include CSV, JSON, or columnar data formats such as Apache Parquet and Apache ORC. You can use Athena to run ad-hoc queries using ANSI SQL, without the need to aggregate or load the data into Athena.

Athena is serverless, so there is no infrastructure to manage, and you pay only for the queries that you run. Athena is easy to use. Simply point to your data in Amazon S3, define the schema, and start querying using standard SQL.

While Amazon Athena is ideal for quick, ad-hoc querying and integrates with Amazon QuickSight for easy visualization, it can also handle complex analysis, including large joins, window functions, and arrays. If you use Athena for complex analysis including large joins, there are some best practices to follow as a rule of thumb for latency and performance of the Amazon Athena Query. These best practices are not exhaustive but should cover most of the issues faced while doing complex analysis with Athena.

Best Practices:

Storage

This section discusses how to structure your data so that you can get the most out of Athena. The same practices can be applied to Amazon EMR data processing applications such as Spark, Presto, and Hive when your data is stored on Amazon S3.

1. Partition your data

Partitioning divides your table into parts and keeps the related data together based on column values such as date, country, region, etc. Partitions act as virtual columns. You define them at table creation, and they can help reduce the amount of data scanned per query, thereby improving performance. You can restrict the amount of data scanned by a query by specifying filters based on the partition. For more details, see Partitioning Data.

Athena supports Hive partitioning, which follows one of the following naming convention:

a) Partition column name followed by an equal symbol ('=') and then the value.

s3://yourBucket/pathToTable/<PARTITION_COLUMN_NAME>=<VALUE>/<PARTITION_COLUMN_NAME>=<VALUE>/

If your dataset is partitioned in this format, then you can run the MSCK REPAIR table command to add partitions to your table automatically.

b) If the "path" of your data does not follow the above format, you can add the partitions manually using the ALTER TABLE ADD PARTITION command for each partition. For example

s3://yourBucket/pathToTable/YYYY/MM/DD/

Alter Table <tablename> add Partition (PARTITION_COLUMN_NAME = <VALUE>, PARTITION_COLUMN2_NAME = <VALUE>) LOCATION 's3://yourBucket/pathToTable/YYYY/MM/DD/';

Note: using the above methodology, you can map any location with what values you want to refer them by.

You can restrict the partitions that are scanned in a query by using the column in the 'WHERE' clause.

```
SELECT dest, origin FROM flights WHERE year = 1991
```

You can also use multiple columns as partition keys. You can scan the data for specific values, and so on.

s3://athena-examples/flight/parquet/year=1991/month=1/day=1/

s3://athena-examples/flight/parquet/year=1991/month=1/day=2/

When deciding the columns on which to partition, consider the following:

- Columns that are used as filters are good candidates for partitioning.
- Partitioning has a cost. As the number of partitions in your table increases, the higher the overhead of retrieving and processing the partition metadata, and the smaller your files. Partitioning too finely can wipe out the initial benefit.
- If your data is heavily skewed to one partition value, and most queries use that value, then the overhead may wipe out the initial benefit.

Example:

The table below compares query run times between a partitioned and Non-partitioned table. Both tables contain 74GB data, uncompressed stored in Text format. The partitioned table is partitioned by the l_shipdate column and has 2526 partitions.

Query	**Non- Partitioned Table**		**Cost**	**Partitioned table**		**Cost**	**Savings**
	Run time	**Data scanned**		**Run time**	**Data scanned**		
SELECT count(*) FROM lineitem WHERE l_shipdate = '1996-09-01'	9.71 seconds	74.1 GB	$0.36	2.16 seconds	29.06 MB	$0.0001	99% cheaper 77% faster
SELECT count(*) FROM lineitem WHERE l_shipdate >= '1996-09-01' AND l_shipdate < '1996-10-01'	10.41 seconds	74.1 GB	$0.36	2.73 seconds	871.39 MB	$0.004	98% cheaper 73% faster

However, partitioning also has a penalty as shown in the following run times. Make sure that you don’t over-partition your data.

Query	**Non- Partitioned Table**	**Cost**	**Partitioned table**	**Cost**	**Savings**

	Run time	Data scanned		Run time	Data scanned		
SELECT count(*) FROM lineitem;	8.4 seconds	74.1 GB	$0.36	10.65 seconds	74.1 GB	$0.36	27% slower

2. Bucket your data

Another way to partition your data is to *bucket* the data within a single partition. With bucketing, you can specify one or more columns containing rows that you want to group together, and put those rows into multiple buckets. This allows you to query only the bucket that you need to read when the bucketed columns value is specified, which can dramatically reduce the number of rows of data to read.

When you are selecting a column to be used for bucketing, we recommend that you select one that has high cardinality (that is, it has a large number of unique values), and that is frequently used to filter the data read during query time. An example of a good column to use for bucketing would be a primary key, such as a user ID for systems.

Within Athena, you can specify the bucketed column inside your Create Table statement by specifying CLUSTERED BY (<bucketed columns>) INTO <number of buckets> BUCKETS. The number of buckets should be so that the files are of optimal size. See the Optimize file sizes section for more details.

To leverage bucketed tables within Athena, you must use Apache Hive to create the data files because Athena does not support the Apache Spark bucketing format. For information about how to create bucketed tables, see LanguageManual DDL BucketedTables in the Apache Hive documentation.

Also note that Athena does not support tables and partitions in which the number of files does not match the number of buckets, such as when multiple INSERTS INTO statements are executed.

The following table shows the difference in a customer table where the c_custkey column is used to create 32 buckets. The customer table is 2.29 GB in size.

Query	**Non- bucketed table**		**Cost**	**Bucketed table using c_custkey as clustered column**		**Cost**	**Savings**
	Run time	Data scanned		Run time	Data scanned		

SELECT count(*) FROM customer where c_custkey = 12677856;	1.53 sec	2.29 GB	$0.01145	1.01 sec	72.94 MB	$0.0003645	97% cheaper 34% faster

3. Compress and split files

Compressing your data can speed up your queries significantly, as long as the files are either of an optimal size (see the next section), or the files are splittable. The smaller data sizes reduce network traffic from Amazon S3 to Athena.

Splittable files allow the execution engine in Athena to split the reading of a file by multiple readers to increase parallelism. If you have a single un-splittable file, then only a single reader can read the file while all other readers sit idle. Not all compression algorithms are splittable. The following table lists common compression formats and their attributes.

Algorithm	**Splittable?**	**Compression ratio**	**Compress + Decompress speed**
Gzip (DEFLATE)	No	High	Medium
bzip2	Yes	Very high	Slow
LZO	No	Low	Fast
Snappy	No	Medium	Very fast

Generally, the higher the compression ratio of an algorithm, the more CPU is required to compress and decompress data.

For Athena, I recommend using either Apache Parquet or Apache ORC, which compress data by default and are splittable.

4. Optimize file sizes

Queries run more efficiently when reading data can be parallelized and when blocks of data can be read sequentially. Ensuring that your file formats are splittable helps with parallelism regardless of how large your files may be.

However, if your files are too small (generally less than 128 MB), the execution engine might be spending additional time with the overhead of opening Amazon S3 files, listing directories, getting object metadata, setting up data transfer, reading file headers, reading compression dictionaries, and so on. On the other hand, if your file is not splittable and the files are too large, the query processing waits until a single reader has completed reading the entire file. That can reduce parallelism.

One remedy to solve your small file problem is to use the S3DistCP utility on Amazon EMR. You can use it to combine smaller files into larger objects. You can also use S3DistCP to move large amounts of data in an optimized fashion from HDFS to Amazon S3, Amazon S3 to Amazon S3, and Amazon S3 to HDFS.

Some benefits of having larger files:

- Faster listing
- Fewer Amazon S3 requests
- Less metadata to manage

Example:

The following table compares query run times between two tables, one backed by a single large file and one by 5,000 small files. Both tables contain 7 GB of data, stored in text format.

Query	**Number of files**	**Run time**
SELECT count(*) FROM lineitem	5000 files	8.4 seconds
SELECT count(*) FROM lineitem	1 file	2.31 seconds
Speedup		72% faster

5. Optimize columnar data store generation

Apache Parquet and Apache ORC are popular columnar data stores. They provide features that store data efficiently by employing column-wise compression, different encoding, compression based on data type, and predicate pushdown. They are also splittable. Generally, better compression ratios or skipping blocks of data means reading fewer bytes from Amazon S3, leading to better query performance.

One parameter that could be tuned is the *block size* or *stripe size*. The block size in Parquet or stripe size in ORC represent the maximum number rows that can fit into one block in terms of size in bytes. The larger the block/stripe size, the more rows can be stored in each block. By default, the Parquet block size is 128 MB and the ORC stripe size is 64 MB. We recommend a larger block size if you have tables with many columns, to ensure that each column block remains at a size that allows for efficient sequential I/O.

Another parameter that could be tuned is the compression algorithm on data blocks. The Parquet default is Snappy, but it also supports no compression, GZIP, and LZO-based compression. ORC defaults to ZLIB, but it also supports no compression and Snappy. We recommend starting with the default compression algorithm and testing with other compression algorithms if you have more than 10 GB of data.

Parquet and ORC file formats both support *predicate pushdown* (also called *predicate filtering*). Parquet and ORC both have blocks of data that represent column values. Each block holds statistics for the block, such as max/min values. When a query is being executed, these statistics determine whether the block should be read or skipped.

One way to optimize the number of blocks to be skipped is to identify and sort by a commonly filtered column before writing your ORC or Parquet files. This ensures that the range between the min and max of values within the block would be as small as possible within each block. This gives it a better chance to be pruned.

You can convert your existing data to Parquet or ORC using Spark or Hive on Amazon EMR. For more information, see the Analyzing Data in S3 using Amazon Athena blog post. See also the following resources:

- Build a Data Lake Foundation with AWS Glue and Amazon S3 blog post
- aws-blog-spark-parquet-conversion Spark GitHub repo
- Converting to Columnar Formats (using Hive for conversion)

Query tuning

Athena uses Presto underneath the covers. Understanding how Presto works provides insight into how you can optimize queries when running them.

This section details the following best practices:

1. Optimize ORDER BY.
2. Optimize joins.
3. Optimize GROUP BY.
4. Optimize the LIKE
5. Use approximate functions.

Bonus tip: Include only the columns that you need.

6. Optimize ORDER BY

The ORDER BY clause returns the results of a query in sort order. To do the sort, Presto must send all rows of data to a single worker and then sort them. This could cause memory pressure on Presto, which could cause the query to take a long time to execute. Worse, the query could fail.

If you are using the ORDER BY clause to look at the top or bottom *N* values, then use a LIMIT clause to reduce the cost of the sort significantly by pushing the sorting and limiting to individual workers, rather than the sorting being done in a single worker.

Example:

Dataset: 7.25 GB table, uncompressed, text format, ~60M rows

Query	Run time
SELECT * FROM lineitem ORDER BY l_shipdate	528 seconds
SELECT * FROM lineitem ORDER BY l_shipdate LIMIT 10000	11.15 seconds
Speedup	98% faster

7. Optimize joins

When you join two tables, specify the larger table on the left side of join and the smaller table on the right side of the join. Presto distributes the table on the right to worker nodes, and then streams the table on the left to do the join. If the table on the right is smaller, then there is less memory used and the query runs faster.

Example:

Dataset: 74 GB total data, uncompressed, text format, ~602M rows

Query	Run time
SELECT count(*) FROM lineitem, part WHERE lineitem.l_partkey = part.p_partkey	22.81 seconds

Query	Run time
SELECT count(*) FROM part, lineitem WHERE lineitem.l_partkey = part.p_partkey	10.71 seconds
Savings / Speedup	~53% speed up

The exception of the rule is when joining multiple tables together and there is the possibility of a cross join. Presto will perform joins from left to right as it does not yet support join reordering. Therefore, you should specify the tables from largest to smallest while ensuring two tables are not specified together that will result in a cross join.

Example:
Dataset: 9.1 GB total, uncompressed, text xormat, ~76M total rows

Query	Run time
SELECT count(*) FROM lineitem, customer, orders WHERE lineitem.l_orderkey = orders.o_orderkey AND customer.c_custkey = orders.o_custkey	Timed Out
SELECT count(*) FROM lineitem, orders, customer WHERE lineitem.l_orderkey = orders.o_orderkey AND customer.c_custkey = orders.o_custkey	3.71 seconds

8. Optimize GROUP BY

The GROUP BY operator distributes rows based on the GROUP BY columns to worker nodes, which hold the GROUP BY values in memory. As rows are being ingested, the GROUP BY columns are looked up in memory and the values are compared. If the GROUP BY columns match, the values are then aggregated together.

When using GROUP BY in your query, order the columns by the cardinality by the highest cardinality (that is, most number of unique values, distributed evenly) to the lowest.

```
SELECT state, gender, count(*)
           FROM census
GROUP BY state, gender;
```

One other optimization is to use numbers instead of strings, if possible, within the GROUP BY clause. Numbers require less memory to store and are faster to compare than strings. The numbers represent the location of the grouped column name in the SELECT statement; for example:

```
SELECT state, gender, count(*)
FROM census
GROUP BY 1, 2;
```

Another optimization is to limit the number of columns within the SELECT statement to reduce the amount of memory required to store, as rows are held in memory and aggregated for the GROUP BY clause.

9. Optimize the LIKE operator

When you are filtering for multiple values on a string column, it is generally better to use regular expressions instead of using the LIKE clause multiple times. This is particularly useful when you are comparing a long list of values.

Example:

Dataset: 74 GB table, uncompressed, text format, ~600M rows

Query	Run time
SELECT count(*) FROM lineitem WHERE l_comment LIKE '%wake%' OR l_comment LIKE '%regular%' OR l_comment LIKE '%express%' OR l_comment LIKE '%sleep%' OR l_comment LIKE '%hello%	20.56 seconds
SELECT count(*) FROM lineitem WHERE regexp_like(l_comment, 'wake|regular|express|sleep|hello')	15.87 seconds
Speedup	17% faster

10. Use approximate functions

For exploring large datasets, a common use case is to find the count of distinct values for a certain column using COUNT(DISTINCT column). An example is looking at the number of unique users hitting a webpage.

When an exact number may not be required—for instance, if you are looking for which webpages to deep dive into, consider using approx_distinct(). This function tries to minimize the memory usage by counting unique hashes of values instead of entire strings. The drawback is that there is a standard error of 2.3%.

Example:

Dataset: 74 GB table, uncompressed, text format, ~600M rows

Query	Run time
SELECT count(distinct l_comment) FROM lineitem;	13.21 seconds
SELECT approx_distinct(l_comment) FROM lineitem;	10.95 seconds
Speedup	17% faster

For more information, see Aggregate Functions in the Presto documentation.

Bonus tip: Only include the columns that you need

When running your queries, limit the final SELECT statement to only the columns that you need instead of selecting all columns. Trimming the number of columns reduces the amount of data that needs to be processed through the entire query execution pipeline. This especially helps when you are querying tables that have large numbers of columns that are string-based, and when you perform multiple joins or aggregations.

Example:

Dataset: 7.25 GB table, uncompressed, text format, ~60M rows

Query	Run time
SELECT * FROM lineitem, orders, customer WHERE lineitem.l_orderkey = orders.o_orderkey AND customer.c_custkey = orders.o_custkey;	983 seconds
SELECT customer.c_name, lineitem.l_quantity, orders.o_totalprice FROM lineitem, orders, customer WHERE lineitem.l_orderkey = orders.o_orderkey AND customer.c_custkey = orders.o_custkey;	6.78 seconds
Savings / Speedup	145x faster