Part 1 - Measuring Query Runtimes Accurately

SQL servers can use memory to make the second run of a query faster than the first run. There are two ways that SQL Server can use memory to make a query faster the second time you run it.

1- **It takes CPU time to figure out how to run a query. SQL Server uses memory to cache execution plans to save time the next time you run the query.**

The first time you run a query using the view, SQL Server has to 'compile' an execution plan to figure out the best way to run the query. SQL doesn't compile the plan when you create the view– it compiles a plan when you first use the view in a query. After all, you could use the view in a query in different ways: you might select only some columns, you could use different 'where' clauses, you could use joins. It doesn't matter too much that you're using a view. When you run a query that references it, behind the scenes SQL Server will expand the TSQL in the view just like any other query, and it will look for ways to simplify it.
SQL Server is designed to store the execution plan for a query in memory in the execution plan cache, in case you run it again. It would be very costly for the CPUs to generate a plan for every query, and people tend to re-run many queries. If you re-run a query and there is already an execution plan in the plan cache, SQL Server can use and save all that compile time.
Restarting the SQL Server, taking the database offline and online, memory pressure, and many server level settings changes will also clear execution plans out of cache, so you must wait for a compile.

2- **It takes time to read data from disk. SQL Server uses memory to cache data in the Buffer Pool, so it doesn't have to go to disk the next time you use that data.**

The first time you run the query it may be using data that is on disk. It will bring that into memory (this memory is called the "buffer pool"). If you run the query again immediately afterward and the data is already in memory, it may be much faster – it depends how much memory it had to go read from disk, and how slow your storage system is. One difference with this type of memory is that your buffer pool memory is not impacted by ALTERING the view. SQL Server does not dump data from the buffer pool when you change a view or procedure.

Instead, it keeps track of how frequently different pages of data are used, and ages out the least useful pages from memory over time.

How is Redshift returning the results without running the queries?

To reduce query execution time and improve system performance, Amazon Redshift caches the results of certain types of queries in memory on the leader node. When a user submits a query, Amazon Redshift checks the results cache for a valid, cached copy of the query results. If a match is found in the result cache, Amazon Redshift uses the cached results and doesn't run the query. Result caching is transparent to the user. Result caching is turned on by default. To turn off result caching for the current session, set the enable_result_cache_for_session parameter to off.

Results table when cache in on:

	Elapsed time (first run) milliseconds	**Elapsed time (second run)** milliseconds
Query 1	17648	13
Query 2	650	11
Query 3	37443	10

Results table when cache in off:

	Elapsed time (first run) milliseconds	**Elapsed time (second run)** milliseconds	**Elapsed time (Third run)** milliseconds
Query 1	17648	17874	17182
Query 2	650	500	472
Query 3	36378	36003	34254

Part 2 - Optimizing Queries

Query 1

Considering the join strategy, DS_BCAST_INNER, we executed below query to find out table distribution key:

```
select * from svv_table_info where "table" like 'customer%' or "table" like 'order%' or "table" like 'lineitem%'
```

database	schema	table_id	table	encoded	diststyle	sortkey1	max_varchar
dev	public	107965	customer	Y, AUTO(ENCODE)	EVEN	AUTO(SORTKEY)	117
dev	public	107967	orders	Y, AUTO(ENCODE)	EVEN	AUTO(SORTKEY)	79
dev	public	107973	lineitem	Y, AUTO(ENCODE)	EVEN	AUTO(SORTKEY)	44

That the distribution key is not aligned with the join condition of the query so, we have changed the distkey for the orders table and lineitem table from ‘EVEN’ to ‘orderkey’:

```
alter table public.orders alter distkey o_orderkey

alter table public.lineitem alter distkey l_orderkey
```

After getting the execution details we came up with these results:

2022-02-21 02:12:03.346...	2022-02-21 02:12:06.151...	2805	30000000	0
2022-02-21 02:12:03.346...	2022-02-21 02:12:06.137...	2791	0	0
2022-02-21 02:12:06.181...	2022-02-21 02:12:07.360...	1179	38512750	113037849
2022-02-21 02:12:06.181...	2022-02-21 02:12:07.360...	1179	38512750	0

The number of 113,037,849 rows has scanned so, to improve the performance, we could sort the data by l_commitdate

```
alter table lineitem alter COMPOUND sortkey (l_commitdate);
```

We re-ran the query again, and Redshift scaned much less data of 38,629,736:

2022-02-21 02:34:16.375...	2022-02-21 02:34:19.139...	2764	0	0
2022-02-21 02:34:16.375...	2022-02-21 02:34:19.158...	2783	30000000	0
2022-02-21 02:34:19.186...	2022-02-21 02:34:20.386...	1200	38512750	38629736
2022-02-21 02:34:19.186...	2022-02-21 02:34:20.386...	1200	38512750	0

Run time before optimization milliseconds	**Runtime after optimization** milliseconds
17648	9470

Query 2

I wrote the query with ‘Where’ statement for achieving the same results with lower elapsed time. Because “Where" restricts results before bringing all rows.

```
SELECT COUNT (* )
FROM lineitem
where L_SHIPDATE between '1992-07-05' and '1992-07-07'
```

Run time before optimization milliseconds	**Runtime after optimization** milliseconds
650	96

References

Little, K. (2016, November 25). *Why is My Query Faster the Second Time it Runs?* Retrieved February 19, 2022, from https://www.littlekendra.com/2016/11/25/why-is-my-query-faster-the-second-time-it-runs-dear-sql-dba-episode-23/

AWS. (2022). Amazon Redshift. https://docs.aws.amazon.com/redshift/latest/dg/r_enable_result_cache_for_session.html