Introduction to the Fundamentals of Amazon Redshift

Max Ganz II @ Redshift Research Project

7th April 2023

Abstract

This is a short, punchy document written to present the critical, central issues involved in Redshift use to a weakly technical (or better) reader. It is not lengthy, highly structured, completely thorough or detailed, but rather directly conveys the key issues as concisely and approachably as possible.

Contents

Introduction

This document is a short, direct, punchy introduction to the fundamentals of Amazon Redshift.

It's written for weakly technical readers - if you know what queries are, and a bit about files and disks, and what processes are, you'll be fine.

This document will enable you to know when you should, and when you should not, use Redshift.

I find practically all users come to Redshift having had experience of conventional relational databases are expecting the same, "but bigger".

This is completely, totally, utterly and absolutely incorrect.

It is critical to realise that although Redshift offers SQL, it is under the hood utterly unlike conventional relational databases, and as such its behaviour, characteristics, failure modes and use cases are *utterly* different.

I am of the view about 95% of clients using Redshift should not be, and should be on a different database, and the reason they get away with using Redshift is because they do not in fact have Big Data; they have small data, and as such as in a position where the hardware overwhelms the data and so they can do pretty much anything and so it all runs in a second or two and no is any the wiser.

Once you start to increase data volumes, or if you use Redshift in ways with which it is particularly unsuited for (real-time interactive applications, small data, rapid data, many concurrent queries, loading data continually to a large table, and so on), then hardware isn't enough and the problems become visible.

Setting the Scene

Redshift is what is known as a sorted, column-store, relational database.

Now, there are in fact only a very few fundamental types of database - about five or so. There's any number of databases out there - Postgres, mySQL, CockroachDB, Cassandra, neo4j, Vertica, Redshift of course - but they're all one of the very few fundamental types, and it is that fundamental type which defines the capabilities of the database and so what you can use it for.

The fundamental types are;

1. map-reduce
2. key-value (graph is a sub-type of key-value)
3. relational (which has four sub-types - a relational database must be row-store or column-store, and must be sorted or unsorted, so two choices with two options each, which gives four sub-types)

That gives six, as the four sub-types of relational are utterly different in the use cases they are viable for; they are in effect different types of database, and so must be considered separately, even though they all offer SQL and so, superficially, appear identical.

Now, when you as a software engineer decide one day to sit down and write a database, you find you are immediately presented with a series of design choices. For example, is your database going to offer SQL, or not? it's necessarily one or the other; it cannot be both. You might then consider, am I looking to support Big Data, or not; for if you are, there are then a range of database methods you now cannot use, because they do not scale. Then it might be, will the database run on only a single machine, or can it run on a cluster? and so on.

In other words, there exists a tree of choices, and every choice you make picks a branch, and takes you to the next choice on that branch.

There are then as you can imagine in fact zillions of types of databases - one database for every possible combination of choices - *but* only a few of them make any sense; these few are orthogonal, as each performs a distinct type of work which cannot be performed by any of the others, and these are the optimal sets of choices, as all of the other sets of choices, when compared to one of these few, are inferior.

This is why there are only the few fundamental types of database.

When you come to select a database for a project, you need to understand the capabilities of the fundamental types, to make the correct choice.

You must also remember that not everything *can* be done. You cannot, for example, have Big Data and SQL and have large volumes of data being constantly loaded into the database. There is no way to do this.

Not one database vendor, at least, those who charge you money, will tell you what their database *cannot* do. All database vendors are selling products which can do everything - except, of course, they're not.

My experience with AWS in this regard is that *everything* AWS publish about Redshift, in the docs, in the blogs, in the support docs, when you talk to Support

and particularly when you talk to technical account managers (TAMs) obfuscated all weaknesses, is relentlessly positive, everything is win-win, and Redshift can do everything.

No matter what use case you approach a TAM with, the answer will be yes.

I regardless *all* information from AWS about Redshift as safe only if you already know what's really going on, so you can see through it; otherwise, it will mislead you.

Row-Store vs Column-Store

To begin with, we'll deal with row-store vs column-store, because it's nice and simple, and we'll need to understand this for later on.

Basically speaking, in a row-store database, a table is a single file, the file consisting of one row after another.

In a column-store database, each column in a table is a single file, and although of course the table still consists of one row after another, the rows have been split up into their columns, and each column is stored, one after the other, in its file.

Now, historically, all databases were row-store. It was easy to implement, and sufficient for how databases were being used.

The drawback of row-store is that you're forced to read all the columns - imagine you have a row with 20 columns, but you're only interested in one column, say, an ID column of some kind. You need to scan the table - which is to say, read the file - but where the rows are stored one after the other, you are having to read *all* the columns, not just the one column you're interested in. Lots more disk I/O.

(Of course you can seek to and read from any byte in a file, so you could in theory seek from ID column to ID column, but in fact seeking to a location is mind-bendingly slow, while performing sequential reads, byte after byte, is very, very fast. It's profoundly faster to get to the next row by reading all of the current row, than by seeking to the ID column in the next row. Also, it might be some columns are variable length, so you don't necessary know in an easy and straightforward way where you would seek to, to read the next ID column.)

With column store, a query reads only the columns which are actually being used in the query.

That's great, but you have to then consider this means the database now has to join the columns back up, when it comes to return rows to the user.

This process is called rematerialization and it's not free - and in fact, it can be operated in ways which make it problematic - I can't explain how so, yet, because there's quite a bit more I need first to explain about other ways in which Redshift works - but I'll come to it.

However, I can give you one simple example.

Imagine you issue a query which is a `SELECT *`. This means you're going to return all the columns in the table to the user. In this case, you would have been better off with row-store - the rows are already joined up - than in column store, where you're going to read all the columns anyway, but now you also have to do the work to rematerialized the rows.

One other consequence of column store is that updating a single row becomes considerably more costly.

With row-store, you mark the original row as deleted and append the new version to the table.

With column store you do the same, but now you have to repeat that work once for each column. If you have a wider table, say 20, or 50, or even 100 columns, that's a lot of work, because it means two disk seeks per column, and that's expensive.

If you're working with large data volumes (Big Data), column-store is the way to go, because the saving in disk I/O by not reading unneeded columns becomes huge where you're reading so many rows; but if you're working with single rows, row-store is the way to go, because it's simple and fast to manipulate single rows.

Sorted vs Unsorted

Relational databases can be sorted or unsorted, and sorting is the key and central method by which Big Data is supported with SQL.

Historically, relational databases have all been unsorted. Postgres, MS-SQL, mySQL, you name it - all unsorted.

What this means is that the database imposes no ordering upon the rows in a table; rows are in any order at all. When new rows are added, they are simply appended to the table, in the order they arrive.

Then in about 2015 or so sorted relational databases began to make an appearance in the market (they'd existed in academic research projects for much longer).

As you can envision, a sorted relational database *does* impose ordering upon the rows in tables.

For a table, one or more columns are selected, and the table is ordered by the values in those columns (so the table is sorted first by all values in the first sorting column, then all rows with identical values in the first column are sorted by the second sorting column, and so on).

Now, unsorted relational databases have the very annoying property that the time taken to retrieve a row from a table increases as the number of rows in the table increases.

This happens because the database has no idea where rows are in the table. If you have a table which has, say, one row for every person in Switzerland, with a column for name, and you want to find the row for "Max Ganz II", the database has no idea where that row is and so has to scan the entire table.

The larger the table, the longer it takes, even though you're still only trying to retrieve a single row.

This is why unsorted relational databases cannot handle Big Data; if you put a petabyte in a table, come back in a week.

(If you had column-store, that would help, but it doesn't fundamentally solve the problem - you still have to scan all of the rows in all of the columns you're using.)

Sorted relational databases - but when and only when sorting is operated correctly - do *not* possess this property; but when operated incorrectly, they *do* possess this property, and then you're back to square one.

This is why sorted relational databases can handle Big Data.

The way it works is this; let's imagine we have again that table for every person in Switzerland, only now we sort it, by name.

So all the rows are now alphabetically ordered, by name, within the table. All the As, then all the Bs, Cs, and so on.

Now here's the magic - which is known as *min-max culling* - what Redshift does is keep track of the minimum and maximum value for each one megabyte block

of rows, and these minimum and maximum values are kept in memory, so they immediately available at all times.

Now, as you can imagine, sorting makes the minimum and maximum values very close together. When we now come to search for "Max Ganz II", Redshift can *immediately* see that a huge number of these one megabyte blocks simply *cannot* contain that name, because a name beginning with "M" cannot be in a block which ranges from say, "Caroline" to "Casey".

As such, Redshift reads and only reads the blocks which *can* contain the value being searched for, and the number of blocks depends on the data you have, *not* the number of blocks, or rows, in the table.

Even better, those blocks which could contain the name are contiguous, so there's no seeking - remember seeking is desperately slow, while reading contiguous data is very fast.

This is why sorted relational databases can support Big Data; queries retrieve only the rows which they need to do the work they have to do, and the time taken to retrieve those rows is not affected by how many rows are in the table.

However, I've been writing - constantly :-) - that sorting must be operated correctly.

So imagine now we have this table of all the people in Switzerland, and we add a column, which is their age.

However, we still sort only by name.

Now we come to issue a query and we ask for everyone who is 25 years old.

Well, now we have a problem - the age column is *not* sorted - and in fact, where we sorted by name, we've basically randomly sorted the age column, and so every block will have pretty much the full range of values.

If I ask Redshift to find all the people aged 25, it's going to need to scan pretty much every block in the table - and in fact, we're back to exactly the same situation an unsorted database is in. The time taken to retrieve rows once again depends on the number of rows in the table.

So what we see that when we choose the sorting order of a table, we are in fact defining the set of queries which will run in a timely manner, and also the set of queries which will *not* run in a timely manner.

As such, when you come to design your tables, you need to select sorting orders which will handle your current set of queries, the set of queries you know you will be getting in the near future, and the general expected line of development in the future.

What's more, it also means both the table designers and the table *users*, the data analysts and so on, *all* need to understand about sorting.

If the table designers come up with some brilliant design, and set the sorting orders appropriately, and then the users have no clue about sorting and do whatever they like, then sorting is not going to be correctly used.

If sorting is not being used correctly, Redshift is under the hood behaving exactly like Postgres *et al*, except Redshift has the cluster and the usual row-store databases are not clustered (but there are some which are, like Postgres-XL or Exasol), and Redshift does not have indexes but the usual row-store databases do, and the usual row-store database have a metric *ton* more functionality than Redshift.

Another emergent property of sorting is that changing tables is a lot of work and so doesn't much happen, but getting new and unexpected queries is easy and happens all the time, and so sooner or later the tables you have are no longer viable at all for the queries you're actually issuing, and you need to redesign your tables.

Clusters

Historically, databases have run on a single computer. It was simple to implement, and met the needs of the day.

These days, databases which are intended for Big Data run on clusters, because no single computer is going to be able to handle Big Data in a timely manner.

Redshift is a clustered database, and its design has a single leader node, which organizes everything, and then from 2 to 128 worker nodes, who do most of the heavy lifting.

(Note that you can start single node Redshift clusters; never do this. A single node cluster is a leader node and a single worker node jammed into a single virtual machine. It's completely off the map, intended purely to allow people to start a cluster to have a look at it, and I wouldn't touch it with a barge pole for *any* kind of actual work *at all*.)

The nodes are virtual machines, just like EC2 virtual machines.

The leader node controls and directs the work performed by the cluster. It receives queries from the user, then decides how to perform the queries, and then uses the worker nodes to get the work done, and then returns the results to the user.

When you start a cluster, you select a node type (the specifications of the node - processor, memory, etc) and every node is that type of node, including the leader node.

You can immediately see an issue here - you can scale the worker nodes by adding more of then, but you cannot scale the leader node, because there's only ever the one.

In practise, unless you operate Redshift incorrectly, this is usually not a problem. I'm not worried about it - if you're operating Redshift incorrectly enough you're hammering the leader node, you have much bigger problems to worry about.

Now for a history lesson about where Redshift came from.

Back in about 2008 a company called ParAccel branched PostgreSQL 8.0.2 (released in 2005) to develop a product first named Maverick and then named ParAccel Analytic Database, ParAccel for short, where these new products were sorted column-store relational databases.

About 2010, AWS were shopping around for a scalable database, and the rumour goes what AWS were offering wasn't great and they found no takers.

However, as it that may or may not be, Amazon invested about $20m in ParAccel in 2011, and a year or so afterwards purchased for a single lump sum an unlimited license to the ParAccel code-base, and from this code-base, Amazon developed Redshift. ParAccel had no involvement with Redshift or AWS beyond selling the license, and they were in time bought by Actian, a graveyard where small database companies go to die.

The upshot of this is that Redshift looks like ParAccel, where ParAccel has a leader node which is Postgres 8.0.2 plus the new code to manage worker

nodes, and so which comes with the enormous wealth of functionality provided by Postgres 8, but where the worker nodes are *not* derived from Postgres, but which are wholly new, and rather bare, minimal things.

Now, the leader node is just the leader node, but worker nodes are divided up into what are known as *slices*.

The docs do not explain what a slice is, and to explain it, I have to first explain a bit about queries.

Users issue SQL queries to Redshift. These queries are sent to the leader node, which converts them into multiple C++ files, compiles these files into binaries, and the binaries are sent out to the worker nodes, which execute the binaries.

The binaries then are being executed, on all the worker nodes, and they perform the work of the query - they are the query - and the results are returned to the leader node, which in turn returns them to the user.

Now, what actually happens is that the complete set of binaries which is a query is actually run *multiple* times - and each complete set of binaries is a slice.

Each slice possess its own portion of each table, and the binaries which are that slice are running the query on its portion of each table.

All this is done to allow improvements in performance by parallelism. When one slice is blocked waiting for say a block from disk, other slices are getting work done.

(You might consider though at this point that you can have up to 50 queries running concurrently, and a query might have say 10 processes running concurrently, and a big node might would have 16 slices. That's 50 * 10 * 16 = 8,000 processes. That's a lot, and we might begin to think swapping between processes is getting expensive.)

Okay, so now we understand slices, back to tables.

Tables as mentioned are distributed across slices.

What this means is that any single row is stored on a single slice only, and when rows are added to tables, there is a mechanism (defined when the table is created) to decide which slice receives each row.

The usual method is to select a single column, and all rows with the same value in that column go to the same slice.

A critical issue here is ensuring that each slice holds the same number of rows, or as close to that as possible.

The concern here is that queries execute concurrently on every slice, where a query can only complete when all the slices have completed the work of the query on the rows held on that slice.

For example, if I ask for the average age of all the people in Switzerland, and every slice holds a part of the table of all people in Switzerland, obviously that query cannot return an answer to the user until all slices have processed their part of the table.

So we see that if a single slice has, say, twice as many rows as the other slices, it will take twice as long to complete - so the fact the other slices complete in half the time as the slow slice is of no benefit at all, because you still have to wait for the slowest slice to complete.

The upshot of this is that queries are as slow as the slowest slice.

Disk

In Redshift all disk storage is in one megabyte blocks.

Redshift *always* reads and writes in one megabyte blocks (just *block* or *blocks* from now on).

You *never* read single rows, or sets of rows; you read single blocks, or sets of blocks.

Remember now also Redshift is column-store, and so each column is stored in its own file.

If you read one row, you're reading a block from the file for every column.

Now imagine we have a table with 20 columns, and we write a single row.

The row goes to a single slice, and each column goes to its own file, and so Redshift writes a block for each column; 20mb of storage and 20mb of disk I/O for what might be 1kb of actual data.

Imagine now we have a cluster with say 40 slices, and we write 40 rows, and each row goes (as it should - tables should be equally distributed over slices, which I'll get to when I talk more about queries) to a different slice.

We've just written 20mb to 40 slices, which is 800mb.

That's 800mb for what is probably 40kb of actual data.

In fact, it's kinda worse than that.

Remember that each column is a file? well, where Redshift is sorted, each column is actually *two* files. One file for the sorted blocks, one file for the unsorted blocks.

When a table is new, or truncates, both files do not exist; the table uses zero disk.

When the first row(s) are written, both files are created, so when we write the very first time, we're actually writing *two* blocks (assuming all the rows fit into a single block), so in our example case of 40 slices and 20 columns, we've used 1.6gb of disk - for what is probably about 40kb of actual data.

Not what you expect, and you won't find it in the docs, and I'll be *very* surprised if you can get a TAM to tell you, unless you tell him first :-)

Now, it used to be, if you then wrote another set of 40 rows, you would again use another 800mb of disk - remember that blocks are immutable, so the existing block isn't touched, and so you're writing new blocks for your new row - and you would *very* rapidly run out of disk.

However, what seems to happen now is that Redshift rapidly merges these new blocks into the existing blocks. If I monitor a table in real-time, and insert single rows, I see the expected block-per-column-per-row, and that builds up for a short tine, but then it disappears, and this is the steady-state behaviour.

So it appears okay, but I think that's using a *lot* of disk I/O, although of course this is a lesser evil than using a lot of disk, but in any case, you shouldn't be

doing this. It is not a suitable use case and will not scale.

Be clear that the use of one megabyte blocks is absolutely sound if you're working with Big Data - you have at least a terabyte of data - because it's efficient and so is the right design choice for Redshift's intended use case.

If you want to do lots of single row operations, you should not be using sorted column-store. If you're doing just one or two, it'll be okay.

`VACUUM`

Now we know that columns are actually two files, one sorted and one unsorted, I can talk about how data gets loaded into tables, because newly loaded data is not sorted, and needs to be sorted.

When a table has just been created, or truncated, it has no rows and in fact has no files - they do not yet exist.

When you add rows to a table, which is by an `INSERT` or `COPY`, the set of incoming rows is sorted with respect to itself, and then appended to the unsorted files. The reason the rows are sorted (with respect to themselves) is to try to make min-max culling more effective on those blocks.

There is an exception to this.

This is when the table has just been created, or truncated; the first `INSERT` or `COPY` is added to the *sorted* files. (Remember, all sets of incoming rows are sorted with respect to themselves, and in this case, with an empty table, we are safe to add these rows to the sorted file).

So now we can imagine this process continuing for a while, and a table building up a whole bunch of unsorted data.

To sort all the unsorted data with respect to the table as a whole, we need to issue a command on that table, which is the `VACUUM` command.

The `VACUUM` command takes the sorted data, and the unsorted data, and sorts it all, compacts any blocks which are not fully occupied with rows, places it all into the sorted files, and the unsorted files are then empty - the table has been sorted.

Now, here's the kicker.

You can run only one `VACUUM` at a time, and that's **PER CLUSTER**.

Not per database, not per table. Per *cluster*.

What this means is that `VACUUM` is a scarce resource; you have 24 hours per day of vacuum time, and that's it.

In fact, you are in a producer-consumer scenario. You are constantly producing unsorted blocks, by adding or modifying records, and consuming them, converting them to sorted blocks, with `VACUUM`.

When the time comes you are producing blocks more quickly than you can consume them, the cluster ultimately degenerates to a fully unsorted state and it's game over.

In fact, when this happens, what you have to do is spin up a second cluster, to get another day's worth of vacuum time, and move part of your workload to that cluster. Not easy or convenient for a system expecting in its design to run on a single cluster.

This also means you need to co-ordinate `VACUUM` use across all users on a cluster, across all the teams and users and databases, because if a `VACUUM` is running and someone else tries to run `VACUUM`, the second user's `VACUUM` simply aborts - but users are going to want to use `VACUUM` at certain times, and what's more, `VACUUM` can take a very long time.

On a reasonably sized cluster (say a dozen `dc2.8xlarge`), something like a few terabyte table takes something like two days to vacuum.

You can now understand the critical limitations sorted relational databases face when it comes to loading data.

Tables *must* be sorted, for min-max culling to be effective, but sorting is constrained; you can sort one table at a time, and as the table becomes larger, sorting takes longer.

We can imagine, for example, a use case where we want to be as near real-time as possible, so we're constantly loading data into a table, but we also want to perform timely searches on this data.

The data being loaded for searching to be timely requires sorting into the table by `VACUUM` - but the more time we've spent loading data, the bigger the table becomes, the longer `VACUUM` takes.

You can see this doesn't end well.

The basic issue here is actually a fundamental property of all databases - the more work you do when loading data, the faster queries go when you run them.

There are databases (such as map-reduce) which do *zero* work on loading, and have the slowest possible queries.

Redshift (and all the sorted column-store relational databases) are the far opposite end of the continuum, where they do the maximum possible work upon loading, and (when operated correctly) have the fastest possible queries.

(Note there is an auto-vacuum in Redshift, but it looks to be wholly ineffectual. The problem seems to be it runs too infrequently, and when it does run, it stops running too easily, when it sees the cluster is busy running user queries. So you have to tackle this yourself. If you ask a TAM about `VACUUM`, he'll tell you auto-vacuum does it for you and you don't need to worry about it - but of course auto-vacuum, even if it did work, cannot solve the problem of trying to constantly vacuum a large table - which is an example of what I mean about everything you hear from AWS being misleading, unless you already know what's going on.)

Column Encodings

Where each column is a file, and has the same data type, and usually data of much the same character, data compression, which is on a per-column basis,

tends - when an appropriate compression method is chosen for the data in hand - to be highly effective.

You specify encodings when you create a table, but you can leave it to Redshift to select encodings.

Be aware that Redshift makes extremely poor encoding choices.

When I take a table Redshift has configured, and hand-select the encodings, I normally save a bit more than 40% disk space. That's an engineers 40%, too, so really actually 40%.

That 40% is a lot of money you're spending on your cluster, and a lot of performance being lost.

One final note - the correct encoding choice depends absolutely on the sorting order of the table, so you have to select the sorting order first, and if you ever change it, you need to re-select the encoding choices.

Queries

Redshift is designed to run only a few queries at once - typically 5 to 10.

To give some colour to this, Redshift has an absolute maximum of 50 concurrent queries, no matter what.

Redshift is designed to scale primarily by the staggering efficiency which is obtained through the correct use of sorting; when sorting is used correctly, queries are blazingly fast, and so the fact you can run only a few queries at once just doesn't matter.

Scaling is secondarily achieved by hardware; adding worker node to the cluster. This though is expensive (double performance, double your Redshift bill), has limits (maximum of about 100 nodes in a cluster), and also not quite everything scales with cluster size - one or two behaviours become *slower* as the cluster becomes larger; in particular *commits* become slower as the cluster becomes larger, and commits are serialized over the cluster as a whole.

When a query adds new blocks, or modifies existing blocks, none of that new data or those changes are visible to any other query, until the adding/modifying query *commits* those changes, and that happens at the very end of the query running.

Commits are serialized over a cluster; there's a queue of outstanding commits, and one commit happens at a time. Where they tick out, one after the other, there's no handling for burstyness; if you have a sudden flood of queries which want to commit (maybe that's how your ETL system works), they're only going to be serviced one at a time, so you can end up with large delays.

It's fabulously simple to commit if the database runs on a single node, and fabulously complicated if the database runs on a cluster, because the cluster needs to be able to handle the situation of a node dying while a commit is occurring, without inducing data corruption.

The more data has been added, or modified, the longer the commit takes. The larger the cluster size, the longer the commit takes. (The size of the tables being committed to makes no difference to anything.)

To give a sense of the times involved, on a two node `dc2.large` cluster (the smallest possible normal cluster);

1. Committing 40mb takes about 0.2 seconds
2. Committing 400mb takes about 3.5 seconds (10x data, 15x time)
3. Committing 800mb takes about 7.4 seconds (20x data, 37x time)

Redshift is not designed for large amounts of data to be frequently added.

Redshift is also not designed for small amounts of data to be frequently added to large tables, because of how long `VACUUM` will take.

Adding new hardware helps almost everything (but not the leader node, of course) and so it the query will generate and write all its new or modified data much more quickly - but the final stage, the commit, will go slower.

Coming back to queries, if sorting is being used incorrectly, you'll get whatever performance you get from the hardware you have, and if your queries are running slowly, then obviously you can have a problem; you can start developing a backlog of queries waiting to execute. Again, a producer-consumer scenario; do not produce queries more quickly than you consume them.

The reason there are only a few concurrent queries is because, broadly speaking, queries need plenty of memory to run in a timely manner - there's nuance here though, which again relates to using sorting correctly or incorrectly (correct use of sorting means queries need practically no memory at all, and you really could run 50 queries concurrently with no problems at all), which is covered when I come to describe joins.

What is always true though is that if a query has too little memory for the work it is doing, then it begins to swap to disk, and that hammers the disk - and disk is the key, primary shared resource which everyone on the cluster is using. Start hammering the disk, and the cluster slows down for everyone.

So you can see the problem - you can run more queries concurrently, but they necessarily have less memory each, and so your actual throughput will be *less*, because queries are swapping to disk due to lack of memory, and so *everyone* is going a lot slower.

AWS a few years ago added to Redshift, what to my mind is an expensive ameliorative functionality to help users using sorting incorrectly and so issuing lots of slow queries, called Concurrency Scaling Clusters (CSC for short).

What it is, is that when your cluster is full and queries are waiting, another cluster is automatically brought up, and begins taking queries from the queue.

The cluster takes a few minutes to start up, has no data when it begins, and is pulling data down from the cloud-based copy of the data in your main cluster (all Redshift clusters keep a copy of their data in S3 or something like it, for k-safety - which is to say, to ensure no data is lost when a node fails), which is a lot slower than using local disk.

The problem with this is that CSC clusters are billed at the per-second rate.

You can have up to ten of these per-second rate clusters.

This is a staggeringly expensive way to cope with operating Redshift incorrectly.

You would do infinitely better to use a different database technology - a cluster based unsorted row-store would do just fine - which *can* run lots of queries at once.

Query Compilation

As previously described, SQL queries are sent to the leader node, where they are converted into multiple C++ files, each of which is compiled.

Compilation takes time, and the query cannot start executing until compilation has finished.

Compilation means queries take longer to start, but run more quickly, which is absolutely the correct choice for a database intended for Big Data, but it's

absolutely the wrong choice for customer-facing interactive use. For those use cases, you want a database which does not compile, such as Postgres or Postgres-XL.

To talk more about compilation, I need now to explain about how Redshift organizes the work done by queries.

Redshift when it receives the SQL for a query organized the work to be done in terms of three concepts, *streams*, *segments*, and *steps*.

The work for a query is first broken up into streams. Streams execute serially. When a stream finishes, its output goes to disk, and is loaded by the next stream.

Streams are composed of segments - and a segment is in fact a single process, the output of a C++ file - and all the segments in a stream execute concurrently.

Segments basically form the nodes on a graph, where the graph is a pyramid shape, with the base nodes doing things like reading rows from a table or network, with the rows then flowing upwards, into the middle nodes, which are doing processing work on those rows, like filtering or aggregation, until finally rows reach the top of the graph, which is the output of the stream.

(The final stream passes its output back to the user - these are the actual results of the query.)

Steps are the atomic operations of work which Redshift can perform - scanning a table, aggregating rows, etc.

When Redshift is deciding how to perform the work required for a query, it composes segments from the available steps.

Now, a simple query will have say two streams and two or three segments in total.

A normal query might have four streams, and ten or twelve segments.

A complex query might have as many as ten streams, and even fifty segments.

A BI application query, or indeed from any software with an automatic SQL generator, where such generators usually produce logically correct but enormous and amazingly complex SQL, might have twenty streams and a hundred segments.

Each segment needs to be compiled.

Now, back in the old days, way back when, all compilation was performed by the leader node.

Queries compiled in parallel, but any given query compiled a single segment at a time.

A compile on a fully unloaded leader node would take about two to five seconds, depending on how complex the segment, but a loaded leader node would take proportionally longer.

You can then imagine a BI application issuing a query with most of a hundred segments taking 500 seconds just to compile - even before running - which is most of ten minutes.

AWS then added a per-AWS-account cache; if a segment was needed which had been compiled, it would be taken from the cache.

The cache is very hit and miss, in my experience.

For example, with one client, we issued an `UPDATE` one hundred times, each updating a single row, one after the other. All that changes were the values being updated.

48 of the queries compiled, 52 were serviced by cache, and so it took about 4.5 minutes.

However, the client informed me exactly once they had seen it run through in about seven seconds - which would mean all the compilations had come from cache.

In any event, the cache is per-region and per-Redshift version (and also per connection method - ODBC vs JDBC vs native Postgres), so when your Redshift cluster is updated, you're facing an empty cache. Your system still needs to be able to cope with that situation (and there's no way to test for it, as you cannot disable the cache).

Fast-forward some years, and AWS introduce what is to my eye the best improvement they've made to Redshift; compile off-load.

With compile off-load, there's a 4 to 8 second period at the start of a query where the leader node distributes the compilation to work to helpers, and then after that period, the helpers very rapidly (milliseconds) return all the compiled segments to the leader node.

Bingo! now we have at most, for any query from any source, about 4 to 8 seconds of compile time. That's fast enough for customer-facing interactive use, if we're using queries which execute quickly.

Except…

About a year ago, the devs changed how off-load was being used.

Originally, there was one compilation, at the start of a query.

Now there can be one *per stream* - which means the 4 to 8 seconds can happen *multiple times*, and of course, the more complex the query (BI applications), the more likely this is to happen.

Customer-facing interactive use is once again off the menu.

So the upshot of all this is that there is usually (unless every segment is found in the cache, which you absolutely cannot expect) a significant and irreducible delay for compilation before queries start, which obviously impacts viable use cases.

BI Applications

AWS, Tableau, and every BI application out there will tell you they run on Redshift and you can use their product on Redshift (and would you very kindly now buy some licenses).

It is true - they all do run on Redshift, they all use SQL and Redshift supports SQL - but your actual *experience* is much more nuanced.

The first issue is query compilation.

I think most of these tools (I know mainly about Tableau) can connect directly to a database and issue queries directly, as the user operates the dashboard. This isn't going to fly, because of compilation delays, but also because the queries generated by BI applications know nothing about sorting and also are usually terribly inefficient. They take a long time to run anyway, for the data volume they process.

(If you have small data, query efficiency is irrelevant, because the hardware overwhelms the data; you can do anything you like, and it all runs in a second or two. Once you start to get more data, that's when you discover it does not scale.)

Now, I think again most of these tools allow you to issue a query which extracts all the data a dashboard could need (copies of the tables in use, basically) and store them in the BI application server, and the dashboard runs from that server.

This solves the problem of query compilation, but where queries generated by BI applications are usually very inefficient, and the data volume being brought out of the server can be very large, these extract queries can run for a long time - hours.

Remember here that Redshift allows only a few queries to run concurrently. You're using one of those scarce query slots for a long time; and also using up one of your BI developers, who is usually sitting there, doing nothing, until the extract completes.

You can work around this by not allowing the BI application to generate queries - you define your own query and make that the single data source for a dashboard. With a bit of luck, the BI application will simply run that query directly. You've lost a bunch of flexibility, but there's no other way.

There's still a further problem, though, which is that BI applications themselves are *not* Big Data capable.

As such, you can't load too much data into them, so running them on Big Data tables is no-no *anyway*.

So, all in all, the only solution I see to all this is that the BI application has a single data source defined, which is your hand-written query, which generates about 100k rows, which is loaded by an extract query into the BI application server and a dashboard uses that and only that table.

This gives you an efficient query (no hogging a slot for hours), avoids query compilation (it only happens once), produces a small enough number of rows that shifting them over the network is no problem, and also loading them into the BI application itself is no problem.

Not really what you expected, and I would say not what was communicated to you when AWS and Tableau told you that you can use Tableau on Redshift.

Joins

When you query a relational database, you are doing two things : you select columns and rows, and you perform joins.

Now, I've written already about what happens with single tables and sorting and querying columns and rows; you need to use min-max culling correctly to get performance.

We must then also see that there is a way in which sorting enables timely *joins*, because if it does not, then we're dead in the water - we would still do not have a way to build relational databases for Big Data, because joins are essential to SQL.

As you may expect, given that Redshift exists, there is indeed a way to use sorting to make joins timely with Big Data.

So, in SQL, there are joins - five of them, I recall offhand; normal, left, right, full and cross.

Well, under the hood, Redshift uses three methods to implement all of these SQL joins; and these methods are a merge join, a hash join, and a nested loop join.

The merge join is the Holy Grail.

This is the and the only method which allows the timely joining of two Big Data tables.

It comes with a wide range of constraints and restrictions, all of which must be met for the merge join to be viable. AWS have never actually enumerated the conditions and restrictions, but you can either figure them out from first principles or you notice them over time; the join must be an equijoin (all join clauses must use the "equals" operator), the columns in the join must be a contiguous subset of the sorting order for both tables which begins with the first column in the sorting order - so the first column in the sort order must be joined, then you could also join the second, and the third, and so on, but you could not join on the third only, or the second and the third - and both tables must be up-to-date with `VACUUM`, and both tables must have the same distribution key.

So here again you can see - the tables must be designed correctly so they *can* merge join, which is hard enough, but then *also* the user must get the query right. The query must follow the sorting orders of the tables.

If you can manage all this, life is amazing. A merge join passes once down each table, uses no memory or processor time to speak of.

Here's some unpublished benchmarks of mine, for a merge join, duration is in seconds;

Slots	Duration
1	16.61
2	16.65
4	16.67
8	16.59
16	16.63
32	16.62
50	16.66

Here we have a single query running only, but we change the number of slots, and the more slots there are, the less and less memory a slot has.

Merge joins just don't care about memory.

If you cannot manage all this, what you get instead is the hash join, which is basically the fall-back join method, so at least you *do* get a join. A hash join will allow you to join one Big Data table, with one small data table, in a tolerably timely manner.

What a hash join does is basically create a transient (it's made for each query, then thrown away) index of the small table - in the same way that Postgres would create an index for a table, and if you've done that, you know how slow that is, because it's a lot of work - and then once the index exists, it iterates once over the big table, looking up each row in the index.

Now, there's a *critical* issue here. If the index fits into memory, everything is more or less fine; it's still a lot of work and resources, but it's tolerable.

The problem comes when the index does *not* fit into memory. Then it must go to disk, and when it runs from disk, you then start getting a *ton* of disk seeks.

This is death and ruin for you - your query will not return in a timely manner - but also for everyone else on the cluster, because the storage devices are now constantly performing seeks, and disk is the critical shared resource.

Now, the thing is, you can perfectly well do issue hash joins which hammer the disk - Redshift is *not* going to tell you this, or warn you, or stop you. The only way to avoid doing this is to know what you're doing, which brings us back to Redshift being knowledge intensive.

As before, one query only, but the number of slots varies, duration in seconds;

Slots	Duration
1	127.43
2	100.16
4	97.18
8	93.20
16	106.64
32	156.02
50	177.78

Eye-opener, isn't it.

Now, these results are *directly* comparable to the merge join results. It's the same two tables being joined, with the same data. All I did was change the join clause so I would get a hash join rather than a merge join; the number of rows, the data, all that, is all the same.

So, first, we see hash joins are *slow*.

Second, what's going on with the durations as the amount of memory in the slot varies?? if we have almost no memory, 50 slots, we're really slow - okay, that makes sense, we're spilling to disk. But when we have *maximum* memory, we're also slow! the sweet spot was actually about 8 slots.

To my eye, having looked in detail at what's going on, it looks like a case of improper optimization. Redshift seems to be thinking, "well, goodness, I have lots of memory - I can invest some time getting ready to do the main work of the query, so it runs faster". Problem is, turns out, the extra time getting ready is *more* than the time saved by getting ready, at least for the query I have.

To cap off this examination of joins, here are the figures for performance when we're running one query per slot, merge joins first table, hash joins second table.

Slots	Mean	StdDev	Throughput	Query Failures
1	16.71	0.000	0.060	0
2	17.21	0.007	0.116	0
4	29.59	0.029	0.135	0
8	57.21	0.007	0.140	0
16	111.36	0.023	0.144	0
30	207.33	0.034	0.145	0

Slots	Mean	StdDev	Throughput	Query Failures
1	100.58	0.000	0.010	0
2	178.27	0.184	0.011	0
4	350.75	0.112	0.011	0
8	705.63	1.700	0.011	0
16	1462.84	3.756	0.010	2
30	2357.27	93.470	0.006	16

Note the maximum number of slots here is 30, due to a bug in how Redshift handles large queues (which I hope has been fixed since I made these benchmarks).

Also note with hash joins, with 16 and 30 slots, we have queries failing - in fact, of the 30 queries issued with the 30 slot test, 16 failed.

The lesson here : don't do hash joins.

If you're not using sorting correctly, everything will be hash joins; go and use a clustered unsorted database which maintains indexes, rather than builds them

on a per-query basis.

Finally, moving on from merge joins and hash joins, we have the nested loop join, which is very, very rarely used. It happens really only when you issue a cross join, either directly, or implicitly by using really awkward join conditions. These are and necessarily killers; they're timely only with two small data tables, and you don't want to use them even then =-)

So, you can now see how much more complex the problem of ensuring tables and queries match up in fact is; it's not merely the case of making sure you arrange the columns in any given table in the way you need for your queries, and then ensure only appropriate queries are issued - you must also and at the same time arrange your tables so they can join correctly, and ensure your users only issue the correct queries for the joins, too.

System Design

So we can see now the challenges of system design in a sorted database.

If we load too much data, we will not be able to keep up with `VACUUM` and tables will become unsorted. Only one `VACUUM` can run at a time, and the larger the table being sorted, the longer the vacuum operation takes.

Also, if we load too much data, we will not be able to keep up with commit (scaling the cluster only makes this bottleneck worse), and that will lead to an ever growing backlog of queries waiting to commit; as we make the cluster larger, commits become slower.

We must avoid inserting or updating small volumes of data (less than a block per slice); the overheads in disk I/O are enormous, and will happily flatten performance for the cluster as a whole.

We need to ensure the queue and slot configuration is correct for the hash joins we do issue (there will always be some), so they all run in memory. If hash joins does not have enough memory, it swaps to disk, which hammers performance for everyone.

We need to design tables so that their sorting orders and distribution keys are such that we will get merge joins on all joins, by all queries, for all joins between two Big Data tables.

We also need to ensure the users issuing these queries understand why they need to adhere to merge joins, and also how to obtain merge joins.

The consequence for almost all of these requirements, if not met, is that we will get hash joins rather than merge joins, which makes queries slow, and this is a problem because only a few queries can run concurrently, and also because hash joins when the tables involved are larger than the memory available to the query, swap to disk, which hammers performance for everyone.

There's not much point in a user adhering to all these requirements, and running a proper query which correctly operates sorting, if other users on the cluster are *not* doing this, and by not doing this, are hammering cluster performance.

Broadly speaking, much of what has to be done is pretty simple and mechanical - don't try to load real-time data into a large table (because you'll run into a wall with `VACUUM`), monitor the system tables to ensure your queries are running their hash joins in memory, etc - but a key requirement, that the sorting orders and distribution keys of tables are such that the queries being executed issue merge joins, this is an art, not a science.

It's a balancing act, between any number of tables, any number of queries - queries you have now, queries you know you will have in the future, the general expected line of system development in the future - there are no hard and fast rules. The developer needs training, so they know about Redshift, and then it's a matter of experience, and aptitude - you need big picture thinkers for this kind of work, and they are systemically rare in computing; most software engineers are deep and narrow thinkers, and this kind of problem is the worst task you could present to them.

So this leads us to what exactly Redshift *is* for.

To my eye, Redshift is for loading Big Data, where you have plenty of time for data loading, into tables designed and used by skilled, experienced developers who understand databases and sorting, who will then issue hand-made queries on the data, performing analytics work.

It is not for anything else, because anything else means sorting is not operated correctly, and when that is the case, other databases become a better choice.

Summary

Redshift is a relational database which, when and only when operated correctly, is capable of timely SQL on Big Data.

I find practically all users come to Redshift having had experience of conventional relational databases are expecting the same, "but bigger".

This is completely, totally, utterly and absolutely incorrect.

It is critical to realise that although Redshift offers SQL, it is under the hood utterly unlike conventional relational databases, and as such its behaviour, characteristics, failure modes and use cases are *utterly* different.

Redshift is designed to provide timely SQL on Big Data, and provides two methods by which it scales performance, such that it can provide timely SQL on Big Data.

The first method is by the brute power of hardware.

Redshift runs on a cluster and as such you can simply add more machines to the cluster. However, this is expensive (to double performance, double your Redshift bill), has limits (you can only have so many machines in a cluster - about 100 machines tops), and not quite everything scales as you add more machines; in particular, the final stage of the work to add or update data, known as committing, becomes longer (and was already long, because there is a cluster rather than a single machine only), and committing is something only one query can do at a time.

The way I normally describe this is to say you are in one of two situations; either the hardware is overwhelming the data, and then you can do anything you like and all queries return in a few seconds, *or*, you are in a situation where the data overwhelms the hardware, and in this situation it is only the second method to provide performance which can give you timely SQL - sorting.

Sorting is the real magic.

In an conventional relational database, the data is unsorted, and as such the time taken to retrieve a row from a table becomes longer as the number of rows in the table increases (even though you're only retrieving one row). This is why unsorted relational databases cannot handle Big Data. Put a petabyte in there and you'll be coming back in a week.

With sorting - but when and only when sorting is correctly operated - this problem goes away. When queries run, the time taken to retrieve rows is no longer affected by the number of rows in the table. This is how sorted relational databases can handle Big Data, and they can do it with minimal hardware - you're not relying now on the brute power of hardware, you're relying on the mind-bendingly vast efficiency you get from sorting.

However, sorting does not inherently or automatically work. You have to operate it correctly, and operating sorting correctly is extremely knowledge intensive, places a wide range of onerous constraints and restrictions upon the system you're designing, the systems surrounding Redshift, the cluster admin, the table designers, and also upon the people *using* the tables, and, finally, sorting does

not enable *all* queries to run in a timely manner - rather, when you come to design tables, you also design your sorting, and this in turn defines the set of queries which will run in a timely manner and so also the set of queries which will *not* run in a timely manner.

Redshift consists of a number of tightly interlocking mechanisms, each of which directly or indirectly interacts with all the others. When all are being operated correctly, Redshift is a marvellous, smoothly operating device, capable of amazing feats - but as soon as you begin to operate even one of those mechanisms incorrectly, its irregular and improper operation is by those direct and indirect interactions communicated throughout the entire mechanism and everything then becomes problematically dysfunctional.

When sorting is not being operated correctly, Redshift is then back in exactly the same situation as an conventional unsorted relational database, and the time taken to retrieve a row once again depends on how many rows there are in a table.

The reason people use sorting at all is because there is no other way. Sorting is the and the only method to obtain timely SQL on Big Data - so if you must have that, then you must have sorting, which means you must honour its requirements, because there is no other way.

In my experience, practically all Redshift systems are operating sorting incorrectly, and provide performance purely through the brute power of hardware. What I've found is that people get away with this is because they do not in fact have Big Data; they have small data only, and so the power of the hardware overwhelms the data, and despite the fact Redshift is being operated incorrectly, it just doesn't matter. All queries execute in a few seconds.

I am of the view that Redshift, when sorting is operated incorrectly, is always the wrong choice of database.

The only method providing performance is the cluster, and there are other databases with clustering - in particular Postgres-XL and Exasol - but sorting, and the complex mechanisms around it, provide Redshift with all kinds of novel and unexpected ways to go wrong - the huge overheads of disk storage and I/O for small data, query compilation, queue and slot organization, the limited number of concurrent queries, the scarcity of `VACUUM` time, and so on and on.

I am of the view it is unwise to use Redshift unless you must have sorting, and you have sorting when and only when you need timely SQL on Big Data.

It's important here to understand that Redshift works very poorly on small data; being designed to handle Big Data means a database has taken a whole of set of design choices which make it no good at all for small data. As it is, because of the power of the hardware and where the data is small, the hardware overwhelms the data and no one notices - which is often a fatal misunderstanding, which only becomes apparent as data volumes grow.

It's important also to understand that a data system which uses sorting correctly has to be designed and implementation that way from the ground up. Sorting imposes far too much on the data system to be retro-fitted; if you make a data system which does not use sorting correctly, you will never be able to modify

that data system, in some low-cost way, to use sorting correctly. The data system will need to be rewritten from scratch - every table, every view, every query - and the users will need to be trained so they know about sorting and how to issue queries correctly.

Moreover, Redshift derives from Postgres 8, and as such is missing a metric ton of amazing functionality available in modern conventional databases. The `CREATE TABLE` syntax for Postgres 15 is three pages long - on Redshift, it's half a page.

Redshift is very difficult to operate correctly, has lots of fun, interesting and above all unexpected failure modes, and you're giving up a ton of functionality, and the reason you bear all these burdens is because of sorting, because sorting is the and the only way to get timely SQL on Big Data.

If you do not need timely SQL on Big Data, or you cannot or will not bear those burdens, you do not use Redshift. You use a conventional database, or a clustered conventional database.

The way it goes with SQL is this : you begin with a single-machine conventional relational database, like Postgres or MS-SQL.

When that's not enough, you move to clustered relational, either row-store (Postgres-XL) or column-store (Exasol). Note that AWS do not offer these types of product (which is one of the reasons, I think, people get moved onto Redshift - it is the only clustered relational AWS offer).

Finally, if a clustered conventional database is not enough, then it's time to bite the bullet, and move to a sorted relational database, such as Redshift.

Revision History

v1

- Initial release.