Redshift

Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the AWS Cloud. An Amazon Redshift data warehouse is a collection of computing resources called nodes, which are organized into a group called a cluster. Each cluster runs an Amazon Redshift engine and contains one or more databases.

Data Management

Understanding the Fundamentals of Redshift

Data Management

Understanding Relational Databases

Simply put, you can think of databases as software that simply stores and organizes data in an efficient way. When we think of databases, we tend to think of what are called Relational Databases because they store data in a *relational* way. This means the data is related to each other in some way. For example, a relational database may have several tables that help a product ordering application. These tables may be named something like *orders, customers, addresses, payments,* etc... You would then be able to connect these tables through what are called primary and foreign keys.

Primary Keys

Primary keys are unique for each record in a database. Lets use this website for example and assume that we have users with the following names:

NAME
John
Lauren
Carl

We might store this data in a table called *users*. Well, just having one table in our websites database is not very useful. Say we also want to see what these users are posting we could also have a *posts* table that may look like the following:

POST
Understanding Redshift
Seattle
AWS

Our database now has two *ENTIRE* tables! They are not very useful, though, because we cannot see who is posting what. This is where *primary keys* become useful. Now, we could setup just one large table that contains all of the websites data, but this quickly grows unwieldy whenever the data models become more complex than a simple blog. So instead we can assign a number that increases by 1 each time we add a new user or post. So we can refactor the tables to look like the following.

user_id (This will be the name of our primary key for users)	**name**
1	John
2	Lauren
3	Carl

post_id (This will be the name of the primary key for posts)	**post_title**
1	Understanding Redshift
2	Seattle
3	AWS

We now have our tables organized with *primary keys*, but we are unable to join the two tables to see who posted what article. For that, we will need to make use of a *foreign key*.

Foreign/Secondary Keys

A *foreign key* is a column or a set of columns in a table whose values correspond to the values of the primary key in another table. In order to add a row with a given foreign key value, there must exist a row in the related table with the same primary key value.

For our example a good foreign key for the *posts* table would be *user_id* so that way we can easily see who posted what article. So we can refactor our table to look something like this:

post_id (This will be the name of the primary key for posts)	**post_title**	**user_id**
1	Understanding Redshift	1
2	Seattle	1
3	AWS	2

We are now able to easily see who posted what article by matching the *user_id* to the *users* table. Our SQL query would look something like this:

```
SELECT
    p.post
    , u.name
FROM posts p

JOIN users u
    -- Join the foreign key of user_id from post to the primary key of user_id in users
    ON p.user_id = u.user_id
```

Our results would then look like the following:

post	name
Understanding Redshift	John
Seattle	John
AWS	Lauren

Data Management

OLAP vs OLTP Databases

Online Transactional Processing (OLTP)

There are two major types of Relational Database architectures: OLTP and OLAP. OLTP stands for "Online Transactional Processing". This original architecture stemmed from early applications (like banking software) needing to process *rows* of information at a given time. You can picture this operation as if you are ordering something from Amazon. When you place your order, it will send a row of information that contains your name, address, credit card details, and special instructions, because it is all relevant to this single transaction. Having the database architecture setup this way makes reads and writes from the database for single transactional details (think row of data) lightning fast. So given the following table:

name	address	credit card number	special instructions
Mike	123 Sesame Street	1234-5678-9101-1123	Hazmat
Kelsey	456 Abc avenue	3211-1019-8765-4321	Fragile

You could query it like so:

```
SELECT
*
FROM orders
```

When the query engine retrieves the information from the database it will first retrieve Mike's row of information, then it will retrieve Kelsey's row of information, and so on until it reaches the end. This is extremely efficient if you are just needing to retrieve a single row of information's detail. Where this now becomes an issue is if you are wanting to retrieve *all* of the names of all the customers, then count how many there are in the table.

OLTP databases need to retrieve the entire row of information even if you are only wanting to select a single column. You can start to see where this becomes an issue if you are only looking to grab a single column and then aggregate that column. This becomes an even larger issue when there are dozens of columns in the table. As you can imagine, this can bog down your database's compute availability. Enter onto the scene the Online Analytical Processing database

Online Analytical Processing (OLAP)

OLAP databases (e.g. Redshift, Terradata) came to popularity because Business Intelligence employees were running queries to determine aggregate metrics like "*How many orders were placed last week?*" or "*What is our best selling product?*", but the query engines for OLTP engines (like MySQL or Postgres) are not optimized for this type of workload. As you recall, OLTP scans *rows* on disks, the main difference in OLAP, is that it scans *columns* on disk. This allows us to query over only the columns we need to aggregate rather than the whole row. So in or table above, if we just select the count of the names like so:

```
SELECT
count(name) as name_count
FROM orders
```

We are only going to scan one block of data (the single column) vs 2 blocks of data (2 rows). As you can imagine this becomes extremely efficient the more rows we have.

> Keep in mind, I am over simplifying how this actually works for the sake of explanation and keeping the introduction to this concept *gentle.* While the data is stored in blocks like this, normally they have a size limit of 1mb on disk and then there are also indexes that are created which I will go into in a different post.

Organizing Redshift

Now that we understand the basics of creating primary keys, the concept of foreign keys, and the basics of OLAP. We will dive into how a distributed database like Redshift is best optimized.

Distributed Compute Engines

In a world driven by data, businesses generate lots and lots of data. We have already discussed the difference between OLAP and OLTP engines, but here I will just focus on how to scale (increase capacity by either creating a bigger computer or more computers) Redshift specifically, which is an OLAP database.

Redshift is an analytical database that functions as a single platform with several different computers (here out referred to as *nodes*) that can both store and process data. The main question is why would I want to create several new nodes instead of just increasing the size of the node my data currently resides on? Two main reasons:

1. Some data is so large that the tables could not fit onto even a single node and thus need to be stored across several nodes.
2. Reduce disk scan by organizing your data across several nodes.

So what does this mean? Let's take a look at how Redshift operates.

Simon Says

Redshift's group of nodes (which we will now refer to as a *cluster*), consist (normally) of a least one leader node and 1 or more worker nodes. The point of the worker nodes is to listen to the instructions of the leader node and return to it the data it is looking for per the query it was given. The leader node generates a plan for the worker nodes to follow that will best optimize the query it was given. Most of this abstraction takes place via Redshift built in operations, but there are some ways that users can help make queries more efficient.

Distribution (Dist) Keys

As you can recall from earlier, data in a Redshift cluster is stored across several nodes. Well, how does Redshift decide how that data gets stored? Normally, if the data is small enough, it will just store a copy of that data on every single node, which is quicker to retrieve than trying to search for

bits of a small dataset across several nodes. If the dataset is larger, Redshift will normally distribute the data in and *Even* fashion, giving equally sized chunks to each node in the cluster. Lets assume that we have the following table:

site	product	employee	value	date
seattle	car	greg	45,000	2022-05-21
portland	scooter	claire	850	2022-06-01
seattle	boat	carl	25,000	2022-06-08
portland	scooter	claire	850	2022-06-15
seattle	boat	carl	25,000	2022-06-04

Now also assume that we have a 3 node cluster with 1 leader node and 2 worker nodes. Normally a table this small would just be copied to every single node, but for example's sake, let's just assume that it will be distributed across the clusters like so:

node	site	product	employee	value	date
2	seattle	car	greg	45,000	2022-05-21
2	portland	scooter	claire	850	2022-06-01
3	seattle	boat	carl	25,000	2022-06-08
3	portland	scooter	claire	850	2022-06-15
3	seattle	boat	carl	25,000	2022-06-04

In this instance, no data is assigned to the leader node, but 2 of the rows get stored on node 2 and 3 rows get stored on node 3.

> OLAP databases actually store data in columnar blocks (generally of 1 Mb), but in order to make the point here and not get to down in the weeds we are just going to use rows for simplicity.

This is great, our data is distributed! Now, we are running into issues because we have very slow query performance. When we ask our analysts how they are querying the data, they send over the following query:

```
SELECT
*
FROM sales

WHERE site = 'seattle'
date > '2022-06-01'
```

We realize what is happening, the analysts are querying based on site, and because the sites are split across two nodes, Redshift has to scan two separate nodes for 3 rows. We can mitigate this inefficiency by telling Redshift that it should distribute the data based on the *site* column. We can do that by setting up the table by providing a *distkey.*

```
CREATE TABLE orders (
  site VARCHAR,
  product VARCHAR,
  employee VARCHAR,
  value NUMERIC,
  date TIMESTAMP
  )
  distkey(site);
```

Now when we take a look at the data it will be organized as follows:

node	site	product	employee	value	date
2	seattle	car	greg	45,000	2022-05-21
3	portland	scooter	claire	850	2022-06-01
2	seattle	boat	carl	25,000	2022-06-08
3	portland	scooter	claire	850	2022-06-15
2	seattle	boat	carl	25,000	2022-06-04

Notice that the sites are on the same node based on their location. Now when the query above is sent to the leader node, it will know to only check node 2 for the data.

Our analysts are super happy with how much the query speed has improved, but they still feel like it takes longer than necessary. There is another optimization that we can try to improve query speed and efficiency. We have told Redshift where to store our data (dist key), now let's tell it in what order to store our data with *sort keys.*

Sort Keys

Sort keys are a great way to prevent unnecessary scanning of data. We have used the dist key to reduce the amount of nodes scanned from 2 to 1. We can now use the *sort key* to reduce the amount of rows scanned. If we take a look back at our query, we can see that the analyst is only wanting to return site *seattle* and dates that are anything greater than June 1st, 2022. Let's assume that it is common to filter base on date. If we check out the order of the data for Seattle is stored on node 3, we can see the dates are out of order. *sort index* is not an actual column, but a reference to the order in which the leader node in Redshift will read the data.

sort index	node	site	product	employee	value	date
1	2	seattle	car	greg	45,000	2022-05-21
2	2	seattle	boat	carl	25,000	2022-06-08
3	2	seattle	boat	carl	25,000	2022-06-04

This is not optimal, because we are going to scan all 3 rows instead of just scanning the two rows for the data we need based on the sort index. We can use a sort key in order to prevent his from happening in the future.

```
ALTER TABLE sales ALTER SORTKEY date DESC;
```

We can now see the data sorted in a more efficient manner for the leader node to scan the data.

sort index	node	site	product	employee	value	date
3	2	seattle	car	greg	45,000	2022-05-21
1	2	seattle	boat	carl	25,000	2022-06-08
2	2	seattle	boat	carl	25,000	2022-06-04

When we run our query now we only scan 1 node and 2 rows vs before sorting and distributing properly we scanned 2 nodes and 5 rows. That's a 60% reduction in data being scanned! While this is a very simplistic example, those numbers are not far off from a real world example of properly distributing and sorting your data.