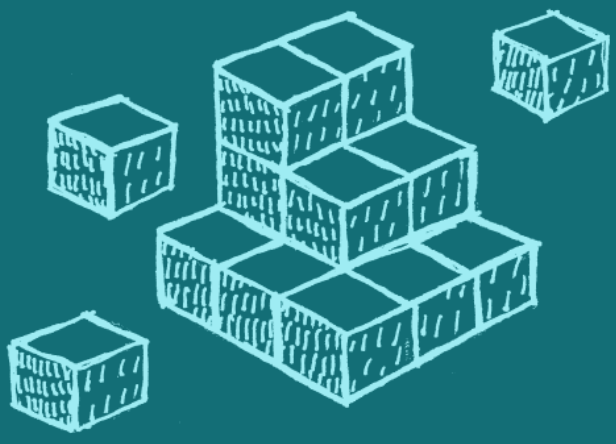
SQL Optimization



SQL Optimization

Written by: Matt David
Reviewed by: Blake Barnhill

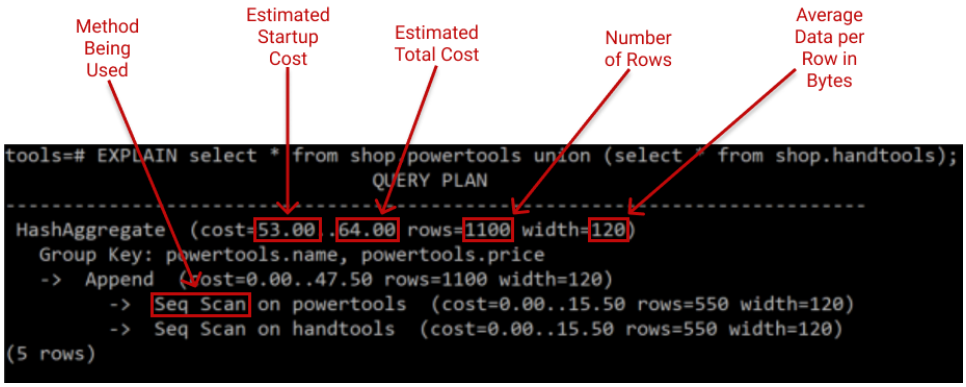
Table of Contents

Query Optimizations

What is a Query Plan

A Query plan is a list of instructions that the database needs to follow in order to execute a query on the data.

Below is an example query plan for a given query:



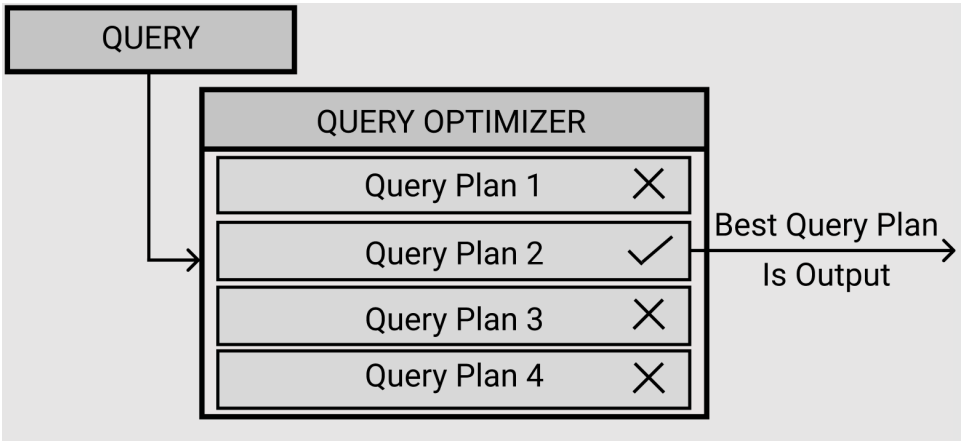
This query plan shows the particular steps taken to execute the given command. It also specifies the expected cost for each section.

Video

The Query Optimizer

SQL is a declarative language. This means that SQL queries describe what the user wants and then the query is transformed into executable commands by the Query Optimizer. Those executable commands are known as Query Plans.

The Query Optimizer generates multiple Query Plans for a single query and determines the most efficient plan to run.



There are often many different ways to search a database. Take for example the following database of tools that has five entries. Each entry has a unique ID number and a non-unique name.

ID	Name
1	Callipers
2	Hammer
3	Screwdriver
4	Wrench
5	Hammer

In order to find a particular tool, there are several possible queries that could be run. For example, the query:

```
SELECT *
FROM tools
WHERE name='Screwdriver';
```

Will return the same thing as the query:

```
SELECT *
FROM tools
WHERE id=3;
```

Scan Vs. Seek

These queries will return the same results, but may have different final query plans. The first query will have a query plan that uses a sequential scan. This means that all five rows of the database will be checked to see if the name is screwdriver and, when run, would look like the following table:

ID	Name
1	Callipers
2	Hammer
3	Screwdriver
4	Wrench
5	Hammer

(green = match) (red = miss) (white = not checked)

The second query will use a query plan which implements a sequential seek since the second query handles unique values. Like a scan, a seek will go through each entry and check to see if the condition is met. However unlike a scan, a seek will stop once a matching entry has been found. A seek for ID = 3 would look like the following figure:

ID	Name
1	Callipers
2	Hammer
3	Screwdriver
4	Wrench
5	Hammer

(green = match) (red = miss) (white = not checked)

This seek only needs to check three rows in order to return the result unlike a scan which must check the entire database.

For more complicated queries there may be situations in which one query plan implements a seek while the other implements a scan. In this case, the query optimizer will choose the query plan that implements a seek, since seeks are more efficient than scans. There are also different types of scans that have different efficiencies in different situations.

Summary

- Query plans are a set of instructions generated by the Query Optimizer.
- They generate estimates of query efficiency
- The Query Optimizer generates multiple query plans and determines which plan is most efficient for a given query.
- Scan Vs. Seek
- Scans search the whole database for matches
- Seeks search the database for a single match and stop once they have found it

Order of a SQL Query

The way to make a query run faster is to reduce the number of calculations that the software must perform. To do this, you'll need some understanding of how SQL executes a query.

Let's take a look at a sample SQL query :

```
SELECT DISTINCT column, AGGREGATE(column)
FROM table1
JOIN table2
ON table1.column = table2.column
WHERE constraint_expression
GROUP BY column
HAVING constraint_expression
ORDER BY column ASC/DESC
LIMIT count;
```

Each part of the query is executed sequentially, so it's important to understand the order of execution :

1. **FROM and JOIN:** The FROM clause, and subsequent JOINs are first executed to determine the total working set of data that is being queried
2. **WHERE:** Once we have the total working set of data, the WHERE constraints are applied to the individual rows, and rows that do not satisfy the constraint are discarded.
3. **GROUP BY:** The remaining rows after the WHERE constraints are applied are then grouped based on common values in the column specified in the GROUP BY clause.
4. **HAVING:** If the query has a GROUP BY clause, then the constraints in the HAVING clause are applied to the grouped rows, and the grouped rows that don't satisfy the constraint are discarded.
5. **SELECT:** Any expressions in the SELECT part of the query are finally computed.
6. **DISTINCT:** Of the remaining rows, rows with duplicate values in the column marked as DISTINCT will be discarded.
7. **ORDER BY:** If an order is specified by the ORDER BY clause, the rows are then sorted by the specified data in either ascending or descending order.
8. **LIMIT:** Finally, the rows that fall outside the range specified by the LIMIT are discarded, leaving the final set of rows to be returned from the query.

Now that we understand the basic structure and order of a SQL query, we can take a look at the tips to optimize them for faster processing in the next chapter.

Resources

1. https://sqlbolt.com/lesson/select_queries_order_of_execution
2. https://www.sisense.com/blog/8-ways-fine-tune-sql-queries-production-databases/

Optimize your SQL Query

8 tips for faster querying

1. **Define SELECT fields instead of SELECT * :** If a table has many fields and rows, selecting all the columns (by using SELECT *) over-utilizes the database resources in querying a lot of unnecessary data. Defining fields in the SELECT statement will point the database to querying only the required data to solve the business problem.
2. **Avoid SELECT DISTINCT if possible:** SELECT DISTINCT works by grouping all fields in the query to create distinct results. To accomplish this goal however, a large amount of processing power is required.
3. **Use WHERE instead of HAVING to define Filters:** As per the SQL order of operations, HAVING statements are calculated after WHERE statements. If we need to filter a query based on conditions, a WHERE statement is more efficient.

4. **Use WILDCARDS at the end of the phrase:** When a leading wildcard is used, especially in combination with an ending wildcard, the database is tasked with searching all records for a match anywhere within the selected field.
   Consider this query to pull cities beginning with 'Char':

   ```
   SELECT City FROM Customers
   WHERE City LIKE '%Char%'
   A more efficient query would be:
   SELECT City FROM Customers
   WHERE City LIKE 'Char%'
   ```

5. **Use LIMIT to sample query results:** Before running a query for the first time, ensure the results will be desirable and meaningful by using a LIMIT statement.
6. **Run Queries During Off-Peak Times:** Heavier queries which take a lot of database load should run when concurrent users are at their lowest number, which is typically during the middle of the night.

7. **Replace SUBQUERIES with JOIN:** Although subqueries are useful, they often can be replaced by a join, which is definitely faster to execute. Consider the example below :

   ```
   SELECT a.id,
   (SELECT MAX(created)
   FROM posts
   WHERE author_id = a.id)
   AS latest_post FROM authors a
   To avoid the sub-query, it can be rewritten with a join as :
   SELECT a.id, MAX(p.created) AS latest_post
   FROM authors a
   INNER JOIN posts p
   ON (a.id = p.author_id)
   GROUP BY a.id
   ```

8. **Index your tables properly:** Proper indexing can make a slow database perform better. Conversely, improper indexing can make a high-performing database run poorly. The difference depends on how you structure the indexes. You should create an index on a column in any of the following situations:
   - The column is queried frequently
   - Foreign key column(s) that reference other tables
   - A unique key exists on the column(s)

Conclusion

When querying a production database, optimization is key. An inefficient query may pose a burden on the production database's resources, and cause slow performance or loss of service for other users if the query contains errors.

When optimizing your database server, you need to tune the performance of individual queries. This is even more important than tuning other aspects of your server installation that affect performance, such as hardware and software configurations. Even if your database server runs on the most powerful hardware available, its performance can be negatively affected by a handful of misbehaving queries.

Resources

1. https://sqlbolt.com/lesson/select_queries_order_of_execution
2. https://www.sisense.com/blog/8-ways-fine-tune-sql-queries-production-databases/

Column and Table Optimizations

Optimization with EXPLAIN ANALYZE

Querying postgres databases, when done properly, can result in extremely efficient results and provide powerful insights. Sometimes however, queries are written in less than optimal ways, causing slow response times. Because of this, it is important to be able to analyze how queries execute and find the most optimized ways to run them.

One method for optimizing queries is to examine the query plan to see how a query is executing and adjust the query to be more efficient. Using the query plan can provide many insights into why a query is running inefficiently.

Explain and Explain Analyze

In postgreSQL, the query plan can be examined using the ***EXPLAIN*** command:

```
EXPLAIN SELECT seqid FROM traffic WHERE serial_id<21;
```

```
bigdb=# EXPLAIN SELECT seqid FROM traffic WHERE serial_id < 21;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx_serial_id on traffic  (cost=0.43..18.70 rows=21 width=41)
   Index Cond: (serial_id < 21)
(2 rows)
```

This command shows the generated query plan but does not run the query. In order to see the results of actually executing the query, you can use the ***EXPLAIN ANALYZE*** command:

```
EXPLAIN ANALYZE SELECT seqid FROM traffic WHERE serial_id<21;
```

```
bigdb=# EXPLAIN ANALYZE SELECT seqid FROM traffic WHERE serial_id < 21;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Index Scan using idx_serial_id on traffic  (cost=0.43..18.70 rows=21 width=41) (actual time=0.067..0.105 rows=20 loops=1)
   Index Cond: (serial_id < 21)
 Planning Time: 0.296 ms
 Execution Time: 2.133 ms
(4 rows)
```

Warning: Adding ***ANALYZE*** to ***EXPLAIN*** will both run the query and provide statistics. This means that if you use ***EXPLAIN ANALYZE*** on a ***DROP*** command (Such as EXPLAIN ANALYZE DROP TABLE table), the specified values will be dropped after the query executes.

The Data Used

The data that is being used to demonstrate optimization is a table of data regarding traffic violations which can be found here. It was imported from the csv file available for download and altered to have a SERIAL row named serial_id. The details of the table are shown here:

\d+ traffic

```
                                    Table "public.traffic"
       Column          |          Type          | Collation | Nullable |                  Default
-----------------------+------------------------+-----------+----------+-------------------------------------------
 seqid                 | character(36)          |           |          |
 dateofstop            | date                   |           |          |
 timeofstop            | time without time zone |           |          |
 agency                | character varying      |           |          |
 subagency             | character varying      |           |          |
 description           | character varying      |           |          |
 location              | character varying      |           |          |
 latitude              | character varying      |           |          |
 longitude             | character varying      |           |          |
 accident              | character varying      |           |          |
 belts                 | boolean                |           |          |
 personalinjury        | boolean                |           |          |
 propertydamage        | boolean                |           |          |
 fatal                 | boolean                |           |          |
 commerciallicense     | boolean                |           |          |
 hazmat                | boolean                |           |          |
 commercialvehicle     | boolean                |           |          |
 alcohol               | boolean                |           |          |
 workzone              | boolean                |           |          |
 state                 | character varying      |           |          |
 vehicletype           | character varying      |           |          |
 year                  | smallint               |           |          |
 make                  | character varying      |           |          |
 model                 | character varying      |           |          |
 color                 | character varying      |           |          |
 violationtype         | character varying      |           |          |
 charge                | character varying      |           |          |
 article               | character varying      |           |          |
 contributedtoaccident | character varying      |           |          |
 race                  | character varying      |           |          |
 gender                | character varying      |           |          |
 drivercity            | character varying      |           |          |
 driverstate           | character varying      |           |          |
 dlstate               | character varying      |           |          |
 arresttype            | character varying      |           |          |
 geolocation           | character varying      |           |          |
 councildistricts      | character varying      |           |          |
 councils              | character varying      |           |          |
 communities           | character varying      |           |          |
 zipcodes              | character varying      |           |          |
 municipalities        | character varying      |           |          |
 serial_id             | integer                |           | not null | nextval('traffic_serial_id_seq'::regclass)
```

Indexes

Indexes are vital to efficiency in SQL. They can dramatically improve query speed because it changes the query plan to a much faster method of search. It is important to use them for heavily queried columns.

For example, an index on the serial_id column in the sample data can make a large difference in execution time. Before adding an index, executing the following query would take up to 13 seconds as shown below:

```
EXPLAIN ANALYZE SELECT * FROM traffic WHERE serial_id = 1;
```

```
bigdb=# explain analyze select * from traffic where serial_id = 1;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..389406.77 rows=1 width=1908) (actual time=13022.771..13024.689 rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on traffic  (cost=0.00..388406.67 rows=1 width=1908) (actual time=11691.284..13012.262 rows=0 loops=3)
         Filter: (serial_id = 1)
         Rows Removed by Filter: 507306
 Planning Time: 0.291 ms
 Execution Time: 13024.774 ms = 13 seconds
(8 rows)
```

13 seconds to return a single row on a database of this size is definitely a suboptimal result. In the query plan we can see that the query is running a parallel sequential scan on the *entire* table which is inefficient. This operation has a high start up time (1000ms) and execution time (13024ms).

A parallel sequential scan can be avoided by creating an index and analyzing it as shown:

```
CREATE INDEX idx_serial_id ON traffic(serial_id);
ANALYZE;
```

```
bigdb=# CREATE INDEX idx_serial_id ON traffic(serial_id);
CREATE INDEX
bigdb=# ANALYZE;
ANALYZE
bigdb=# EXPLAIN ANALYZE SELECT * FROM traffic WHERE serial_id = 1;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Index Scan using idx_serial_id on traffic  (cost=0.43..8.45 rows=1 width=381) (actual time=0.525..0.528 rows=1 loops=1)
   Index Cond: (serial_id = 1)
 Planning Time: 4.011 ms
 Execution Time: 0.587 ms
(4 rows)
```

This shows that when using an index, the execution time drops from 13024.774 ms to 0.587 ms (that is a 99.99549% decrease in time). This is a dramatic decrease in execution time. The planning time does rise by 3.72 ms because the query planner needs to access

the index and decide if using it would be efficient before it can start the execution. However the rise in planning time is negligible compared to the change in execution time.

Not all indexes will have the same amount of impact on queries. It is important to use explain analyze before and after implementing an Index to see what the impact was. Read Blake Barnhill's article on indexing for more information.

Indexes are not always the answer. There will be times when a sequential scan is better than an index scan. This is the case for small tables, large data types, or tables that already have enough indexes to the specified query.

Partial Indexes

Sometimes it is best to use a partial index as opposed to a full index. A partial index is an index that stores ordered data on the results of a query rather than a column. Partial indexes are best for when you want a specific filter to operate quickly. For example, in this table, there are many types of vehicles that are recorded:

```
SELECT DISTINCT vehicletype FROM traffic
GROUP BY vehicletype;
```

```
bigdb=# SELECT DISTINCT vehicletype from traffic GROUP BY vehicletype;
        vehicletype
 ---------------------------
  01 - Motorcycle
  02 - Automobile
  03 - Station Wagon
  04 - Limousine
  05 - Light Duty Truck
  06 - Heavy Duty Truck
  07 - Truck/Road Tractor
  08 - Recreational Vehicle
  09 - Farm Vehicle
  10 - Transit Bus
  11 - Cross Country Bus
  12 - School Bus
  13 - Ambulance
  13 - Ambulance(Emerg)
  14 - Ambulance
  14 - Ambulance(Non-Emerg)
  15 - Fire Vehicle
  15 - Fire(Emerg)
  16 - Fire(Non-Emerg)
  17 - Police(Emerg)
  18 - Police Vehicle
  18 - Police(Non-Emerg)
  19 - Moped
  20 - Commercial Rig
  21 - Tandem Trailer
  22 - Mobile Home
  23 - Travel/Home Trailer
  24 - Camper
  25 - Utility Trailer
  26 - Boat Trailer
  27 - Farm Equipment
  28 - Other
  29 - Unknown
 (33 rows)
```

Many studies are done on motorcycle safety. For these studies, it would be wise to use a partial index on only motorcycles as opposed to an index which also includes unneeded information about other vehicle types. To create a partial index that only indexes rows involving motorcycles, the following query can be run:

```
CREATE INDEX idx_motorcycle ON traffic(vehicletype)
WHERE vehicle = '01 - Motorcycle';
ANALYZE;
```

```
bigdb=# CREATE INDEX idx_motorcycle ON traffic(vehicletype) WHERE vehicletype = '01 – Motorcycle';
CREATE INDEX
bigdb=# ANALYZE;
ANALYZE
```

This index will only store data relating to motorcycle violations, which will be much faster to search through than searching through the entire table.

Here are a few final tips about indexing:

1. Remember to ***ANALYZE***: it is important to run ***ANALYZE;*** after creating an index in order to update the statistics on the index. This lets the query planner make the most informed decision on when to use an index)
2. Remember to ***VACUUM***: It is important to run ***VACUUM ANALYZE*** after significant modifications to the table. This command will clean up the index's statistics, throwing away old values and adding new values. This command will run automatically periodically, however it is useful to run this command after heavy modifications for quicker results.
3. Make sure your query can use the index: avoid using Regex patterns with a wild card at the beginning such as ***LIKE '%[pattern]%'*** this makes the query planner unable to use an index. You can use Regex patterns with wildcards at the end such as ***LIKE '[pattern]%'***.

Data Types

Another important aspect of efficiency is the data types being used. Data types can have a large impact on performance.

Different data types can have drastically different storage sizes as shown by this table from the postgreSQL documentation on **numeric types**:

Table 8.2. Numeric Types

Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

The dataset of traffic violations contains 1,521,919 rows in it (found using the **COUNT** aggregation). We need to consider the data type that requires the least amount of space that can store the data we want. We added a serial column to the data, which starts at 0 and increments by one each row. Since the data is 1,521,919 rows long we need a data type that can store at least that amount of data:

- **smallserial** is the best choice If the numbers of rows is under 32768 rows.
- **serial** is the best choice if the number of rows is more than small serial's max or less than bigserials minimum.
- **bigserial** is the appropriate choice if there are more than 2,147,483,647 rows.

1,521,919 is greater than the **smallserial** limit (32,768) and less than the limit for **serial** (2,147,483,647), so **serial** should be used.

If we chose **bigserial** for the column, it would use twice the amount of memory needed to store each value because every value that is bigserial is stored as 8 bytes instead of 4 bytes for serial. While this is not terribly significant for very small tables on tables at the size of *traffic* or larger, this can make a large difference. In the traffic data set this would be an extra 6,087,676 bytes (6MB). While this is not too significant, it does impact efficiency of scans and inserts. The same principle applies to larger data types as well such as char(n), text data types, date/time types etc.

An example of this is, if a copy of traffic is created where **serial** is replaced with **bigserial**, then scan times rise. We can see this by comparing EXPLAIN ANALYZE results:

Results from Original Table:

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM traffic;
```

```
Finalize Aggregate  (cost=33240.90..33240.91 rows=1 width=8) (actual time=194.472..194.473 rows=1 loops=1)
  ->  Gather  (cost=33240.68..33240.89 rows=2 width=8) (actual time=194.394..196.828 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate  (cost=32240.68..32240.69 rows=1 width=8) (actual time=181.269..181.270 rows=1 loops=3)
              ->  Parallel Index Only Scan using test on traffic  (cost=0.43..30655.35 rows=634133 width=0) (actual time
                    Heap Fetches: 0
Planning Time: 0.552 ms
Execution Time: 197.126 ms
```

Results from Copy:

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM traffic2;
```

```
Finalize Aggregate  (cost=87578.69..87578.70 rows=1 width=8) (actual time=1137.888..1137.888 rows=1 loops=1)
  ->  Gather  (cost=87578.47..87578.68 rows=2 width=8) (actual time=1137.780..1139.469 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate  (cost=86578.47..86578.48 rows=1 width=8) (actual time=1131.300..1131.300 rows=1 loops=3)
              ->  Parallel Seq Scan on traffic2  (cost=0.00..85969.38 rows=243638 width=0) (actual time=0.629..1059.796 rows=507306 loops=3)
Planning Time: 0.569 ms
Execution Time: 1139.514 ms
(8 rows)
```

As we can see from the images above, the time to aggregate on the original table is 197 ms or 0.2 seconds. The time to aggregate on the inefficient copy is 1139 ms or 1.1 seconds (5.5 times slower). This example clearly shows that data types can make a large impact on efficiency.

Summary

- EXPLAIN ANALYZE is a way to see exactly how your query is performing.
- Indexes
  - Best when created on unique ordered values
  - Remember to ANALYZE after creating
  - Remember to VACUUM ANALYZE
  - Remember to write queries so that indexes can be used (e.g. use: LIKE ‘string%’ Don’t use: LIKE ‘%string%’)
- Partial Indexes
  - Can be used for frequently queried subsections of a table
- Data Types
  - Ensure that the smallest data type is used
    - Be careful of this on tables with high throughput. Tables may grow past the data type’s size which will cause an error.

References:

1. https://thoughtbot.com/blog/postgresql-performance-considerations
2. https://statsbot.co/blog/postgresql-query-optimization/
3. https://www.sentryone.com/white-papers/data-type-choice-affects-database-performance

Indexing

What is Indexing?

Indexing makes columns faster to query by creating pointers to where data is stored within a database.

Imagine you want to find a piece of information that is within a large database. To get this information out of the database the computer will look through every row until it finds it. If the data you are looking for is towards the very end, this query would take a long time to run.

Visualization for finding the last entry:

SELECT * FROM friends WHERE name = 'Zack';

friends		
id	**name**	**city**
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

If the table was ordered alphabetically, searching for a name could happen a lot faster because we could skip looking for the data in certain rows. If we wanted to search for "Zack" and we know the data is in alphabetical order we could jump down to halfway through the data to see if Zack comes before or after that row. We could then half the remaining rows and make the same comparison.

SELECT * FROM friends WHERE name = 'Zack';

friends_name_asc	
Name	**Index**
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

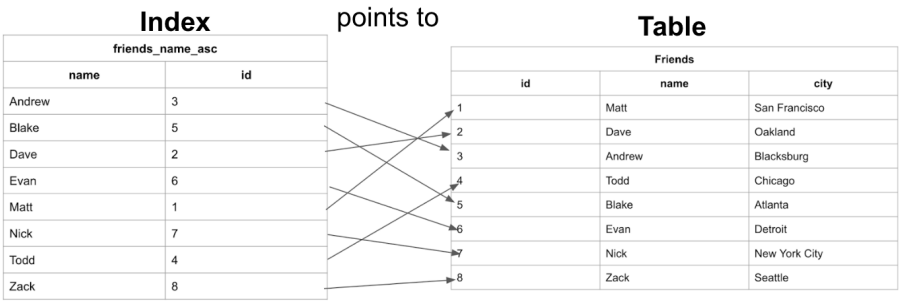
This took 3 comparisons to find the right answer instead of 8 in the unindexed data.

Indexes allow us to create sorted lists without having to create all new sorted tables, which would take up a lot of storage space.

What Exactly is an Index?

An index is a structure that holds the field the index is sorting and a pointer from each record to their corresponding record in the original table where the data is actually stored. Indexes are used in things like a contact list where the data may be physically stored in the order you add people's contact information but it is easier to find people when listed out in alphabetical order.

Let's look at the index from the previous example and see how it maps back to the original Friends table:



friends_name_asc

name	id
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

Friends

id	name	city
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

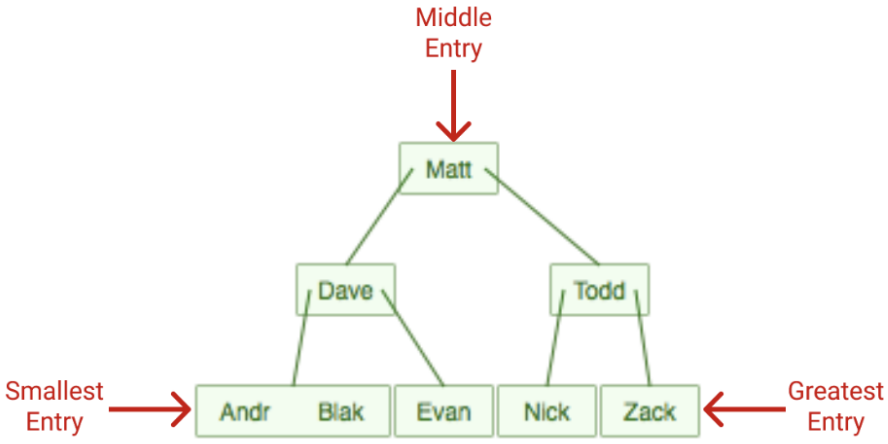
We can see here that the table has the data stored ordered by an incrementing id based on the order in which the data was added. And the Index has the names stored in alphabetical order.

Types of Indexing

There are two types of databases indexes:

1. **Clustered**
2. **Non-clustered**

Both clustered and non-clustered indexes are stored and searched as B-trees, a data structure similar to a binary tree. A B-tree is a "self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time." Basically it creates a tree-like structure that sorts data for quick searching.



Here is a B-tree of the index we created. Our smallest entry is the leftmost entry and our largest is the rightmost entry. All queries would start at the top node and work their way down the tree, if the target entry is less than the current node the left path is followed, if greater the right path is followed. In our case it checked against Matt, then Todd, and then Zack.

To increase efficiency, many B-trees will limit the number of characters you can enter into an entry. The B-tree will do this on it's own and does not require column data to be restricted. In the example above the B-tree below limits entries to 4 characters.

Clustered Indexes

Clustered indexes are the unique index per table that uses the primary key to organize the data that is within the table. The clustered index ensures that the primary key is stored in increasing order, which is also the order the table holds in memory.

- Clustered indexes do not have to be explicitly declared.
- Created when the table is created.
- Use the primary key sorted in ascending order.

Creating Clustered Indexes

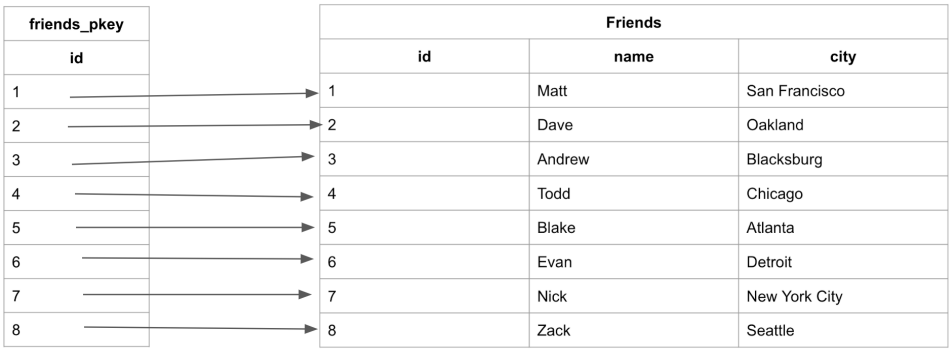
The clustered index will be automatically created when the primary key is defined:

```
CREATE TABLE friends (id INT PRIMARY KEY, name VARCHAR, city VARCHAR);
```

Once filled in, that table would look something like this:

Friends		
id	**name**	**city**
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

The created table, “friends”, will have a clustered index automatically created, organized around the Primary Key “id” called “friends_pkey”:

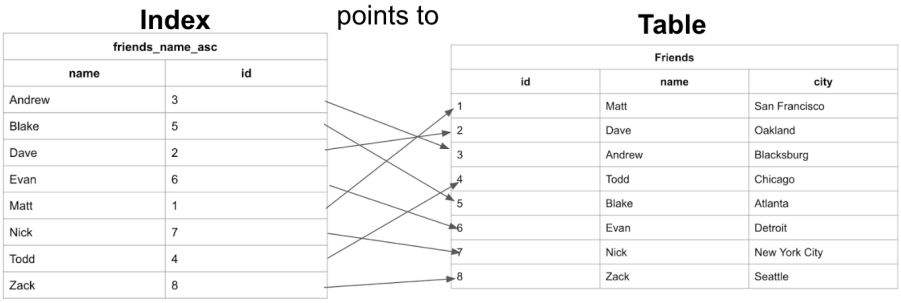


When searching the table by “id”, the ascending order of the column allows for optimal searches to be performed. Since the numbers are ordered, the search can navigate the B-tree allowing searches to happen in logarithmic time.

However, in order to search for the “name” or “city” in the table, we would have to look at every entry because these columns do not have an index. This is where non-clustered indexes become very useful.

Non-Clustered Indexes

Non-clustered indexes are sorted references for a specific field, from the main table, that hold pointers back to the original entries of the table. The first example we showed is an example of a non-clustered table:



They are used to increase the speed of queries on the table by creating columns that are more easily searchable. Non-clustered indexes can be created by data analysts/ developers after a table has been created and filled.

Note: Non-clustered indexes are **not** new tables. Non-clustered indexes hold the field that they are responsible for sorting and a pointer from each of those entries back to the full entry in the table.

You can think of these just like indexes in a book. The index points to the location in the book where you can find the data you are looking for.

Index

Non-clustered indexes point to memory addresses instead of storing data themselves. This makes them slower to query than clustered indexes but typically much faster than a non-indexed column.

You can create many non-clustered indexes. As of 2008, you can have up to 999 non-clustered indexes in SQL Server and there is no limit in PostgreSQL.

Creating Non-Clustered Databases(PostgreSQL)

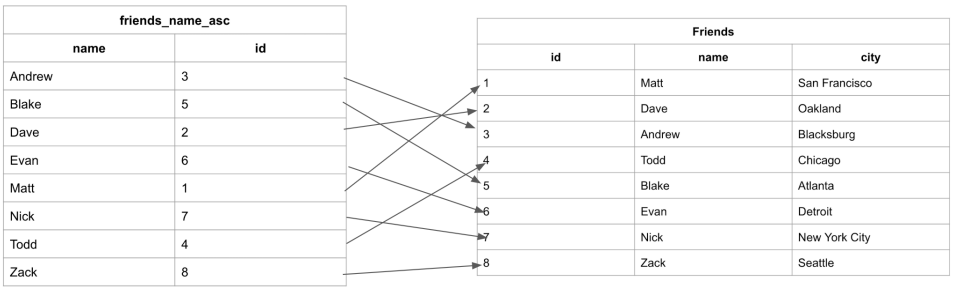
To create an index to sort our friends' names alphabetically:

```
CREATE INDEX friends_name_asc ON friends(name ASC);
```

This would create an index called "friends_name_asc", indicating that this index is storing the **names** from "friends" stored alphabetically in **ascending** order.

friends_name_asc	
name	**id**
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

Note that the "city" column is not present in this index. That is because indexes do not store all of the information from the original table. The "id" column would be a pointer back to the original table. The pointer logic would look like this:

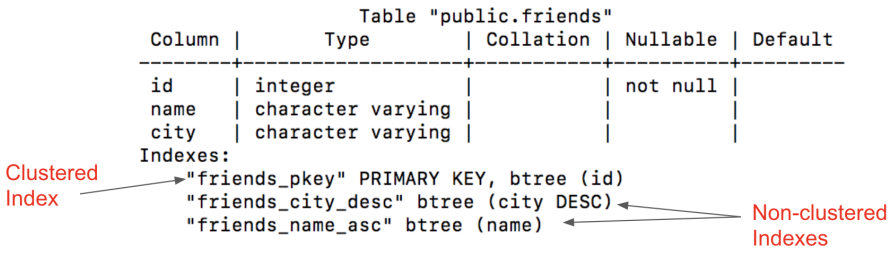


Creating Indexes

In PostgreSQL, the "\d" command is used to list details on a table, including table name, the table columns and their data types, indexes, and constraints.

The details of our friends table now look like this:

Query providing details on the friends table: \d friends;



Looking at the above image, the “friends_name_asc” is now an associated index of the “friends” table. That means the query plan, the plan that SQL creates when determining the best way to perform a query, will begin to use the index when queries are being made. Notice that “friends_pkey” is listed as an index even though we never declared that as an index. That is the **clustered index** that was referenced earlier in the article that is automatically created based off of the primary key.

We can also see there is a “friends_city_desc” index. That index was created similarly to the names index:

```
CREATE INDEX friends_city_desc ON friends(city DESC);
```

This new index will be used to sort the cities and will be stored in reverse alphabetical order because the keyword “DESC” was passed, short for “descending”. This provides a way for our database to swiftly query city names.

Searching Indexes

After your non-clustered indexes are created you can begin querying with them. Indexes use an optimal search method known as binary search. Binary searches work by constantly cutting the data in half and checking if the entry you are searching for comes before or after the entry in the middle of the current portion of data. This works well with B-trees because they are designed to start at the middle entry; to search for the entries within the tree you know the entries down the left path will be smaller or before the current entry and the entries to the right will be larger or after the current entry. In a table this would look like:

SELECT * FROM friends WHERE name = ‘Zack’;

friends_name_asc	
Name	**Index**
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

Comparing this method to the query of the non-indexed table at the beginning of the article, we are able to reduce the total number of searches from eight to three. Using this method, a search of 1,000,000 entries can be reduced down to just 20 jumps in a binary search.

Binary Search Complexity	
Number of Entries	**Greatest number of searches to find target**
8	3
100	7
1,000	10
10,000	14
100,000	17
1,000,000	20

When to use Indexes

Indexes are meant to speed up the performance of a database, so use indexing whenever it significantly improves the performance of your database. As your database becomes larger and larger, the more likely you are to see benefits from indexing.

When not to use Indexes

When data is written to the database, the original table (the clustered index) is updated first and then all of the indexes off of that table are updated. Every time a write is made to the database, the indexes are unusable until they have updated. If the database is constantly receiving writes then the indexes will never be usable. This is why indexes are typically applied to databases in data warehouses that get new data updated on a scheduled basis(off-peak hours) and not production databases which might be receiving new writes all the time.

NOTE: The newest version of Postgres (that is currently in beta) will allow you to query the database while the indexes are being updated.

Testing Index performance

To test if indexes will begin to decrease query times, you can run a set of queries on your database, record the time it takes those queries to finish, and then begin creating indexes and rerunning your tests.

To do this, try using the EXPLAIN ANALYZE clause in PostgreSQL.:

```
EXPLAIN ANALYZE SELECT * FROM friends WHERE name = 'Blake';
```

Which on my small database yielded:

```
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on friends  (cost=0.00..1.06 rows=1 width=68) (actual time=1.320..1.320 rows=0 loops=1)
   Filter: ((name)::text = 'Blake'::text)
   Rows Removed by Filter: 5
 Planning Time: 4.225 ms
 Execution Time: 1.834 ms
(5 rows)

(END)
```

This output will tell you which method of search from the query plan was chosen and how long the planning and execution of the query took.

Only create one index at a time because not all indexes will decrease query time.

- PostgreSQL's query planning is pretty efficient, so adding a new index may not affect how fast queries are performed.
- Adding an index will always mean storing more data
- Adding an index will increase how long it takes your database to fully update after a write operation.

If adding an index does not decrease query time, you can simply remove it from the database.

To remove an index use the DROP INDEX command:

```
DROP INDEX friends_name_asc;
```

The outline of the database now looks like:

```
                  Table "public.friends"
 Column |       Type        | Collation | Nullable | Default
--------+-------------------+-----------+----------+---------
 id     | integer           |           | not null |
 name   | character varying |           |          |
 city   | character varying |           |          |
Indexes:
    "friends_pkey" PRIMARY KEY, btree (id)
    "friends_city_desc" btree (city DESC)
```

Which shows the successful removal of the index for searching names.

Summary

- Indexing can vastly reduce the time of queries
- Every table with a primary key has one clustered index
- Every table can have many non-clustered indexes to aid in querying
- Non-clustered indexes hold pointers back to the main table
- Not every database will benefit from indexing
- Not every index will increase the query speed for the database

References:

https://www.geeksforgeeks.org/indexing-in-databases-set-1/ https://www.c-sharpcorner.com/blogs/differences-between-clustered-index-and-nonclustered-index1
https://en.wikipedia.org/wiki/B-tree
https://www.tutorialspoint.com/postgresql/postgresql_indexes.htm
https://www.cybertec-postgresql.com/en/postgresql-indexing-index-scan-vs-bitmap-scan-vs-sequential-scan-basics/#

Partial Indexes

Partial indexes store information on the results of a query, rather than on a whole column which is what a traditional index does. This can speed up queries significantly compared to a traditional Index if the query targets the set of rows the partial index was created for.

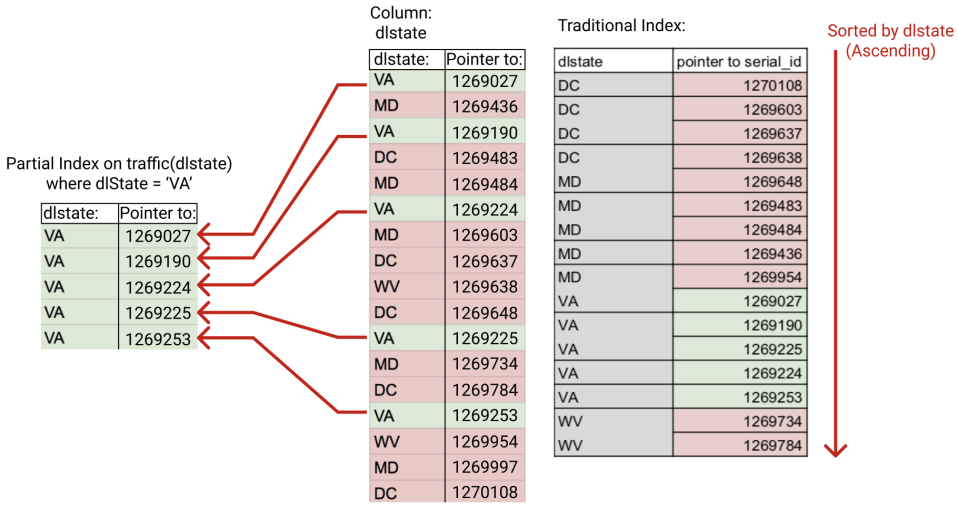
Creating indexes is a common practice to optimize data warehouses, as explored in this article on traditional indexing. Partial Indexes are less discussed but should be incorporated along with traditional indexes into any optimization strategy due to their dramatic performance effects.

Basic Partial Index

A partial index can be as simple as an index that stores data on a subsection of a column. Take for example data.gov's dataset on traffic violations in Montgomery County, MD. This table has many columns that could benefit from partial indexing. Columns that have many distinct groups in them are good for partial indexes as there is a large amount of data in the column. This is because columns like this are often filtered for one particular group.

One example of this is the column **dlstate** (the state that the driver has a driver's license from). There are 71 distinct values (Includes out of country plates like QC for Quebec and includes XX for no plate) in the database for this column. Say you are studying how many Virginia driver's are involved in Montgomery County traffic violations.

We are going to be filtering all our queries to where **dlstate** = "VA" so applying a partial index on the **dlstate** state column where **dlstate** = "VA" will make subsequent queries much faster. For this example we will focus on Virginia:



The command for creating a Partial Index is the same command for creating a traditional index with an additional WHERE filter at the end:

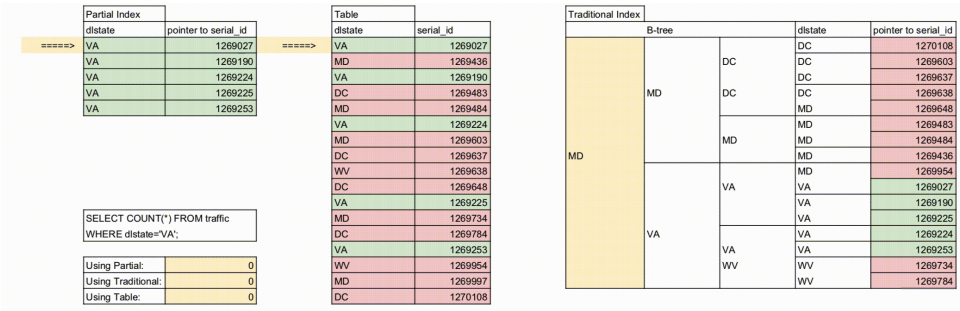
```
CREATE INDEX idx_dlstate_va ON traffic (dlstate) WHERE dlstate='VA';

ANALYZE;
```

```
bigdb=# CREATE INDEX idx_dlstate_va ON traffic (dlstate) WHERE dlstate = 'VA';
CREATE INDEX
Time: 1166.617 ms (00:01.167)
bigdb=# ANALYZE;
ANALYZE
Time: 27665.924 ms (00:27.666)
```

Let's now look at querying a sample of the data to see why Partial Indexing is much faster than querying the full table or an indexed version of the table.

Lets use a subset of the data and run the following query:

```
SELECT COUNT *
FROM traffic
WHERE dlstate='VA';
```



- Partial Index only has to move through rows where **dlstate** = VA.
- No index has to move through every row to find each place where **dlstate** = VA.
- Traditional index has the data sorted on dlstate but still has to traverse the b-tree to find where the rows where **dlstate** = VA are.

Remember to **ANALYZE;** after creating an index. **ANALYZE;** will gather statistics on the index so that the query planner can determine which index to use and how best to use it.

NOTE: Indexing will lock out writes to the table until it is done by default. To avoid this, create the index with the CONCURRENTLY parameter:

```
CREATE INDEX CONCURRENTLY [index name]
ON [table name (column name(s))] [WHERE [Filter]];
```

Time Comparisons

Now that the index has been created, and we have an understanding as to how the different types of indexes will be traversed let's compare the query times where there is no index, a full index, and a partial index on the full data set:

No Index

- The speed of running a COUNT aggregation where the dlstate='VA' with a **No Index** is:

```
EXPLAIN ANALYZE SELECT COUNT(*)
FROM traffic WHERE dlstate='VA';
```

```
Finalize Aggregate  (cost=91767.19..91767.20 rows=1 width=8) (actual time=1779.005..1779.005 rows=1 loops=1)
  ->  Gather  (cost=91766.97..91767.18 rows=2 width=8) (actual time=1778.904..1780.918 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate  (cost=90766.97..90766.98 rows=1 width=8) (actual time=1755.363..1755.363 rows=1 loops=3)
              ->  Parallel Seq Scan on traffic  (cost=0.00..90712.48 rows=21795 width=0) (actual time=0.187..1751.305 rows=17050
                    Filter: ((dlstate)::text = 'VA'::text)
                    Rows Removed by Filter: 490256
Planning Time: 1.957 ms
Execution Time: 1782.121 ms
```

Speed: 1.79 sec or 1784 ms

With Traditional Index

- The speed of running a COUNT aggregation where the dlstate='VA' with a **Traditional Index** is:

```
EXPLAIN ANALYZE SELECT COUNT(*)
FROM traffic WHERE dlstate='VA';
```

Index Scan Using Traditional Index

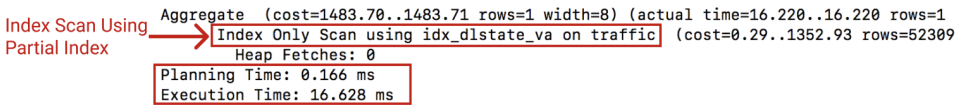
```
Aggregate  (cost=1566.17..1566.18 rows=1 width=8) (actual time=59.470..59.470 rows=1 loops=1)
  ->  Index Only Scan using idx_dlstate on traffic  (cost=0.43..1439.95 rows=50487 width=0)
        Index Cond: (dlstate = 'VA'::text)
        Heap Fetches: 0
Planning Time: 0.217 ms
Execution Time: 59.534 ms
(6 rows)
```

Speed: 0.06 sec or 59.7 ms

With Partial Index

- The speed of running a COUNT aggregation where the dlstate='VA' with a **Partial Index** is:

```
EXPLAIN ANALYZE SELECT COUNT(*)
FROM traffic WHERE dlstate='VA';
```



Speed: 0.02 sec or 17.8 ms

Speed Comparison:

Speed Comparisons:	Planning Time (ms)	Execution Time (ms)	Total Time (ms)
No Index	1.957	1782.121	1784.078
Traditional Index	0.217	59.534	59.751
Partial Index	0.166	16.628	16.794

As this table shows that, while adding an index of either variety is a significant improvement, a partial index is roughly 3.5 times faster than a traditional index in this situation.

Advanced Partial Indexes

Partial indexes can become more advanced.

Like Traditional indexes:

- Can be Multi-column
- Can use Different structures
    - E.g. B-tree, GIN, BRIN, GiST, etc.

Unique to Partial Indexes

- Can use complicated filters
- Smaller storage cost
- More Specific than Traditional Index

Complicated Filters

It is important to balance how specific the partial index with the frequency of queries that can use it. If the partial index is too specific, it will not be used often and simply be a waste of memory.

For example, a partial index could be created on a column with multiple filters such as the column 'arresttype' where the incident takes place from 4-4:30AM and zipcodes='12':

```
bigdb=# CREATE INDEX time_4_to_430_zip_12_idx ON traffic(arresttype)
WHERE zipcodes='12' AND timeofstop BETWEEN '04:00:00' AND '04:30:00';
```

This would significantly speed up some queries, however this partial index is so specific it may never be used more than a few times.

Sometimes however, if a specific set of filters are used a lot it can dramatically increase performance. This is what makes partial indexes so powerful. If there are a large number of queries regarding specific times between 9am-5pm, we can create an index on serial_id for these times.

```
bigdb=# CREATE INDEX time_9to5_idx ON traffic(serial_id)
WHERE timeofstop BETWEEN '09:00:00' AND '17:00:00';
```

This new index will make filtering by times between 9am and 5pm much quicker. This includes times inside this range as well such as 12:00-1:00 pm.

Before Index:

```
Finalize Aggregate  (cost=93824.87..93824.88 rows=1 width=8) (actual time=5009.635..5009.635 rows=1 loops=1)
  ->  Gather  (cost=93824.66..93824.87 rows=2 width=8) (actual time=5009.151..5011.522 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate  (cost=92824.66..92824.67 rows=1 width=8) (actual time=5001.398..5001.398 rows=1 loops=3)
              ->  Parallel Seq Scan on traffic  (cost=0.00..92294.57 rows=212036 width=0) (actual time=2.470..4976.633 rows=173209 loops=3)
                    Filter: ((timeofstop >= '09:00:00'::time without time zone) AND (timeofstop <= '17:00:00'::time without time zone))
                    Rows Removed by Filter: 334097
Planning Time: 0.419 ms
Execution Time: 5013.638 ms
```

After Index:

```
Finalize Aggregate  (cost=11787.53..11787.54 rows=1 width=8) (actual time=244.470..244.470 rows=1 loops=1)
  ->  Gather  (cost=11787.31..11787.52 rows=2 width=8) (actual time=244.325..246.522 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate  (cost=10787.31..10787.32 rows=1 width=8) (actual time=222.059..222.060 rows=1 loops=3)
              ->  Parallel Index Only Scan using time_9to5_idx on traffic  (cost=0.42..10257.22 rows=212036 width=0)
                    Heap Fetches: 0
Planning Time: 8.967 ms
Execution Time: 247.785 ms
```

Here the execution time drops from 5013 ms to 247 ms (~20x faster with index) which shows that partial indexes can save time.

Partial Indexes are also usually less memory intensive than traditional indexes:

```
public | time_9to5_idx              | index | matt  | traffic        | 11 MB
public | time_idx                   | index | matt  | traffic        | 33 MB
```

The traditional version of the index is 3 times the size of the partial index. The trade off here is that the traditional index can improve a wide range of queries where as the partial index is more specific, but also faster.

Summary:

- Partial indexes only store information specified by a filter
- Partial indexes can be very specific
    - This can be good, but make sure to balance practically and memory size. Too specific an index becomes unusable.
- Partial indexes save space compared to traditional indexes

References:

1. https://www.postgresql.org/docs/11/sql-createindex.html
2. https://www.youtube.com/watch?v=clrtT_4WBAw

Multicolumn Indexes

Multicolumn indexes (also known as composite indexes) are similar to standard indexes. They both store a sorted “table” of pointers to the main table. Multicolumn indexes however can store additional sorted pointers to other columns.

Standard indexes on a column can lead to substantial decreases in query execution times as shown in this article on optimizing queries. Multi-column indexes can achieve even greater decreases in query time due to its ability to move through the data quicker.

Syntax

```
CREATE INDEX [index name]
ON [Table name]([column1, column2, column3,...]);
```

Multicolumn indexes can:

- be created on up to 32 columns
- be used for partial indexing
- only use: b-tree, GIN, BRIN, and GiST structures

Video

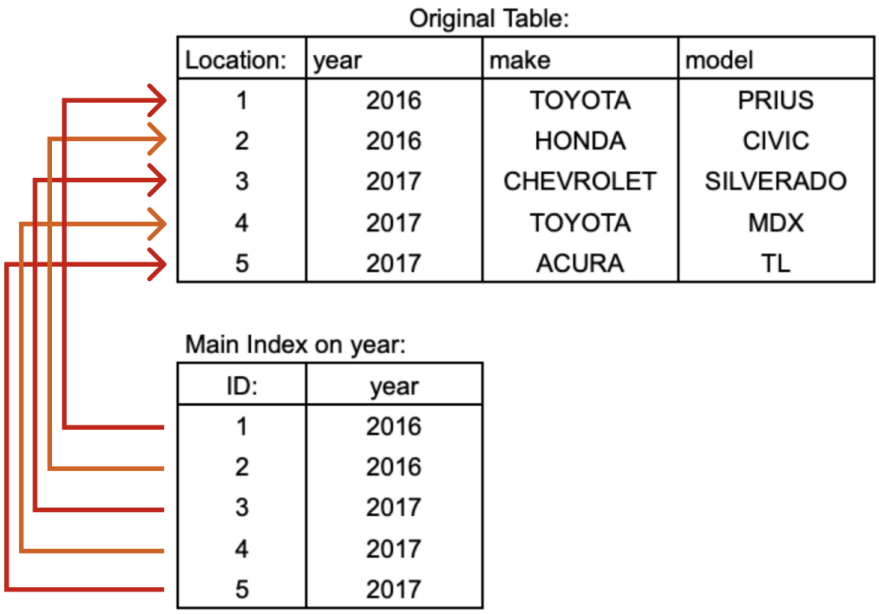
What is a multicolumn index?

Multicolumn indexes are indexes that store data on up to 32 columns. When creating a multicolumn index, the column order is very important. This is due to the structure that multicolumn indexes possess. Multicolumn indexes are structured to have a hierarchical structure. Take for example this table:

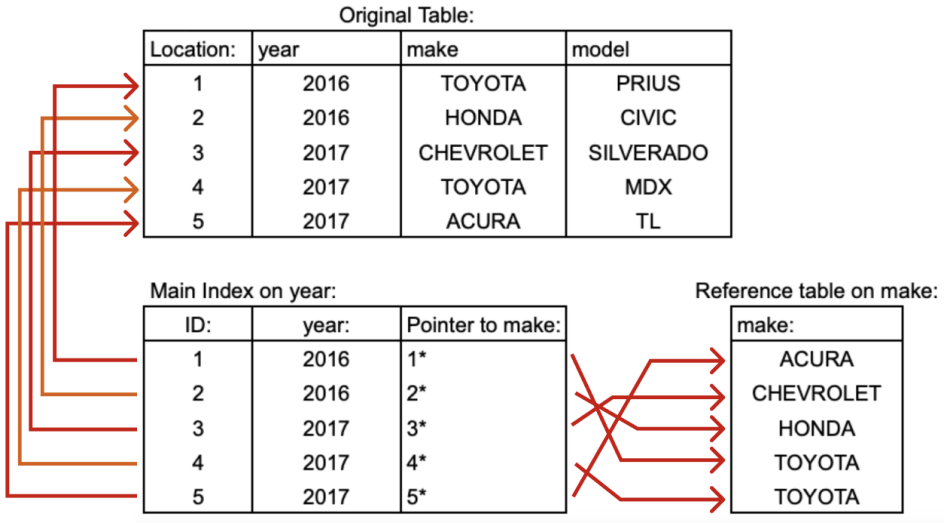
Original Table:

Location:	year	make	model
1	2016	TOYOTA	PRIUS
2	2016	HONDA	CIVIC
3	2017	CHEVROLET	SILVERADO
4	2017	TOYOTA	MDX
5	2017	ACURA	TL

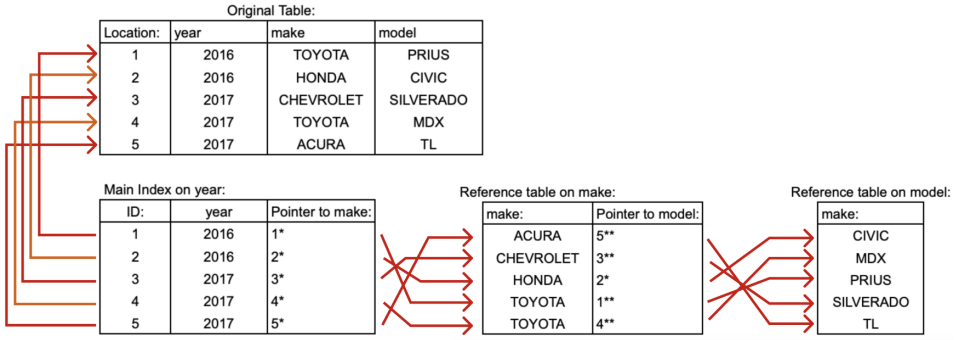
A traditional index on this table would look like this:



The index points back to the table and is sorted by year. Adding a second column to the index looks like this:



Now the index has pointers to a secondary reference table that is sorted by make. Adding a third column to the index causes the index to look like this:



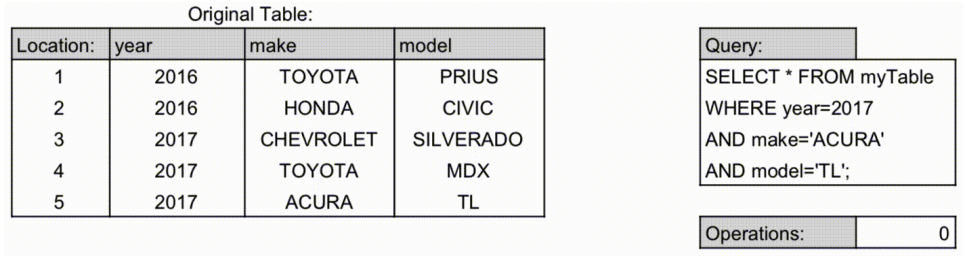
In a three column index we can see that the main index stores pointers to both the original table and the reference table on make, which in turn has pointers to the reference table on model.

When the multicolumn index is accessed, the main portion of the index (the index on the first column) is accessed first. Each entry in the main index has a reference to the row's location in the main table. The main index also has a pointer to the secondary index where the related make is stored. The secondary index in term has a pointer to the tertiary index. Because of this pointer ordering, in order to access the secondary index, it has to be done through the main index. This means that this multicolumn index can be used for queries that filter by just year, year and make, or year, make, and model. However, the multicolumn index cannot be used for queries just on the make or model of the car because the pointers are inaccessible.

Multicolumn indexes work similarly to traditional indexes. You can see in the gifs below how using a multicolumn index compares to using both a sequential table scan and a traditional index scan for the following query:

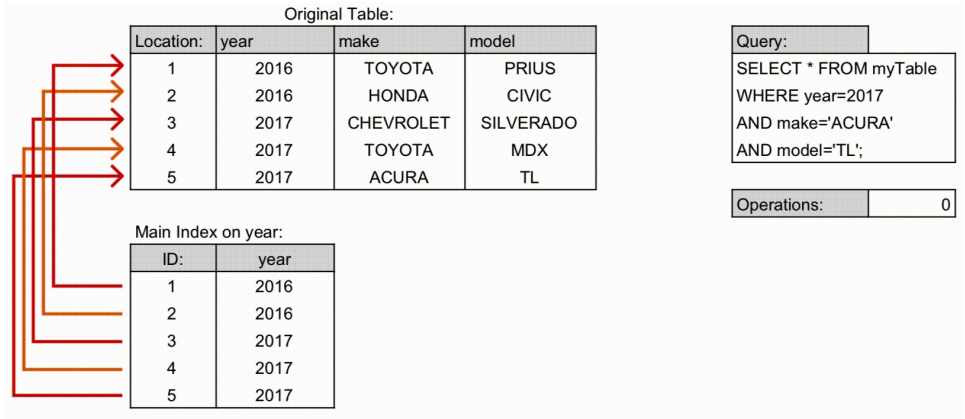
```
SELECT * FROM myTable
WHERE year=2017
AND make='ACURA'
AND model='TL';
```

Table Scan



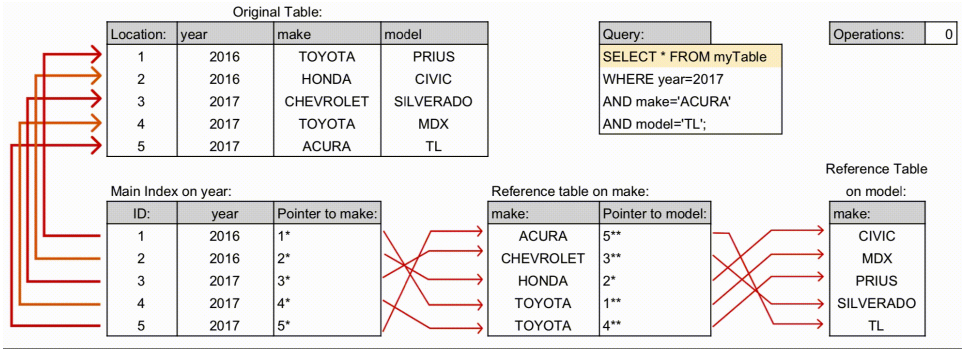
- Scans every row for correct entry or entries

Traditional Index



- Can filter out wrong years using the index, but must scan all rows with the proper year.

Multicolumn Index



- Can filter by all 3 columns allowing for much fewer steps on large data sets

From these gifs you can see how multicolumn indexes work and how they could be useful, especially on large data sets for improving query speeds and optimizing.

Performance

Multicolumn indexes are so useful because, when looking at the performance of a normal index versus a multicolumn index, there is little to no difference when sorting by just the first column. For an example look at the following query plans:

Standard Index:

```
Bitmap Heap Scan on traffic_data  (cost=1227.87..82702.65
  Recheck Cond: (year = '2001'::smallint)
  Heap Blocks: exact=28422
  ->  Bitmap Index Scan on standard_index_vehicle_year  (
        Index Cond: (year = '2001'::smallint)
Planning Time: 0.857 ms
Execution Time: 6070.473 ms
```

Multicolumn Index:

```
Bitmap Heap Scan on traffic_data  (cost=1571.87..83046.65
  Recheck Cond: (year = '2001'::smallint)
  Heap Blocks: exact=28422
  ->  Bitmap Index Scan on mult_col_index_vehicle  (cost=
        Index Cond: (year = '2001'::smallint)
Planning Time: 0.217 ms
Execution Time: 6263.815 ms
```

These two query plans show that there is little to no difference in the execution time between the standard and multicolumn indexes.

Multicolumn indexes are very useful, however, when filtering by multiple columns. This can be seen by the following:

Create standard index:

```
CREATE INDEX standard_index_vehicle_year ON traffic_data(year);
```

Create multicolumn index:

```
CREATE INDEX mult_col_idx_vehicle ON traffic_data(year, make, model);
```

Query run in Images:

```
EXPLAIN ANALYZE SELECT * FROM traffic_data
WHERE year='2001' AND make='CHEVROLET' AND model='TAHOE';
```

Table Scan:

```
Gather  (cost=1000.00..93466.34 rows=5 width=13)
  Workers Planned: 2
  Workers Launched: 2
  ->  Parallel Seq Scan on traffic_data  (cost=0
        Filter: ((year = '2001'::smallint) AND (
        Rows Removed by Filter: 507243
Planning Time: 2.300 ms
Execution Time: 4788.052 ms
```

Standard Index:

```
Bitmap Heap Scan on traffic_data  (cost=1144.67..81728.97 rows=4 width=13) (actual time=87.452.
  Recheck Cond: (year = '2001'::smallint)
  Filter: (((make)::text = 'CHEVROLET'::text) AND ((model)::text = 'TAHOE'::text))
  Rows Removed by Filter: 63591
  Heap Blocks: exact=28422
  ->  Bitmap Index Scan on standard_index_vehicle_year  (cost=0.00..1144.67 rows=61899 width=0)
        Index Cond: (year = '2001'::smallint)
Planning Time: 0.881 ms
Execution Time: 1820.427 ms
```

Multicolumn Index:

```
Index Only Scan using mult_col_index_vehicle on traffic_data  (cost=0.43..24.
  Index Cond: ((year = '2001'::smallint) AND (make = 'CHEVROLET'::text) AND (
  Heap Fetches: 192
Planning Time: 1.283 ms
Execution Time: 6.948 ms
```

	Execution Time:
Table Scan	4788.052 ms
Standard Index	1820.427 ms
Multicolumn Index	6.948 ms

The table above shows the execution times of each index on the given query. It shows clearly that, in the right situation a multicolumn index can be exactly what is needed.

Summary

- Multicolumn indexes:
  - Can use b-tree, BRIN, GiST, and GIN structures
  - Can be made on up to 32 columns
  - Can be used for Partial Indexing
- Perform comparably to traditional indexes on their single column
- Perform much better once additional Columns are added to the query.
- Column order is very important.
  - The second column in a multicolumn index can never be accessed without accessing the first column as well.

References

http://www.postgresqltutorial.com/postgresql-indexes/postgresql-multicolumn-indexes/

https://medium.com/pgmustard/multi-column-indexes-4d17bac764c5

https://www.bennadel.com/blog/3467-the-not-so-dark-art-of-designing-database-indexes-reflections-from-an-average-software-engineer.htm

Modeling Data

Start Modeling Data

Data Modeling sounds really scary, like a big chore and months of work.

But it is not so bad and you can get started in less than 10 minutes.

For this example we use BigQuery and dbt. BigQuery is one of Google's cloud database offerings. dbt which stands for Data Build Tool is a data modeling tool created by Fishtown Analytics.

https://cloud.google.com/bigquery/

BigQuery comes with a set of public data sets that are great for practicing data modeling on. I will be using the Stack Overflow data set they have.

You can start using Google Cloud's various services for free but you will need to upgrade the billing so that you can connect dbt to Google Cloud. If you have not signed up for Google Cloud platform services before they will give you a $300 credit (which is more than enough to run this test thousands of times) so don't worry about the costs in trying this out.

Installing dbt

To install you can visit their documentation page here:

https://docs.getdbt.com/docs/macos

Or you can follow along below, most of what is here is straight from their docs anyways.

I suggest using homebrew to install dbt, it makes it extremely easy. If you do not have homebrew open your terminal on your Mac and put in the following command.

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/mas
```

After it installs put the following commands into terminal.

```
brew update
brew tap fishtown-analytics/dbt
brew install dbt
```

You will also need git:

```
brew install git
```

Create a folder on your computer (I named my dbt Projects). We are going to populate it with all the files and subfolders dbt needs to get started. We do this by navigating into that folder through the terminal.

Then we run the following inside of terminal.

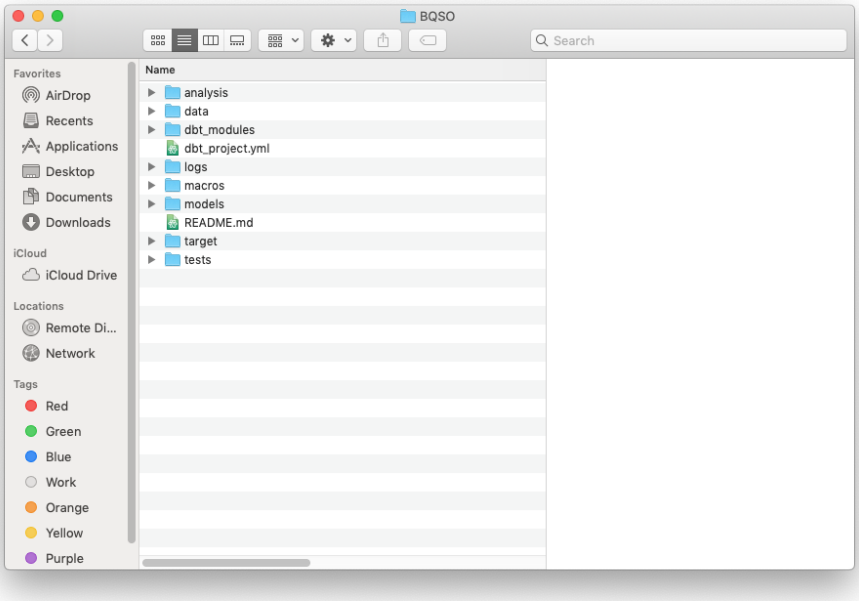
```
dbt init [name of project]
```

I called mine BQSO, so my terminal command looked like this:

```
dbt init BQSO
```

Navigate inside the folder to see all the folders and files dbt created for us

```
cd BQSO
```



Configuring BigQuery

To get dbt to work with BigQuery we need to give it permission. The way to do this is by setting up profile (think account) with login information. Basically you have to create a profile in dbt's folder and then you will link that profile to this specific DBT project that you just created.

Go to dbt's profiles (a sample profiles.yml file was created when we ran the dbt init command)

```
open /Users/[your username]/.dbt
```

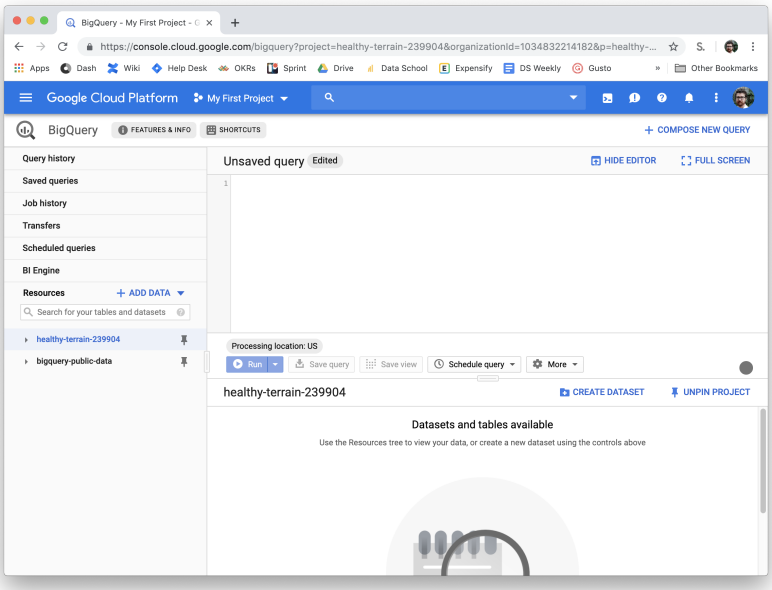
This will pop open a file called profiles.yml which is the most challenging part of this tutorial. Configuring the profiles yml file. As a starter you can copy paste the code below to replace what is in the file, replacing one field with your own information.

```
my-bigquery-db:
target: dev
outputs:
dev:
type: bigquery
method: service-account
project: happy-vegetable-211094
dataset: soCleaned
threads: 1
keyfile: /users/matt/BigQuerykeyfile.json
timeout_seconds: 300
```

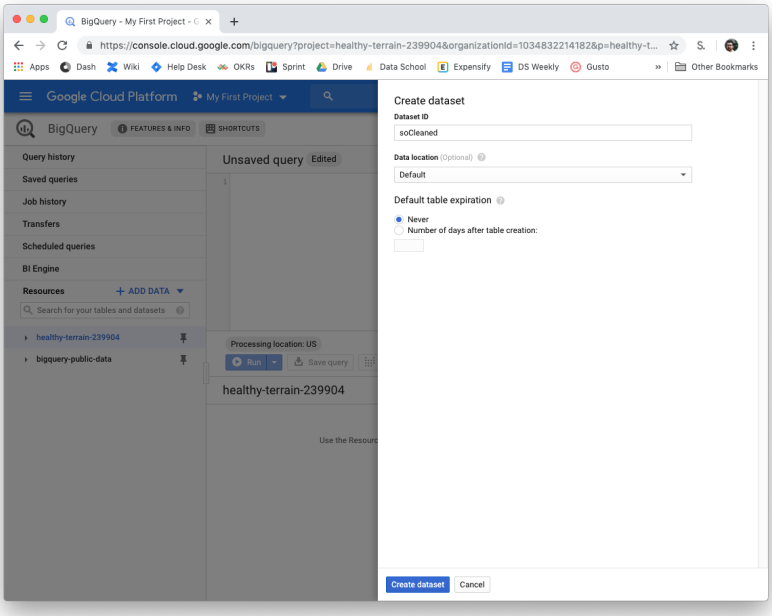
Now I will mark where you will need to update with your own info with **bold**.

- `my-bigquery-db:`

    - This is the name which will be used to link the profile (account details/login info) to the project.
    - I think this name makes sense but feel free to change it to whatever name you would like.
- `type: bigquery`
    - The type of database, no surprises here.
- `method: service-account`
    - How you will connect. This is specific to the database chosen, for bigquery this is how you do it.
- `project: healthy-terrain-239904`
    - **update with your own info**
    - This is the project name, it will be a weirdly named thing inside of BigQuery on the left.

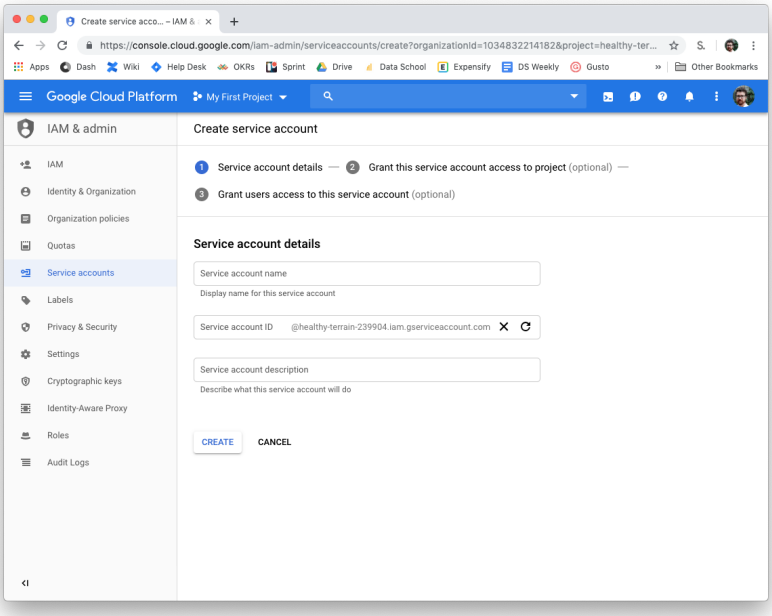


    - Replace **healthy-terrain-239904** with your project name.
    - You may need to create a new project in BigQuery but there should be a default one which is fine to use for this example
- `dataset: soCleaned`
    - **update with your own info**
    - The name of Schema (Schema are called datasets in BigQuery) you will be putting the modeled data in



    - Inside of BigQuery click on your project (healthy-terrain-239904)

    - On the right you will see Create Data Set, click that
    - The Dataset ID will be the name of the schema
    - Replace **soCleaned** with your schema name that you put in the Dataset ID
- `keyfile: /users/matt/BigQuerykeyfile.json`
    - **update with your own info**
    - You do this by going to IAM & admin in BigQuery (hidden in the hamburger menu on the left)



    - Click Service Accounts
    - Click Create Service Account
    - Create a name for it (a name like dbt)
    - Select Role - BigQuery Admin
    - Create key - JSON
    - This will download the key to your computer
    - You will need to put the file path in the yml file so place it somewhere that makes sense to you

Once you update all of those fields in your dbt profile (profiles.yml) you now need to link that profile to the project we created.

Go to the project folder we had created earlier (BQSO in my case) and open the yml file inside of it.

dbt_project.yml

Now you only need to update one thing in this file, you need to se the profile to the name we just created:

```
profile: 'my-bigquery-db'
```

This is the link to the profile we just created, so if you changed that name to something else replace ‘my-bigquery-db’ with whatever you created. It does need the single quotes around the name of the profile.

Creating a New Table with Modeled Data

Go to the models folder in your project and create a new .sql file. In that .sql file you can write a SQL statement that’s output will be modeled data. Try adding this text to the .sql file and save it:

```
{{ config(materialized='table') }}
SELECT *
FROM 'bigquery-public-data.stackoverflow.posts_questions'
```

```
ORDER BY view_count DESC
LIMIT 1
```

Whatever you named the .sql file will be the name of the table in the schema (dataset) In my case I saved it as firstModel.sql

Running your Models

Go to terminal, make sure you are in the project folder of the dbt project and type

```
dbt run
```

Boom, refresh BigQuery and see the new table. You can query it with a simple.

```
SELECT *
FROM soCleaned.firstModel
```

Can you believe the most viewed post is about git? Classic.

You have now modeled data and queried modeled data. Not so scary right?

Why did we did we model this in the first place?

Well querying this modeled data took 0.2 seconds and processed 473bytes (granted this is just a single row with 20 columns)

When I do this query on the full Stack Overflow data set it took 20.3 seconds and processed 26.64 GB

How did this happen? We moved the larger query which operated on the full Stack Overflow data set to occur when we typed dbt run. dbt created a table with the results of that query which has one row of data. We then can query this new table which is much smaller through big query and get our result much faster. In fact we can query it as many times as we want without incurring the cost of the large query again.

Note: We will have to do dbt run again if we want to load the latest data into the modeled table. dbt run is often done on a schedule (typically at night) so that users know how up to date the data is they are dealing with.

There are many other things we can do with modeling and dbt such as cleaning up data, simplifying the schema, or find other ways to get more performance out of a query. Stay tuned for more!

Scheduling Modeling

Dbt is a great tool for creating, storing, and then running queries against a database. These queries can be for any purpose but we will be talking about how they can be used to create and update simplified tables and views. This allows you to create a set of table and views that are more easily queryable by the rest of your organization so they can find insights faster. Making data easier to use is an important piece of optimizing your SQL and Data Warehouse because this is where most of the time is spent, determining the right query to run.

You can automate these queries that simplify the data to run on a schedule with dbt Cloud. In order to use dbt Cloud you need to set up dbt linked to a GitHub repository and a data warehouse. If you have not done this already you can check out The Data School's article on dbt and BigQuery here.

Setting up dbt Repository

Once you have set up your dbt and BigQuery, you need to set up your GitHub repository to store your dbt files. To do this, first go into your GitHub and create a new repository. Copy the link to that new repository and save it, you will need it to link your dbt files.

Now, in your dbt folder run these commands:

'git init.' 'git add .' 'git commit -m 'commit message'' 'git remote add (the URL you saved)' 'git push -u origin master'

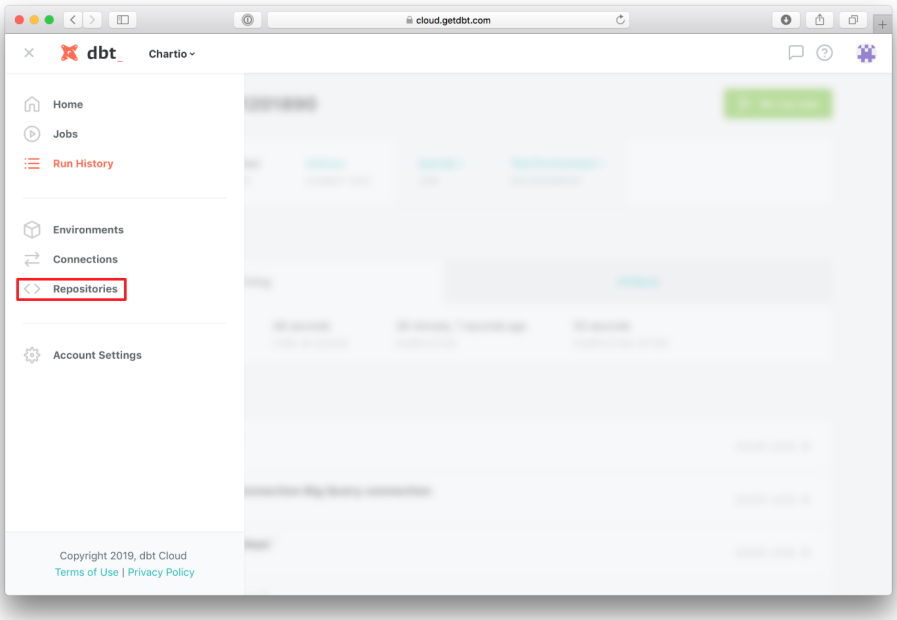
After this you can refresh your GitHub account to make sure you have all your files and you are ready to begin scheduling dbt.

Using dbt Cloud

Once you have created your dbt Cloud account there are just a few steps between you and automated dbt runs.

1. Link your dbt repository to your dbt Cloud account

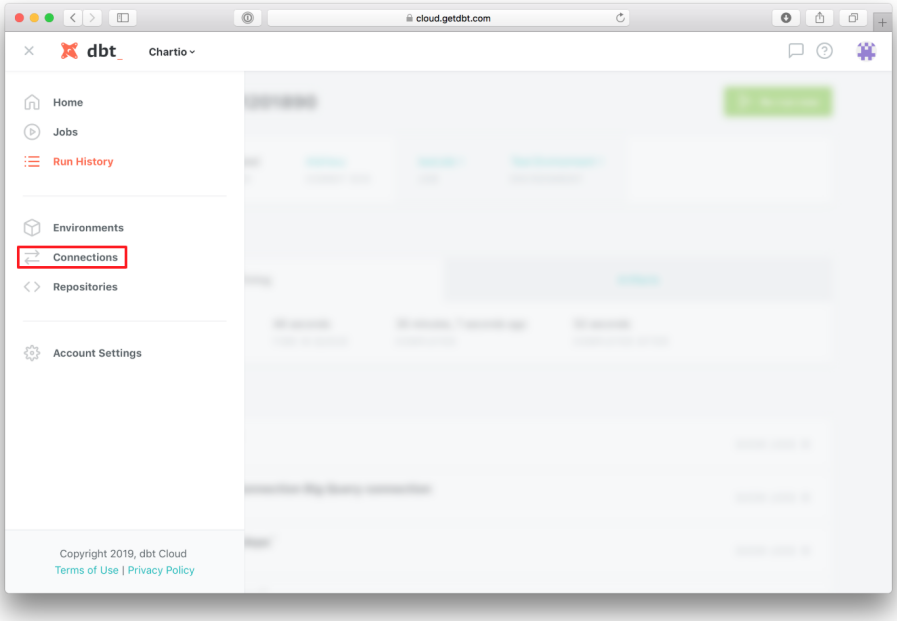
In the side menu select "Repositories":



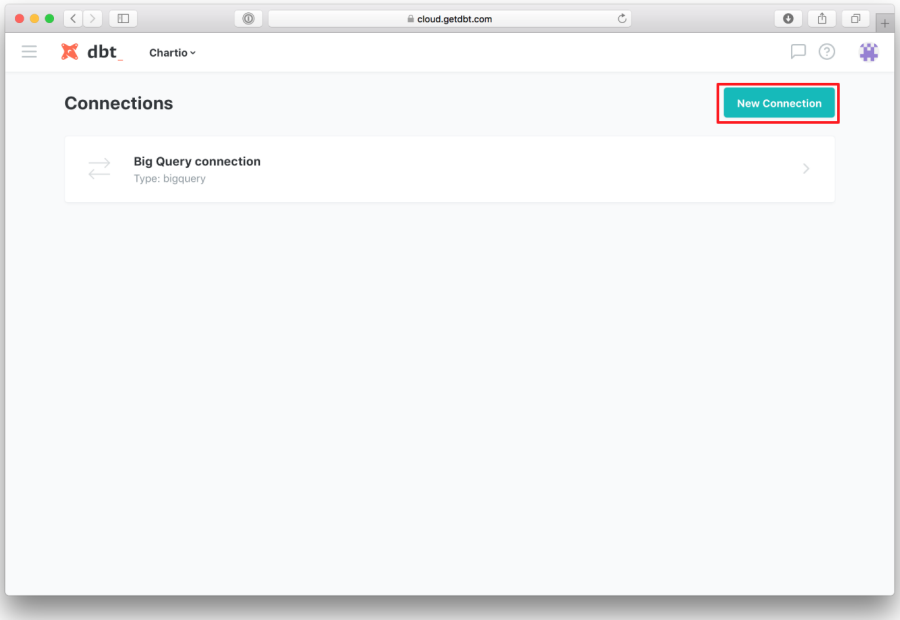
Click "Add New Repository" and link your GitHub account to your dbt account.

1. Create a dbt Cloud connection

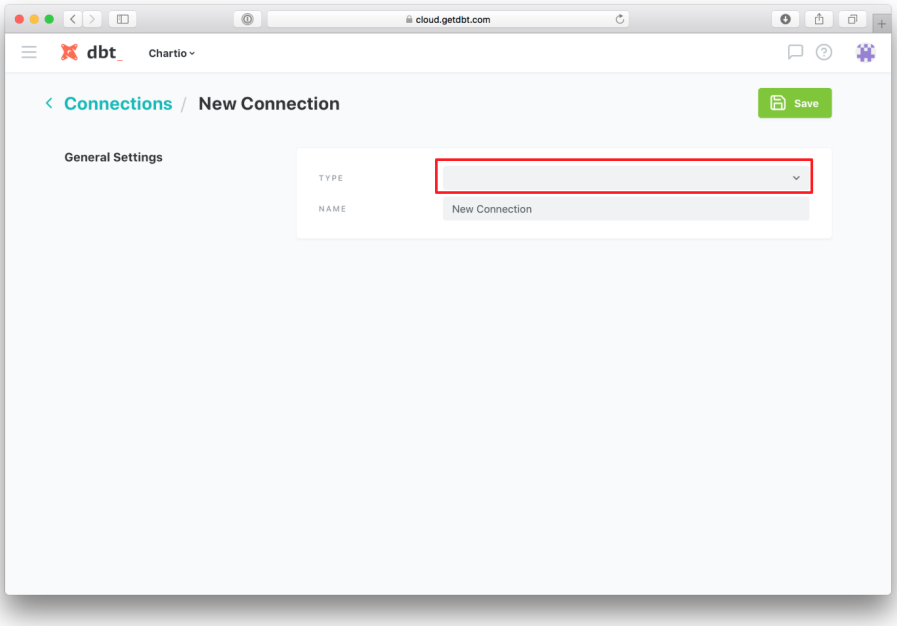
In the side menu select “Connections”:



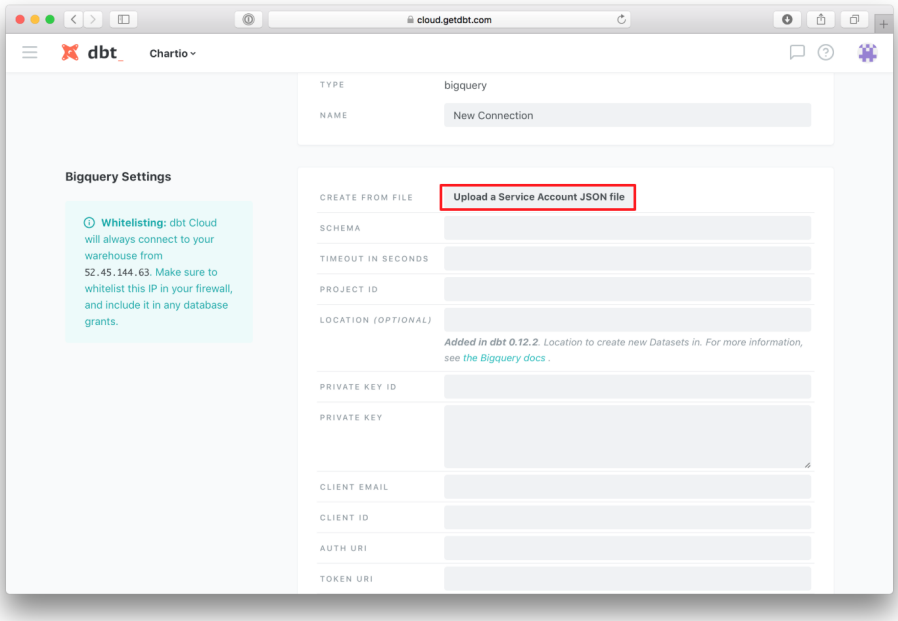
Then click “New Connection”:



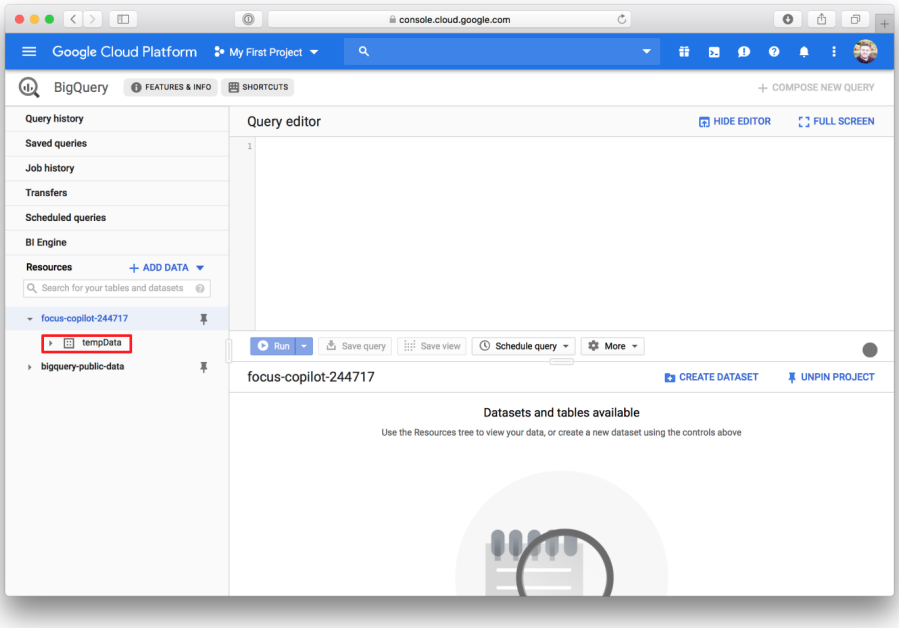
In the “Type” drop down, select the type of database you want to connect. For this example, we will connect a BigQuery database:



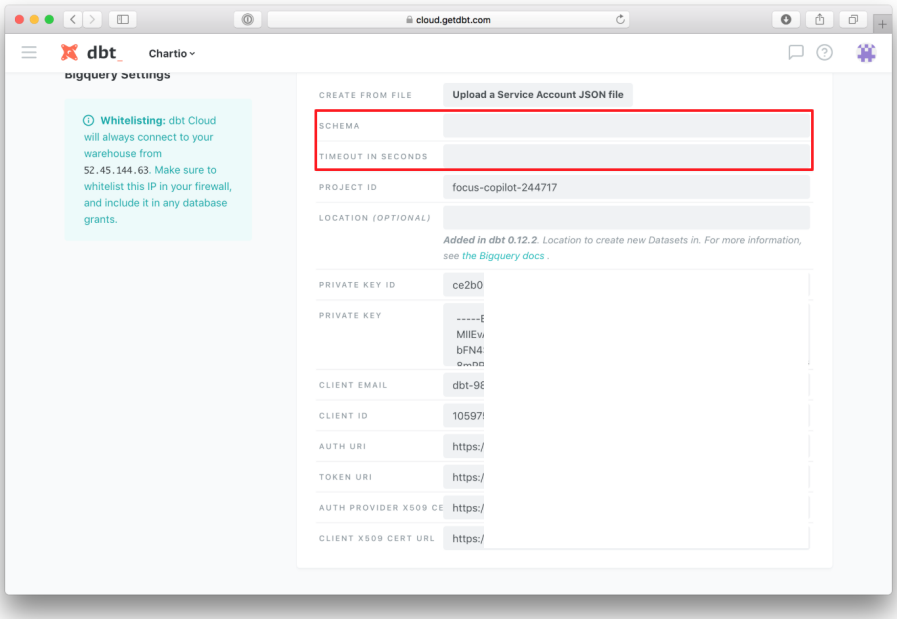
The easiest way to fill in the information to build your “Connection” is to “Upload a Service Account JSON file” that you used when linking dbt to your BigQuery account:



Now the only information you should have to fill out for yourself is the “Schema”, the name of the table in your BigQuery to connect to:



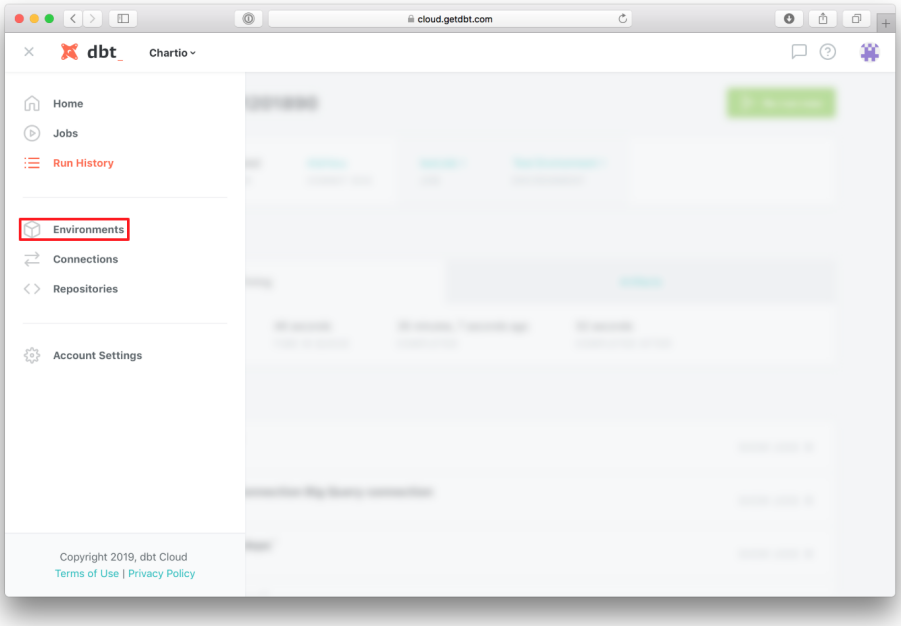
and your desired “Timeout in Seconds”, the length of time your query is allowed to run before the system terminates it:



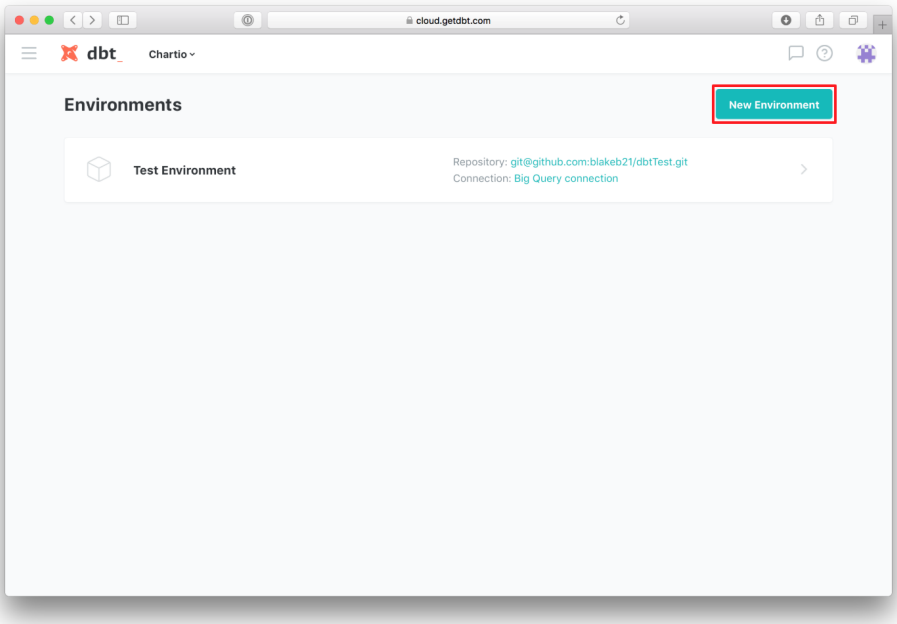
Once you have filled out those two fields, you can press “Save” in the top right.

1. Create a dbt Cloud environment

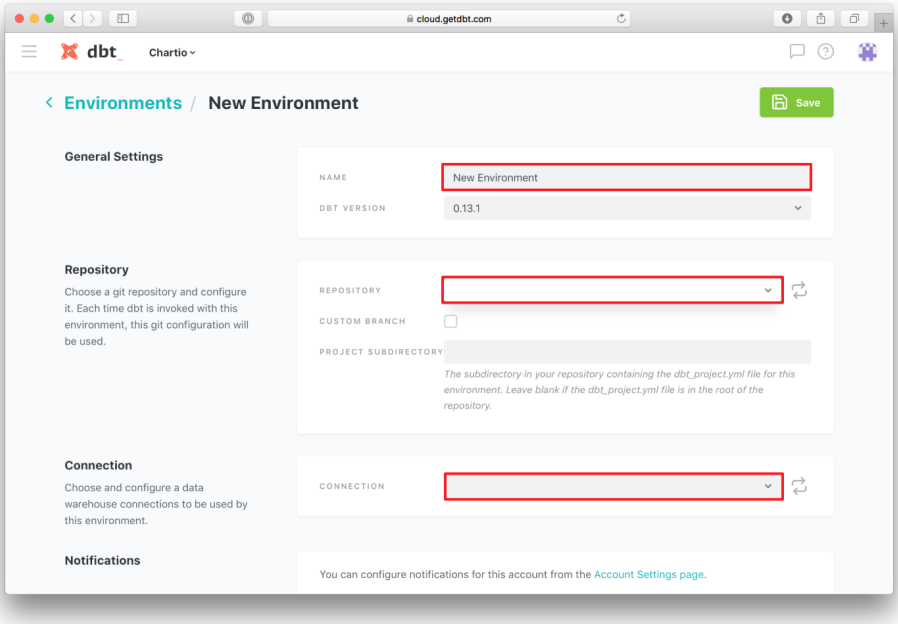
In the side menu select “Environments”:



Then select “New Environment” to create your environment:



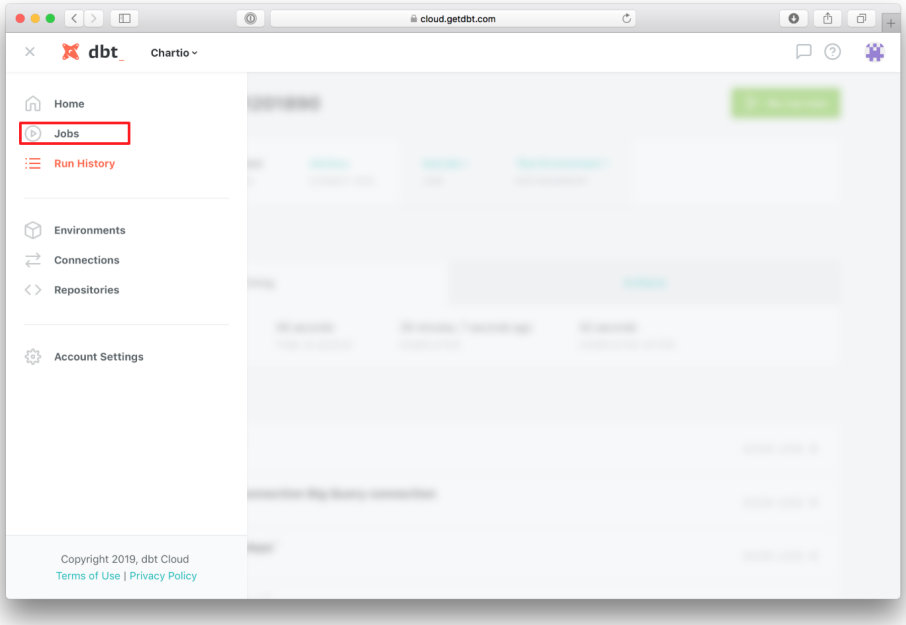
Then begin filling in the required information. Give your “Environment” a name, select the repository you would like to link it to, and select the database “Connection” you would like this environment to have:



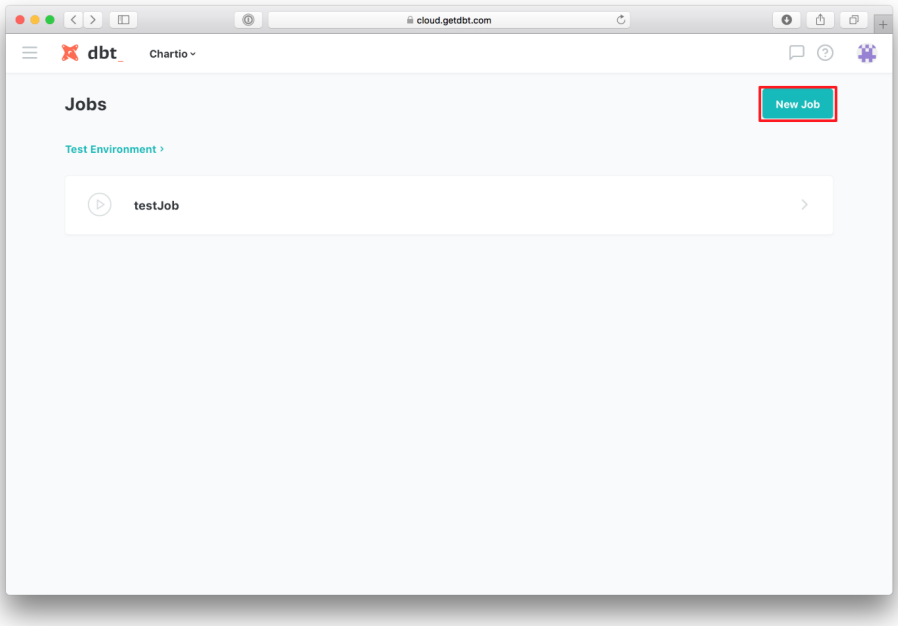
Once you have selected your desired options from the drop down menus, click “Save” to save this “Environment”.

1. Schedule your first job

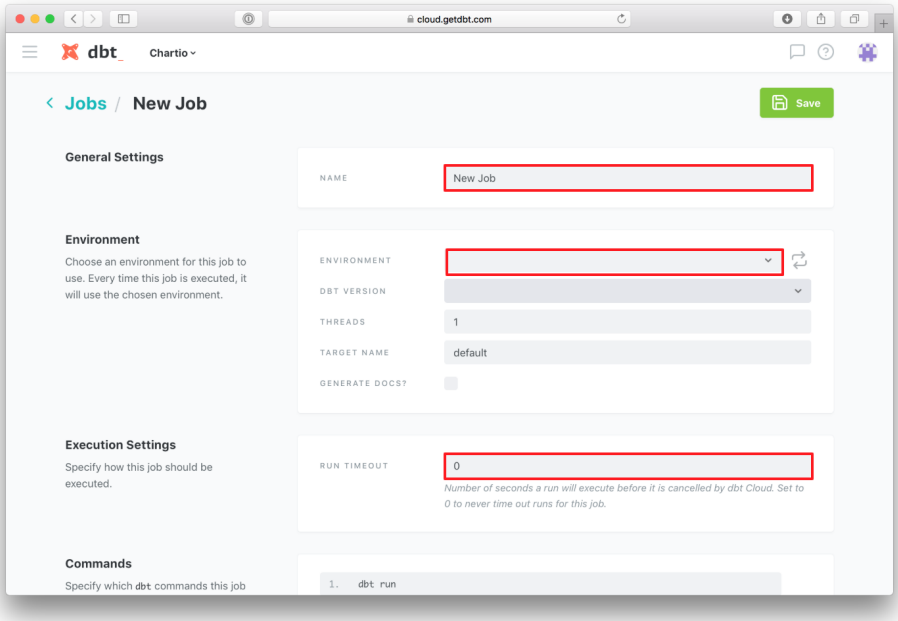
To schedule your first dbt query, select “Jobs” from the side menu:



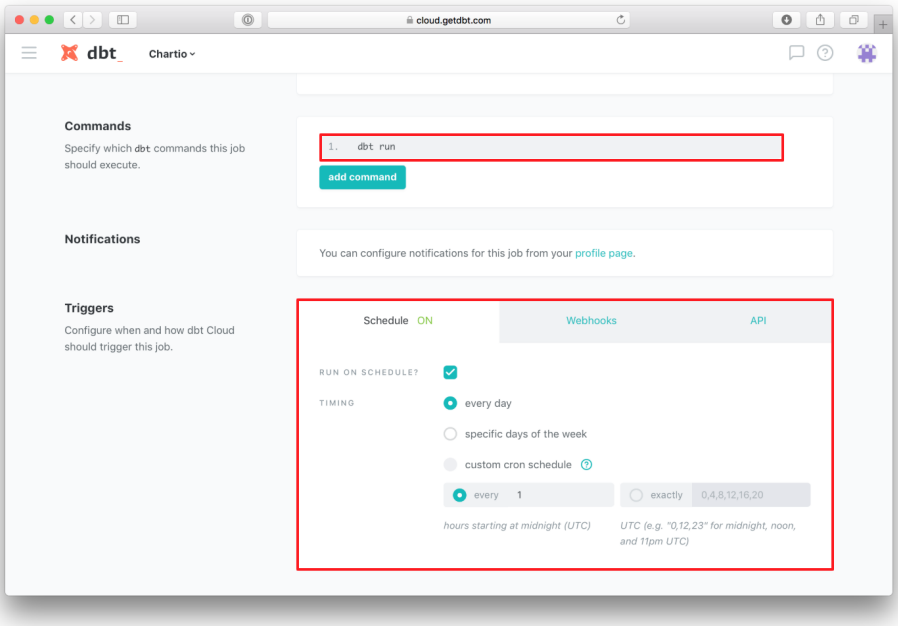
Click on “New Job”:



Give your new “Job” a name, select the “Environment” you want to automate, you can also change the “Run Timeout” by default the query will never timeout:



Next, you can tell the “Job” which dbt commands to run, ‘dbt run’ is used by default. Finally, you can tell the “Job” when you want it to run:



Once you have filled in all the required fields, you can click “Save” in the top right and the job will begin running on schedule. Now your queries will be regularly updated and ready to use by others in your organization.

Summary

- Create your dbt and BigQuery instances
- Link your dbt files to a GitHub repository
- Link your GitHub repository to dbt Cloud
- Connect your database to dbt Cloud
- Create an “Environment” linking your repository and database
- Create a “Job” to automatically run your “Environment”

References

https://cloud.getdbt.com

Views

What is a view?

Views are a way to store a long query for easier access. When a view is created, it stores a query as a keyword, which can be used later instead of typing out the entire query. Long and complicated queries can be stored under a single name which allows them to be used easily.

For example, instead of typing:

```
SELECT state, vehicletype, year, make, model, color FROM traffic
WHERE state='MD';
```

Every time that details about a vehicle are needed, you can create a view:

```
CREATE VIEW vehicle_details AS
SELECT state, vehicletype, year, make, model, color FROM traffic
WHERE state='MD';
```

Once a view is created, the details about the vehicle can be accessed through the view:

```
SELECT * FROM vehicle_details;
```

Query without using the view	bigdb=# SELECT state, vehicletype, year, make, model, color FROM traffic WHERE state='MD' LIMIT 5; state | vehicletype | year | make | model | color MD | 02 – Automobile | 2006 | NISSAN | TK | BLACK MD | 02 – Automobile | 2002 | BMW | X5 | SILVER MD | 02 – Automobile | 2005 | HOND | CIV 2S | SILVER MD | 02 – Automobile | 2002 | BMW | X5 | SILVER MD | 02 – Automobile | 2002 | BMW | X5 | SILVER (5 rows)
Query which is using the view	bigdb=# SELECT * FROM vehicle_details LIMIT 5; state | vehicletype | year | make | model | color MD | 02 – Automobile | 2006 | NISSAN | TK | BLACK MD | 02 – Automobile | 2002 | BMW | X5 | SILVER MD | 02 – Automobile | 2005 | HOND | CIV 2S | SILVER MD | 02 – Automobile | 2002 | BMW | X5 | SILVER MD | 02 – Automobile | 2002 | BMW | X5 | SILVER (5 rows)

bigdb=# SELECT state, vehicletype, year, make, model, color FROM traffic WHERE state='MD' LIMIT 5;
bigdb=# SELECT * FROM vehicle_details LIMIT 5;

As we can see, the two queries return the same result. The only difference between the two queries is the length of the queries in terms of characters.

Creating a view

Creating a view follows this form:

```
CREATE VIEW [view name] AS [SELECT statement/Query to store]
[(optional) WHERE [condition]];
```

In the first example, a view was created on the details of a vehicle. For this view, the name vehicle_details was used and the query used to create the view was:

```
SELECT state, vehicletype, year, make, model, color FROM traffic
WHERE state='MD';
```

The view will store the query above. This means that when the view is used, the query that is stored in the view will be accessed and run. In other words running a standard view is no different from running the query it was created on in terms of execution. The only difference is the length of the query that needs to be written by the user. As such, creating views is mainly for simplifying the writing of queries, not the running of queries.

They can also be used to allow users or groups access to only specific sections of a table or database without allowing access to the entire thing. Limiting columns in a view will produce some performance improvements on SELECT * queries since the amount of table being pulled is less. However this is not justification for creating views of every column

combination so that SELECT * can always be used. In fact it is a better practice to discourage the use of SELECT * and have people query specifically for the columns they care about because then the least amount of data is being pulled on every query.

Using Views

Views can be used in a variety of ways and with several optional parameters:

TEMP/TEMPORARY

Adding TEMP or TEMPORARY to the creation of a view creates a view that is automatically dropped at the end of the user's session.

Example:

```
CREATE TEMP VIEW myView AS SELECT serial_id FROM traffic;
```

WITH CHECK OPTION

Adding 'WITH CHECK OPTION' to the end of a CREATE VIEW statement ensures that, if the view is updated, the update does not conflict with the view. For example, if a column is created on a view where dlstate must be 'MD', then the user cannot INSERT a row into the view where the dlstate is 'VA.'

It will return an error :

```
ERROR: new row violates check option for view [name of view]
```

Example:

```
bigdb=# CREATE VIEW testview AS SELECT seqid, serial_id, dlstate
bigdb-# FROM traffic WHERE dlstate='MD' WITH CHECK OPTION;
 CREATE VIEW
bigdb=# INSERT INTO testview VALUES ('1', '1', 'VA');
 ERROR:  new row violates check option for view "testview"
```

LOCAL and CASCADED

Adding LOCAL or CASCADED to CHECK OPTION will designate the scope for the CHECK OPTION. If LOCAL is added, the CHECK only applies to that specific view. CASCADED on the other hand, applies the CHECK to all views that the current view is dependent on.

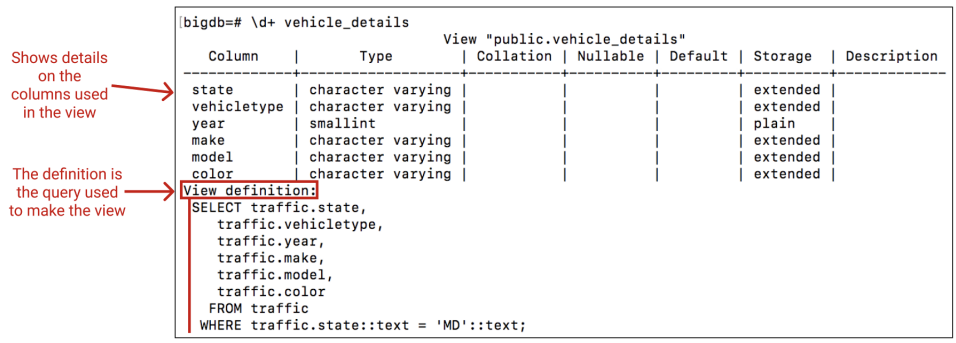
Example:

```
CREATE VIEW myView AS SELECT serial_id FROM traffic
WITH LOCAL CHECK OPTION;
```

VIEW definition

To see the definition (underlying query) of a view, you can use:

```
\d+ [view name]
```



Updating Views

Views can be updated by using the following syntax:

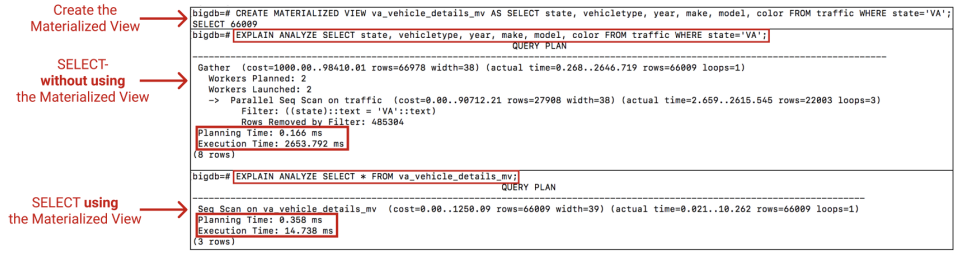
```
UPDATE [Name of View]
SET [Column Name]=[Value to set to], [Column Name]=[Value to set to], etc
WHERE [condition];
```

Views can only be updated if they follow these criteria:

- The view must not be created with an aggregate as a column
- The view must not contain: DISTINCT, GROUP BY, or HAVING in its definition
- The view must be on only one table
- If the view is built on another view, that view must the criteria above

Materialized Views

Materialized views are similar to standard views, however they store the result of the query in a physical table taking up memory in your database. This means that a query run on a materialized view will be faster than standard view because the underlying query does not need to be rerun each time the view is called. The query is run on the new Materialized view:



This query plan shows the materialized view being used as a table and being scanned. It also shows a significant difference in speed between the two methods.

Materialized views are generally most effective on computation-intensive views. Materializing a view has several benefits but also several drawbacks.

Pros:

- Faster - Aggregations and Joins are run beforehand
- Can be Indexed for even greater speed
- Does not update after each run: saves server's time by preventing unnecessary refreshing

Cons:

- Takes more memory
- Does not update after each run: Must be refreshed so data may not be completely up to date
- Must be refreshed one of two ways:
    - Manually:
        - Completely Recalculate entire view (possibly expensive)
        - Supports all SQL
    - Incrementally:
        - Does not support some functions and outer joins
        - Only update specific rows
        - Note: Not all SQL types support this. (e.g. can be done in Oracle and DBT, but not PostgreSQL.)

To create a materialized view, add the MATERIALIZED keyword:

```
CREATE MATERIALIZED VIEW myView AS [Query];
```

To refresh the view manually use the following command:

```
REFRESH MATERIALIZED VIEW [view name]
```

Summary

- Standard Views
  - Store a query that can be referenced by the name of the view
  - Mostly for ease of use, but can also be used for security/permissions purposes
  - Has Options:
    - TEMP/TEMPORARY
    - WITH CHECK OPTION
    - CASCADED/LOCAL
- Materialized Views
  - Like standard views but store results in a table.
  - Can be indexed
  - Must be refreshed
    - Manual Full refresh: Refreshes all values
    - Incremental Refresh: Only refreshes modified values
  - Requires more memory

References

http://www.postgresqltutorial.com/postgresql-views-with-check-option/

https://www.percona.com/blog/2011/03/23/using-flexviews-part-one-introduction-to-materialized-views/

https://www.tutorialspoint.com/sql/sql-using-views.htm

https://www.postgresql.org/docs/9.2/sql-createview.html

Databases

Redshift Optimization

What is Redshift?

Redshift is a fully managed, columnar store data warehouse in the cloud hosted by Amazon Web Services(AWS). Redshift can handle petabytes of data and is accessible 24/7 for their customers.

Redshift has many advantages for companies looking to consolidate their data all in one place. It is fully managed, very fast, highly scalable and is a part of the high used AWS platform.

Fully Managed

Amazon Redshift is fully managed, meaning that Redshift does all of the backend work for their customers. This includes setting up, managing, and scaling up the database. This means that Redshift will monitor and back up your data clusters, download and install Redshift updates, and other minor upkeep tasks.

This means data analytics experts don't have to spend time monitoring databases and continuously looking for ways to optimize their query performance. However, data analysts do have the option of fully controlling Redshift, this just means they have to spend the time to learn all of Redshift's functionality and how to use it optimally for themselves.

Very Fast

Redshift databases are very fast. Redshift databases are designed around the idea of grouping processing nodes known as clusters. Clusters are broken into two parts: a single leader node and a group of computer nodes.

The Leader Node is responsible for**:**

- Developing query plans
- Assigning tasks to processing nodes to optimize performance.
- Receiving and compiling all of the data returned by the processing nodes.

Processing nodes are responsible for:

- Performing the queries that are assigned to it by leader node.

There are two types of processing nodes:

- **Dense Store(DS)**- Dense stores nodes are stored on large Hard Disk Drives(HDDs), which are cheaper and have a higher capacity, but are slower than **DC** nodes.
- **Dense Compute(DC)**- Dense compute nodes are stored on Solid State Drives(SSDs) which give them the advantage of being a lot faster, depending on the task and drive types SSDs can be anywhere from 4 to 100 times faster than HDDs, but they are also more expensive and smaller capacity than **DS** nodes.

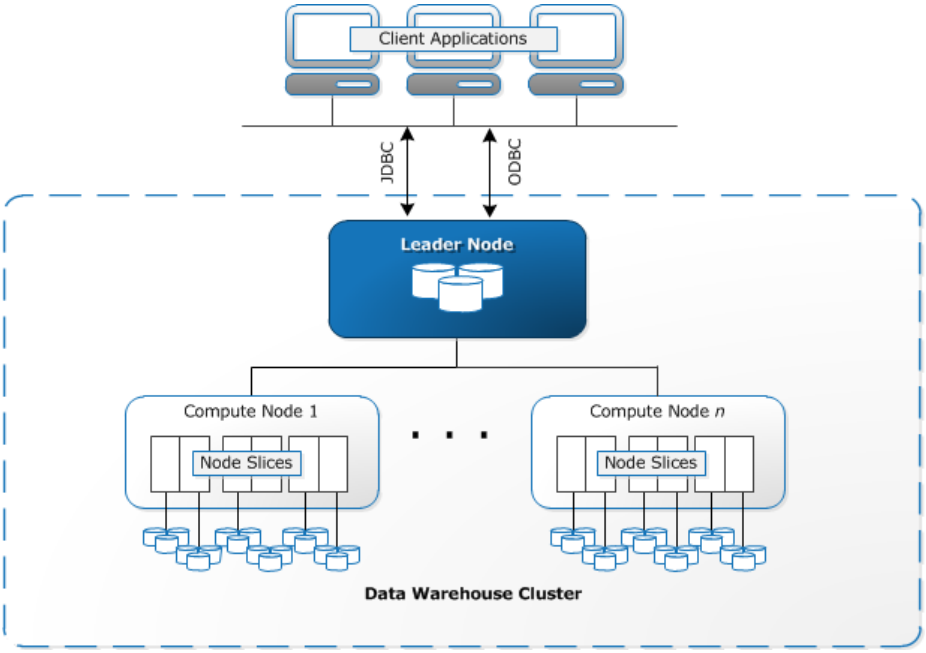
There are several tiers of processing nodes that have varying levels of storage and memory capacities. As databases grow or become more heavily queried, Redshift will upgrade the node tiers to maintain performance levels. More information on node type and pricing can be found here.

Data is stored across processing nodes in smaller subsections of a processing node which are known as **slices**. Slices are assigned a portion of the processing node's memory, the quick access store used to hold data while it is in use, and storage, the long term storage location for the database, to manage queries on.

Data is assigned to the processing node's slices by the leader node and is stored evenly across all of the processing nodes in the database.

Following this structure, Redshift has had to optimize their queries to be run across multiple nodes concurrently. Query plans generated in Redshift are designed to split up the workload between the processing nodes to fully leverage hardware used to store database, greatly reducing processing time when compared to single processed workloads.

Below is an image provided by AWS to explain their cluster interface:



Source:
https://docs.aws.amazon.com/redshift/latest/dg/c_high_level_system_architecture.html

Redshift is one of the fastest databases for data analytics and ad hoc queries. Redshift is built to handle petabyte sized databases while maintaining relatively fast queries of these databases.

Data Compression

Redshift uses a column oriented database, which allows the data to be compressed in ways it could not otherwise be compressed. This allows queries to perform faster because:

- Data takes up less space in memory
- Compression reduces the cost on the CPU
- Can provide optimizations for certain queries that cannot be performed on decompressed data

Data compressions allows for increased performance of the database at the cost of more complex query plans. For the end user, this complication of query plans has no effect and is one of the reasons it is so heavily used.

Highly Scalable

Being stored with the cluster system explained above means that Redshift is highly scalable. As your database grows and expands past what your current configuration can handle, all that needs to be done to reduce query times and add storage is add another processing node to your system. This makes scaling your database over time very easy.

While query execution time is decreased when another node is added, it is not decreased to a set execution time. As processing nodes are added, query plans take longer to form and transferring from many nodes takes greater time.

If your database become more heavily queried over time you may also have to upgrade the node types you are using to store your database. Redshift will do this automatically to maintain a high level of performance.

Query Optimization

As databases grow, the settings used to create the database initially may no longer be the most efficient settings to run your database. To address this, Amazon created the "Schema Conversion Tool" that allows you to easily migrate an existing database into a new database with new, more optimized settings. A guide on migrating your slow database can be found here.

AWS Platform

AWS is an ever expanding cloud platform provided by Amazon. AWS is widely used in the business space to host websites, run virtual servers, and much more. Redshift is a part of Amazon's ecosystem which means it can be easily linked with other Amazon services like: S3, DynamoDB, and Amazon EMR using Multiple Parallel Processing(MPP). MPP is the process of using multiple process nodes to speed up the transfer of data. When all of your services are on AWS you can optimize more than just your data queries by improving transfer times to other databases or buckets on your AWS account.

Another benefit of being on the AWS platform is the security. AWS allows you to grant very specific security clearances to their AWS instances and the same goes for Redshift. You can create a variety of "Security Groups" and "IAM"(Identity and Access Management) settings to lock down your data and keep it safe from outside groups. This optimizes time savings by freeing users from having to maintain third party security settings.

Summary

- Redshift is fully managed by AWS and does not require maintenance by the customer
- Redshift is very fast thanks to their cluster system of dividing work between processing nodes.
    - These nodes are divided further into slices where the data is actually stored and queried
- Redshift is highly scalable because of the cluster system, expanding is as easy as adding another node and redistributing data.
- Redshift benefits from being a member of the AWS family and can be used seamlessly with several other AWS products.
    - AWS also provides a layer of security for Redshift

References:

https://docs.aws.amazon.com/redshift/latest/mgmt/welcome.html
http://db.csail.mit.edu/madden/html/theses/ferreira.pdf
https://hevodata.com/blog/amazon-redshift-pros-and-cons/

BigQuery Optimization

BigQuery is a fully managed cloud database hosted by Google. BigQuery is highly-scalable and elastic, allowing for high speed queries on large amounts of data. It handles optimizing your warehouse for you. Let's explore how it does this.

BigQuery provides several key features:

- Elastic Structure
- Fully Managed
- Data Streaming
- BigQuery Add-on Services
- Flexible Pricing

Elastic Structure

BigQuery is designed with performance and scalability in mind. BigQuery is split into two parts:

- the storage layer
- the compute layer

Storage

The storage layer only handles, you guessed it, storage of data in the database. BigQuery will automatically partition the storage drives that your database requires and automatically organize your data as a column oriented database. BigQuery can host anywhere from a few GigaBytes of data to massive PetaByte databases.

Compute

The compute layer is separate from the storage and is only used to perform queries on the database. This separation of storage and compute power allows databases to scale quickly without large amounts of hardware changes on Google's side.

Fully managed

BigQuery does not require you to choose what hardware your database will use or require you to configure the database's settings. This allows for quick and easy set up of the database. Simply upload or route your database to BigQuery and begin querying your data. BigQuery fully manages the database with no requirements from the user, allowing the user to spend more time on their queries and less time keeping their databases up to speed.

Data Streaming

One big advantage of using BigQuery is high speed data streaming. This high speed data streaming allows users to ingest 100,000 rows of data per second to any data table. This provides a huge benefit for power users who operate based on the live analysis that is coming in to their database.

Services

BigQuery can be linked with several services:

- BigQuery ML- Allows data scientists to create machine learning models using their databases to accomplish tasks like creating product recommendations and predictions.
- BigQuery BI Engine- Create dashboards to analyze complex data and develop insight into business data.
- BigQuery GIS- Plot complex GIS data on a map to better understand the relationship between data and it's real world application.

All of these features increase the productivity of those using BigQuery and add to the value of the service.

Flexible Pricing

BigQuery is priced by data instead of by time used, meaning that BigQuery charges by the GigaByte stored and by the TeraByte queried. This should be taken into account when planning what type of database you want to implement. This payment system works best for databases that do not run queries often because they will not process as much data. The specifics on BigQuery pricing can be found here.

Optimizations

There are no hardware or performance tuning options within BigQuery because BigQuery automatically configures all of that for its users. The only way to optimize your BigQuery database is to write SQL queries that perform most optimally, more on that here from BigQuery. You can read another article we wrote on optimizing SQL queries here. You can also checkout this article on analyzing your SQL queries here.

Summary

- The two part structure of BigQuery allows for easy expansion and quick processing of any database
- Databases are fully managed and maintained by the BigQuery system and require no user input
- BigQuery's data streaming allows for high speed data importing and live updates to data.
- BigQuery's services add even more strength to the system and increase the ability of its users.
- BigQuery's "pay as you use" system is perfect for databases that are not queried often

References

https://cloud.google.com/bigquery/?gclid=EAIaIQobChMIoeqlj_aT4wIVmonICh3t9QbPEAAYASABEgKjxPD_BwE&tab=tab2

Snowflake Optimization

Snowflake is a cloud-based elastic data warehouse or Relational Database Management System (RDBMS). It is a run using Amazon Amazon Simple Storage Service (S3) for storage and is optimized for high speed on data of any size.

The amount of computation you have access to is also completely modifiable meaning that, if you want to run a computationally intense set of queries, you can upgrade the amount of resources you have access to, run the queries, and lower the amount of resources you have access to after. The amount of money that you are charged is dependent on what you use. It has many features to help deliver the best product for a low price.

What is Snowflake and how does it Work?

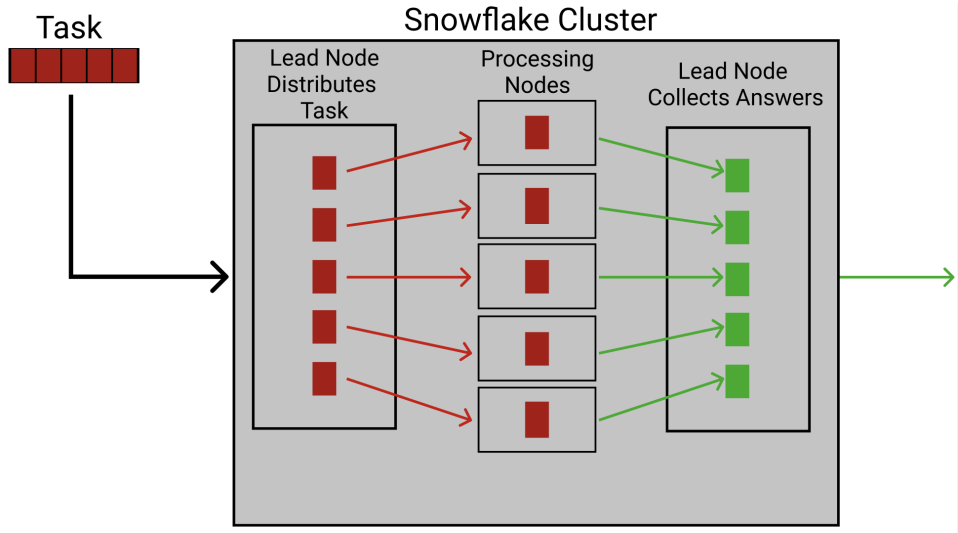
In SQL there are two main database types:

- Databases for **Online Transaction Processing** (OLTP):
  - These are optimized for retrieving a single row or small amount of rows.
  - Uses Row Store
- Databases for **Online Analytical Processing** (OLAP):
  - These are optimized for the performance of analyses like aggregations.
  - Uses Column Store

Snowflake is designed to be an OLAP database system. One of snowflake's signature features is its separation of storage and processing:

- Storage is handled by Amazon S3. The data is stored in Amazon servers that are then accessed and used for analytics by processing nodes.
- Processing nodes are nodes that take in a problem and return the solution. These nodes are grouped into clusters.

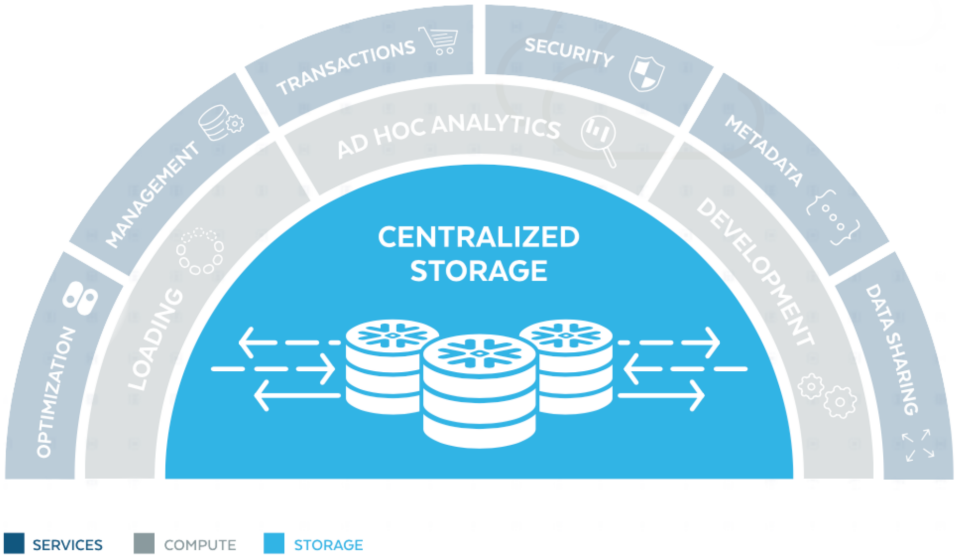
The cluster uses MPP (massively parallel processing) to compute any task that it is given. MPP is where the task is given to the cluster's lead node, which divides the task up into many smaller tasks which are then sent out to processing nodes. The nodes each solve their portion of the task. These portions are then pulled together by the lead node to create the full solution:



Since the data is stored in S3, snowflake will have slightly longer initial query times. This will speed up as the data warehouse is used however due to caching and updated statistics.

Snowflake's Architecture

Snowflake has a specialized architecture that is divided into three layers: storage, compute, and services.



The Storage Layer:

- Elasticity: As mentioned before, storage is separated from the compute later. This allows for independent scaling of storage size and computer resources.
- Fully Managed: Snowflake data warehouses optimizations are fully managed by snowflake. Indexing, Views, and other optimization techniques are all managed by snowflake. The consumer can focus on using the data rather than structuring it.
- Micro-partitions: One optimization that Snowflake uses is micro-partitioning. This means that snowflake will set many small partitions on the data, which are then column stored and encrypted.
- Pruning: Micro-partitions increase efficiency through pruning. Pruning is done by storing data on each micro-partition and then, when a certain massive table needs to be searched, entire blocks of rows can be ignored based on the micro-partitions. This means that only pages which contain results will be read.

The Compute Layer:

- Caching: Snowflake will temporarily cache results from the storage layer in the compute layer. This will allow similar queries to run much faster once the database has been “warmed up.”
- ACID Compliance: Snowflake is ACID compliant. This means that it follows a set of standards to ensure that their databases have:
  - **Atomicity:** If a part of a transaction fails, the whole transaction fails.
  - **Consistency:** Data can not be written to the database if it breaks the databases’ rules.
  - **Isolation:** Multiple Transaction blocks can not interfere with each other and be run concurrently.
  - **Durability:** Ensures that data from completed transactions will not be lost in transmission.

The Services Layer:

- DML and DDL: There are 4 sub-languages in SQL:
  - **Data Definition Language** (DDL): DDL is the sub-language that handles defining and describing structures in sql.
    - Common Examples of DDL Commands: **CREATE**, **RENAME**, **ALTER**.
  - **Data Manipulation Language** (DML): DML handles manipulating data in structures defined by DDL.
    - Common Examples of DML Commands: **SELECT**, **INSERT**, **DELETE**.
  - **Data Control Language** (DCL): DCL handles controlling access and privileges.
    - Common Examples of DCL Commands: **GRANT**, **REVOKE**.
  - **Transaction Control Language** (TCL): TCL handles control of transaction blocks.
    - Common Examples of TCL Commands: **START**, **ROLLBACK**.

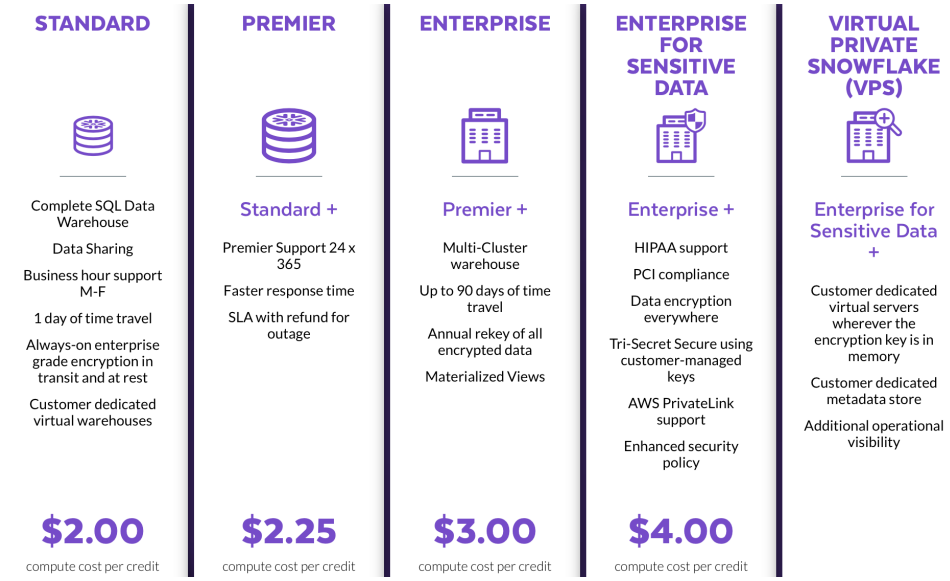
Of these 4 sub-languages, Snowflake adds DDL and DML functions at the service layer. DCL and TCL are mostly handled by through the Snowflake GUI or automatically.

- Meta-data: Snowflake controls how meta-data is stored through the services layer, allowing it to be used to create micro-partitions and further optimizing the structure of the database.
- Security: Security is handled at the security layer. Snowflake does many things to increase security including: multi-factor authentication, encryption (at rest or in transit).

How is it Priced?

Snowflake has a variety of factors that impact the price of using their data warehouse. The first is what type of data you store.

Warehouse Type



These different tiers decide what level of security, speed, and support are needed. They have corresponding costs as well. The basic package costs $2.00 per credit used. Credits are how snowflake measures the computations done.

Warehouse Size

While many data warehouse companies have a cost based on the maximum processing power that your database might need, snowflake is priced based on the amount of processing that is used. When you turn on your database, you can set what amount of processing power you want. The size of the warehouse can range from X-Small to 3X-Large. These sizes directly relate to how much computational power you can access and correspondingly how expensive the warehouse is. Here is a table from the snowflake documentation regarding the costs of three different sizes:

Running Time	Credits (X-Small)	Credits (X-Large)	Credits (3X-Large)
0-60 seconds	0.017	0.267	1.067
61 seconds	0.017	0.271	1.084
2 minutes	0.033	0.533	2.133
10 minutes	0.167	2.667	10.667
1 hour	1.000	16.000	64.000

(Full list of sizes: X-Small, Small, Medium, Large, X-Large, 2X-Large, 3X-Large)

As the table shows the cost (in credits) is based on the size of the warehouse and the amount of time it is run for. For example, if you run a medium data warehouse, but need to run some intense queries for an hour, you can upgrade to a X-Large box for an hour. You can set this upgrade to automatically revert to medium after an hour as well to ensure you only pay for what you need.

Another Important detail about snowflake that we can determine from the table above is that snowflake has near linear scalability. An X-small warehouse costs 1 credit/hour (Note: this is per cluster. Multi-cluster warehouses will be charged based on the number of clusters). This price doubles for each tier that the warehouse goes up, as does the computational power.

Snowflake is designed for ease-of-use and easy hands-off optimization. Perhaps the best feature of snowflake is how easy it is to use. It is very simple once acclimated to, and can save you or your staff a lot of time and hardship handling technical details that even other RDBMS will not. As such it is a great tool for optimizing your database and optimizing your time.

Summary

- Snowflake is a RDBMS designed for OLAP usage
- Uses Amazon S3 for storage
    - Separated storage leads to increased initial query times, but much better elasticity.
    - Has near linear scaling in compute time (E.g. 2x the compute power = ½ the time)
- Has Architecture built for high speed under high load
- Extremely flexible pricing.
    - Pricing by second used.
    - Scheduled downgrading
    - Tiers of data storage
    - Linear pricing (2x the compute power = 2x the price)

Resources:

https://www.snowflake.com/

https://aptitive.com/blog/what-is-snowflake-and-how-is-it-different/

https://youtu.be/XpaN-PqSczM

https://www.lifewire.com/the-acid-model-1019731

https://www.w3schools.in/dbms/database-languages/

https://www.youtube.com/embed/Qzge2Mt84rs?autoplay=1