Week 6: Star Schemas and Query Optimization

Advanced SQL Concepts

Contents

Row vs. Column Storage in Databases

Relational databases typically store information in one of two ways:

- **row-oriented** (where one row of information is stored together on disk)
- **column-oriented** (where one column of information is stored together on disk)

Take for example a `businesses` table with two rows:

Businesses Table

business_id	name	reviews
92kues81oa9	Easttown Bar and Grill	287
9ak28js7jh0k	Uncle Micky's Hoagies	190

Row Oriented Storage

Columnar Storage

Why does it matter how data is stored on disk? Because loading data from disk is one of the most costly operations a program can perform. In order to perform computation, computers load data from disk into memory. These are rough numbers and approximations, but fetching data from a system's memory is roughly **200ns** (200 nanoseconds). Fetching data from disk is roughly **1,000,000 ns**, roughly about 5,000 times slower (http://norvig.com/21-days.html#answers).

You can reference the following visualization for a visual representation of the differences in scale between these types of operations:

Database Access Patterns

When you are setting up your data pipelines, and deciding between PostgreSQL, MySQL, Redshift, BigQuery, MongoDB, etc., the first question to ask is *how will the data be accessed?*

E-Commerce Transactions

> *The database is for a large e-commerce vendor that focuses on footwear and shoes. Customers place orders on a website and these orders are stored in a database. Customers often want to look up their orders to check its current status. There are significant seasonality effects and certain times of the year become hot spots where lots of customers are logging in to buy discounted shoes.*

Now map this business description into the types of SQL queries and statements you are most likely to see. Whenever a customer makes a new purchase, we'll see something like this in the database:

```
INSERT INTO orders
VALUES (
  91288492, -- order ID (primary key)
  218294, -- customer ID
  9283100221, -- product ID
  4, -- quantity
  98.99, -- total USD paid for order
  0, -- discount
```

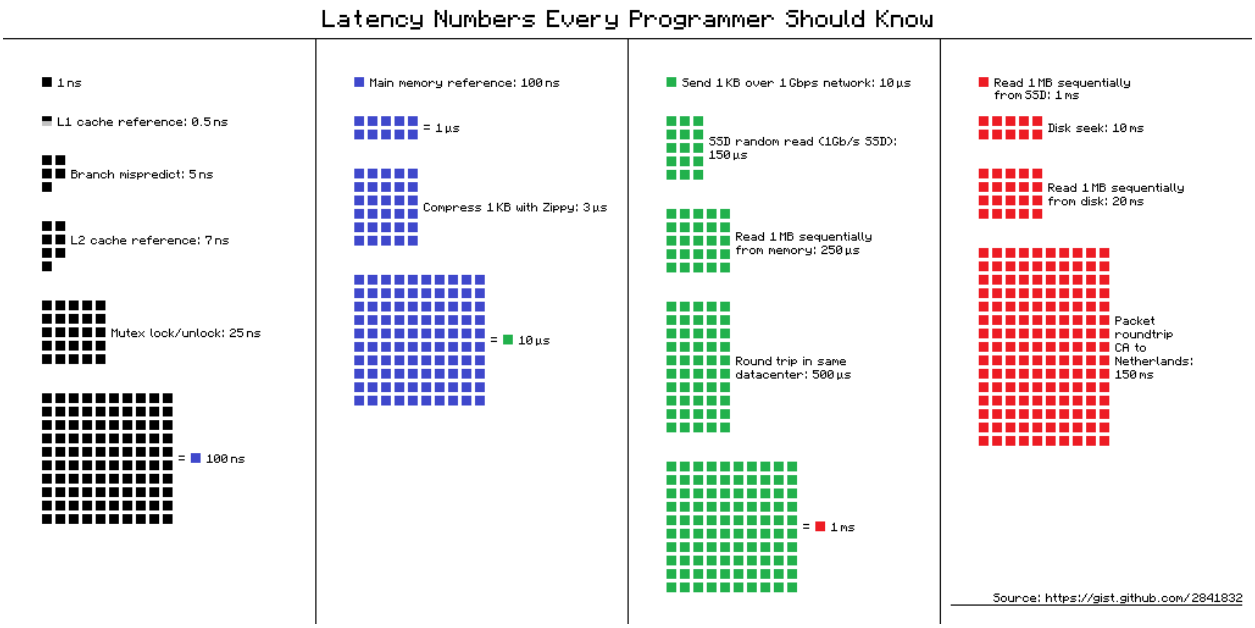


Figure 1: Latencies

```
  '2020-11-03 12:32:02', -- date of order
  'PROCESSING' -- order status
)
```

Whenever a customer loads an order on the website to check the status of it, the backend database will need to run the following query:

```
SELECT *
FROM orders
WHERE order_id = :order_id
```

This query looks like a **transactional** query, ie. one that is used in day-to-day operations to keep the business running.

Notice that in both circumstances, we are working with individual records (ie. one order at a time), but we often need to know multiple pieces of information about that particular record - for instance, the `customer_id`, the `product_id`, the `order_status`, etc.

Let's take a look at how this looks via row-level storage:

Overall, the entire process required 1 fetch from disk. Let's look now at what happens in this workflow when we are using a **column-based** storage system:

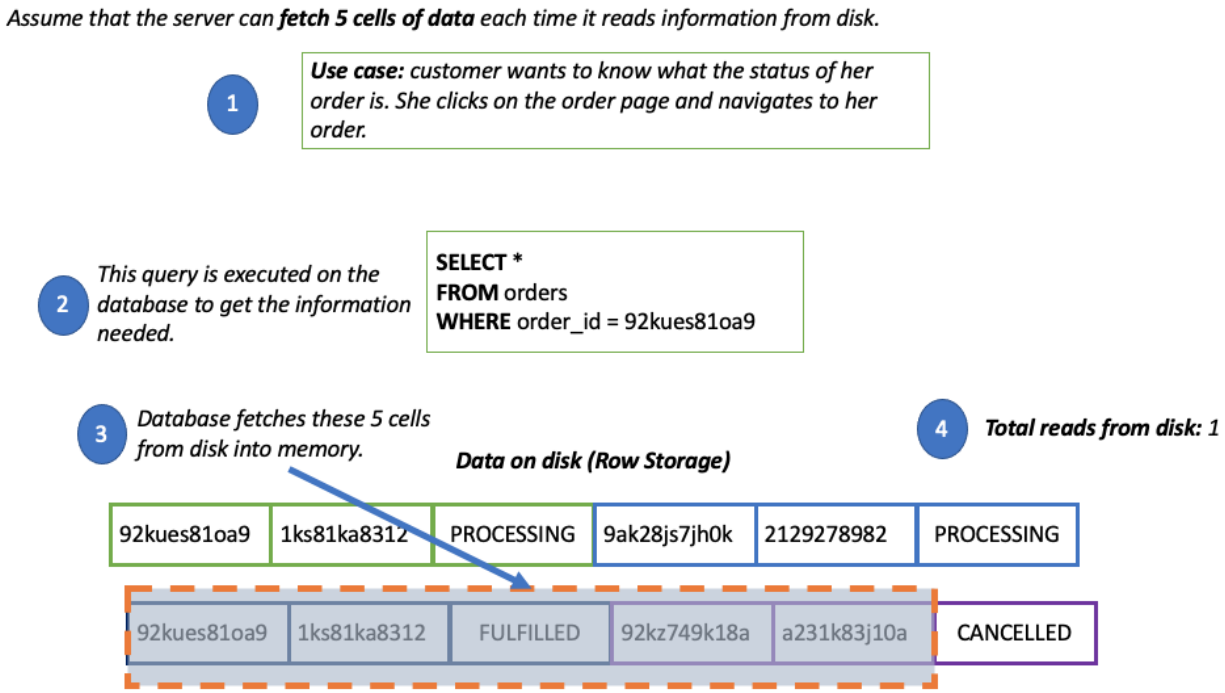
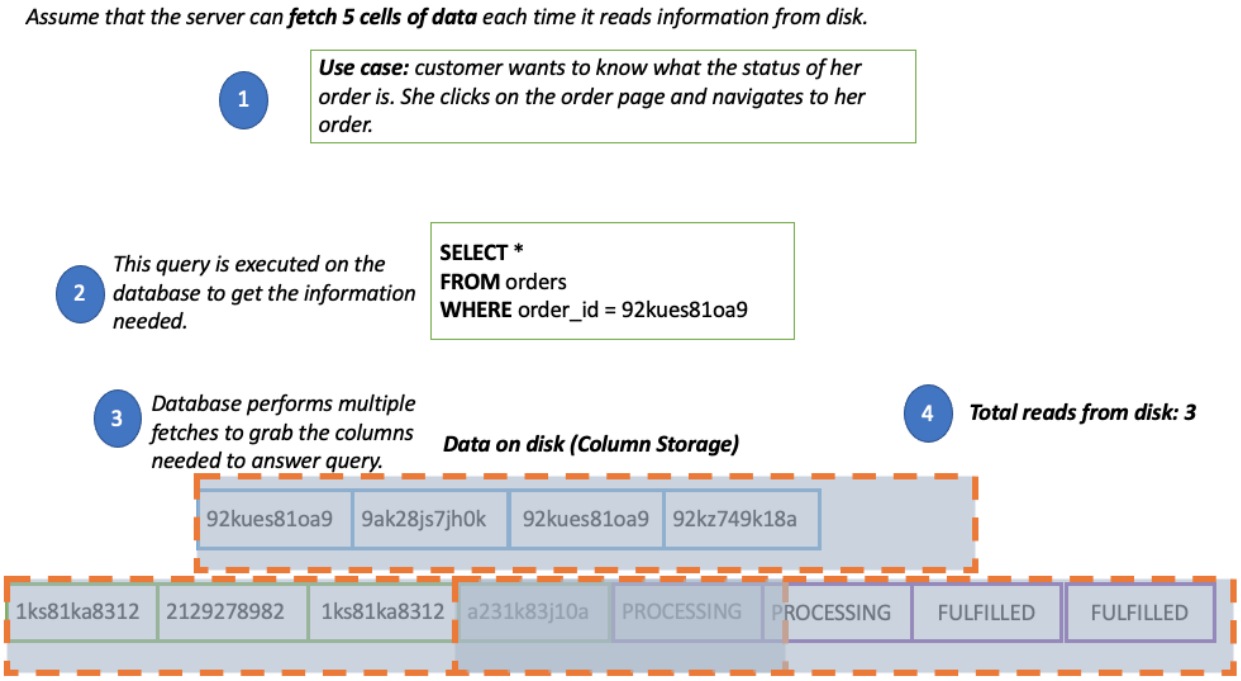


Figure 2: Fetching Order Information Workflow with Row Level Transactions



Notice how it takes **3** reads from disk to get all of the data, since the information we are looking for is spread across the entire disk and is not centralized in one place. This is far more inefficient, and will result in significant delays when querying for information.

So what is **columnar-based storage** good for?

Analytics Reporting Use Case

Imagine this alternative use case:

> *Our fulfillment team needs an analytics dashboard that updates in real-time letting them know what percentage of orders are in which status `FULFILLED`, `DELAYED`, `CANCELLED`, etc. for the last 30 days.*

We should know enough this point to be able to translate this into a SQL query. First, we know that the result set data must update in real time, which should be an indicator that we want to use a view. Second, the only column we really care about is the `order_status` column (in actuality, we also need to know the `order_date`, but put that aside for now). To answer this particular question, we don't need to know the customer IDs, or product IDs.

So our view would look something like this:

```
CREATE OR REPLACE VIEW rolling_30_day_order_status_view AS
SELECT status, COUNT(*) AS num_orders
FROM orders
WHERE order_date >= 'now'::TIMESTAMP - '1 month'::interval
GROUP BY 1
ORDER BY 2
```

This looks a lot more like an **analytics** query - something that an analytics or data science team member would be concerned about.

Now the access patterns are flipped. We just want one field - `status` (remember, ignore the fact that we have to filter for `order_date` first).

What does this look like in a database using columnar storage?

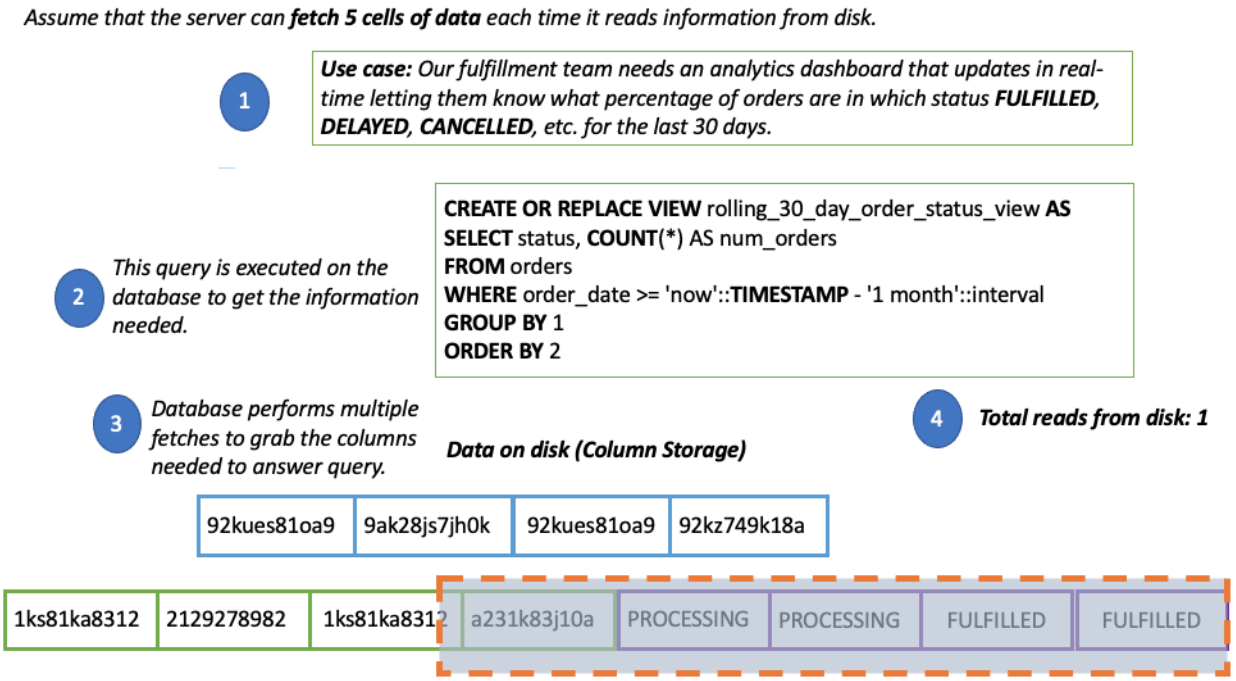


Figure 3: Fetching the Order Status Column for an Aggregation Query Using Columnar Storage

Advantages of **row-oriented** database storage:

- easy to write data (`INSERT`, `UPDATE`, `DELETE`)
- easy to get single records
- efficient for **transactional** access patterns (such as the query to fetch the order information for a single `order_id`) - **Online Transactional Processing**

Advantages of **column-oriented** database storage:

- significantly more efficient for aggregation queries
- efficient for **analytics** access patterns - **Online Analytics Processing (OLAP)**.

PostgreSQL and MySQL use **row-oriented** storage. Google BigQuery, Snowflake, Redshift, Apache Cassandra, and Microsoft Azure SQL Data Warehouse use **columnar** storage.

Indices

Let's come back to this query:

```
SELECT status, COUNT(*) AS num_orders
FROM orders
WHERE order_date >= 'now'::TIMESTAMP - '1 month'::interval
GROUP BY 1
ORDER BY 2
```

Notice that we have to filter on `order_date`. So how will this query actually work? Under the hood, we are going to have to sequentially scan each row and check its `order_date` value to see if it is within the past 30 days. This is quite inefficient. If we have 1,000,000 rows in our database, that means we'll need to scan through all 1 million rows.

We need a better way of finding the rows we want, and this is where **indices** come in.

Query Plans

Let's go to our `yelp` database and run the following SQL command (notice the `EXPLAIN` before the query!):

```
EXPLAIN
SELECT *
FROM reviews
WHERE business_id = 'p0iEUamJVp_QpaheE-Nz_g'
```

The result we get is a bit different than a traditional result set:

QUERY PLAN
Bitmap Heap Scan on reviews (cost=64.14..7813.57 rows=2046 width=671)
Recheck Cond: (business_id = 'p0iEUamJVp_QpaheE-Nz_g'::text)
-> Bitmap Index Scan on business_id_fkey (cost=0.00..63.62 rows=2046 width=0)
Index Cond: (business_id = 'p0iEUamJVp_QpaheE-Nz_g'::text)

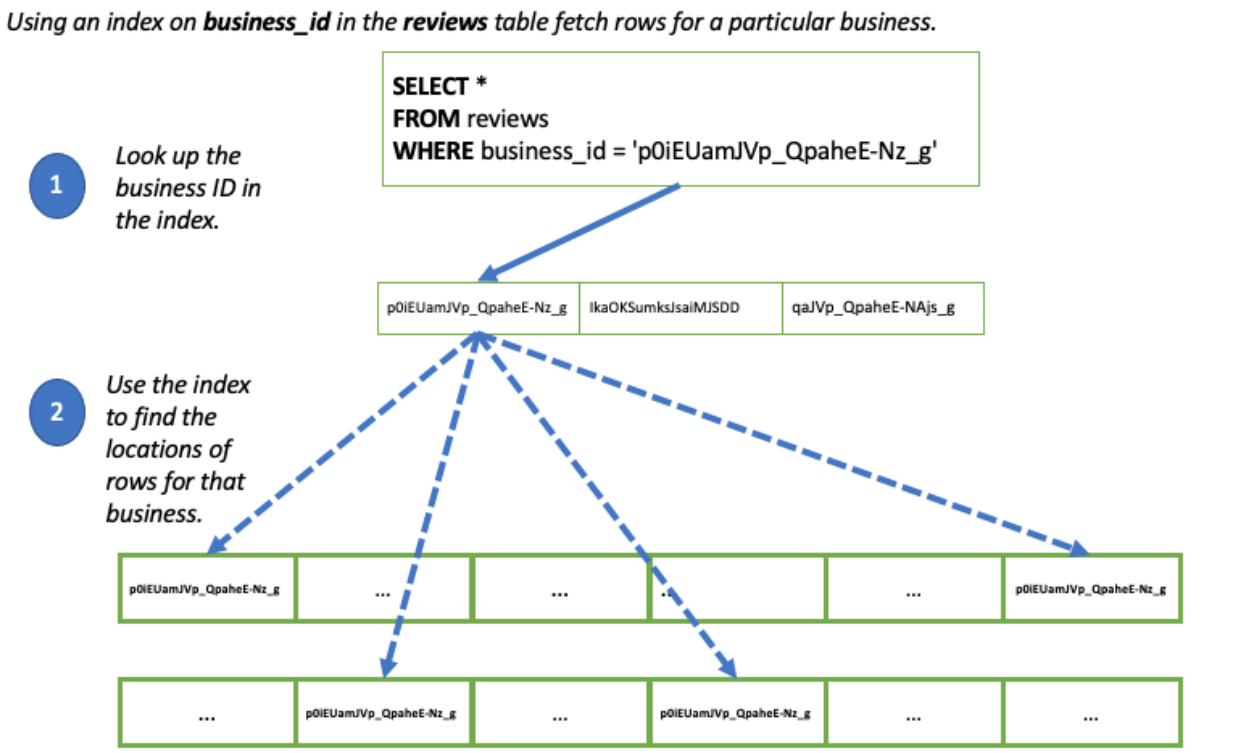
This result bears some explaining:

- This is an example of a **query plan**, which is the set of instructions that PostgreSQL is sending itself to perform your query.
- You read query plans from the bottom up (so in this case, `Index Cond` is the first row to read).
- An `Index Scan` means that PostgreSQL is looking up locations for the rows
- This query is going to perform the following operations:

1. Look up the index for `business_id` equal to `p0iEUamJVp_QpaheE-Nz_g` (Bitmap Index Scan)
2. Look up the rows that have this `business_id` based on the locations returned from the index in step 1 (Bitmap Heap Scan)

3. Fetch each column from the `reviews` table for those rows.

- The **cost** is a somewhat arbitrary quantifier for how "expensive" each operation is. However, you can see that the total cost of this query is `7813.57`. Of that cost, a very small fraction (`64/7813`) of it was spent looking up the rows in the index. The rest of the cost (`7749 / 7813`) was spent fetching the 2046 relevant rows of data from disk.



We can see all the indices that have created by using the following command:

```
SELECT *
FROM pg_indexes
WHERE tablename NOT LIKE 'pg%';
```

schemaname	tablename	indexname	tablespace	indexdef
public	reviews	reviews_pkey	NA	CREATE UNIQUE INDEX reviews_pkey ON public.reviews USING btree (review_id)
public	businesses	businesses_pkey	NA	CREATE UNIQUE INDEX businesses_pkey ON public.businesses USING btree (business_id)
public	users	users_pkey	NA	CREATE UNIQUE INDEX users_pkey ON public.users USING btree (user_id)
public	reviews	business_id_fkey	NA	CREATE INDEX business_id_fkey ON public.reviews USING btree (business_id)
ychen220	businesses	state_index	NA	CREATE INDEX state_index ON ychen220.businesses USING btree (state)
ychen220	tips	tips_business_id_index	NA	CREATE INDEX tips_business_id_index ON ychen220.tips USING btree (business_id)
ychen220	tips	tips_compliment_count_index	NA	CREATE INDEX tips_compliment_count_index ON ychen220.tips USING btree (compliment_count)

Note that in the reviews table, there's two indices - `reviews__pkey` and `business_id_fkey`.

So our query used the `business_id_fkey` to look up the location of the business IDs efficiently, without having to scan the entire `reviews` table.

Let's look now at the difference between a query that does not have an index and one that does:

```
SELECT *
FROM businesses
WHERE state = 'AZ'
```

We can create another copy of `businesses` in our personal schema:

```
CREATE TABLE ychen220.businesses
AS
SELECT *
FROM businesses
```

We'll create an index using the following command:

```
CREATE INDEX state_index ON ychen220.businesses USING btree(state);
```

However, if we actually run `EXPLAIN` to look at the query plan between the `public.businesses` (which does not have an index on `state`) and our personal one (which now does have an index on `state`) - you can see that we still get the same exact query plan, but different costs:

```
EXPLAIN
SELECT *
FROM public.businesses WHERE state = 'AZ'
```

QUERY PLAN
Seq Scan on businesses (cost=0.00..9653.62 rows=17 width=612)
Filter: (state = 'AZ'::text)

```
EXPLAIN
SELECT *
FROM ychen220.businesses WHERE state = 'AZ'
```

QUERY PLAN
Seq Scan on businesses (cost=0.00..5.62 rows=1 width=328)
Filter: (state = 'AZ'::text)

A **sequential scan** of a table is done by iteratively (think a for loop scanning all item pointers of all pages in the table.

Although we still use a sequential scan, our scan is significantly faster and costly. We scan only one row and return 15 rows , instead of scanning 17 rows. We only have to scan 1 item pointer (the pointer to where `state` is `AZ` before our query can return results.)

1. **YOUR TURN** Copy over the `tips` table into your own personal schema. Then, run the following query and see what type query plan is generated:

```
EXPLAIN
SELECT *
FROM ychen220.tips
WHERE business_id = '0bF6jv97Z6VbruRpepUOcw';
```

You'll likely find a sequential scan occurring. Now, add an index to this table so that we are able to reduce the cost of the query.

Your final result should look similar to this - you'll know that your query is properly using your index if the query plan lists an `Index Scan`:

QUERY PLAN
Bitmap Heap Scan on tips (cost=4.62..23.19 rows=45 width=119)
Recheck Cond: (business_id = '0bF6jv97Z6VbruRpepUOcw'::text)
-> Bitmap Index Scan on tips_business_id_index (cost=0.00..4.61 rows=45 width=0)
Index Cond: (business_id = '0bF6jv97Z6VbruRpepUOcw'::text)

In your answer, write the actual `CREATE INDEX` command that you used.

2. **YOUR TURN** In our company, we have to often filter for tips that have high compliment counts. That means as an analytics team, we tend to write queries that resemble the below:

```
SELECT *
FROM tips
WHERE compliment_count  > X -- x is some number
```

Add an additional index to your personal `tips` table so that queries that use the `compliments` field as part of its `WHERE` clause are optimized for performance.

QUERY PLAN
Index Scan using tips_compliment_count_index on tips (cost=0.28..9.21 rows=8 width=119)
Index Cond: (compliment_count > 0)

In your answer, write the actual `CREATE INDEX` command that you used.

Star Schemas

Now that we've learned a brief bit about how to optimize our queries, the next task is to begin thinking about how we can design scalable, highly performant databases that can meet our analytics needs as the volume of data grows into Big Data territory. What if we have millions and millions of orders?

How can we design our tables in such a way that we can easily meet any analytics query need in the future?

A core pattern to be familiar with in terms of Data Warehouse architectures is a **star schema**:

Imagine that we are running a customer service call center. We have the following entities:

- `customer_call` - a specific call received about a product
- `customer` - the caller making the call
- `staff` - the staff member answering the call
- `time` - the time of day the call occurred
- `product` - the product the call was in regards to

A **star schema** design for this enterprise would look like this (http://www.rapid-business-intelligence-success.com/star-schema.html):

There's two primary types of tables here:

- A **fact table**, which contains the most granular-level event that we are keeping track of in our database (in this case, a `customer_call`)
- Multiple **dimension tables**, which contain different attributes about the event (such as the product that the call was about, the time the call occurred, the customer information of the caller, etc.)

The **fact table** contains foreign key references to the dimension tables. For example, the `Customer ID` field in the `Customer Call` table is a foreign key that references the `Customer ID` primary key of the `Customer` table.

The dimension tables will contain any relevant indices based on the access patterns that your analytics reporting requires. For example, if we commonly report based on the month of the call, there may likely be a `Month` index on the `Time` table.

Advantages of Star Schema

- **Easier Joins**: Join logic is usually just a simple `fact_table.customer_id = customer.customer_id`. Compared to a highly normalized database, where you may have 3-4 levels of joins to get the information you want, you often have only one-level joins in star schemas.

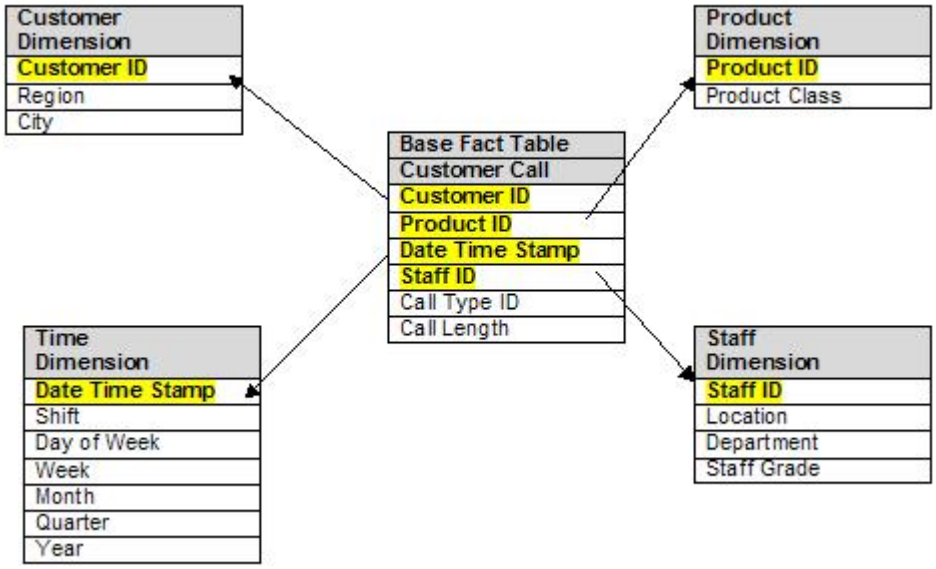


Figure 4: Star Schema Architecture

- **Takes Advantage of Query Optimization**: Using proper indexing, you can achieve extremely high performance on star schema-designed databases - often up to billions of rows of data.

Disadvantages of Star Schemas

- **Less Flexibility**: You design a star schema around a central transaction, or event. If your business model or reporting doesn't fit this definition, taking advantage of a star schema can be difficult.
- **Difficult to Model Complex Relationships**: It is not easy or intuitive how to represent many to many relationships using a star schema. For example, what if multiple callers and multiple support staff can be on a call?
- **Denormalized Data**: Star Schemas are not as normalized as some of the tables that we have seen before. As a result, the same issues that come with denormalized data (making updates to fields) apply to the dimension tables.

3. **YOUR TURN** Copy over the `bookings` and `members` tables in the `club` database to your personal schema. Add a foreign key on the `bookings` table that references the primary key of the `members` table.

Then, please add an index to the `members` table based on the following business requirement:

> We often want to filter and sort members based on when they joined the club, since we have promotions based on their join date.

Optimize the following query:

```
SELECT *
FROM bookings b
JOIN members m ON m.memid = b.memid
WHERE joindate > '2012-08-01';
```

```
EXPLAIN
SELECT *
FROM bookings b
JOIN members m ON m.memid = b.memid
```

```
WHERE joindate > '2012-08-01';
```

Your final result should look similar to this - you'll know that your query is properly using your index if the query plan lists an `Index Scan` (for example, this below shows an unoptimized query plan that is still using sequential scans):

QUERY PLAN
Hash Join (cost=1.65..84.15 rows=2739 width=99)
Hash Cond: (b.memid = m.memid)
-> Seq Scan on bookings b (cost=0.00..70.44 rows=4044 width=24)
-> Hash (cost=1.39..1.39 rows=21 width=75)
-> Seq Scan on members m (cost=0.00..1.39 rows=21 width=75)
Filter: (joindate > '2012-08-01 00:00:00'::timestamp without time zone)

Snowflake Schemas

Since one of the primary concerns and disadvantages of star schemas is that they are denormalized, a derivative of the star schema architecture, called the **snowflake schema**, was created (https://www.guru99.com/star-snowflake-data-warehousing.html):

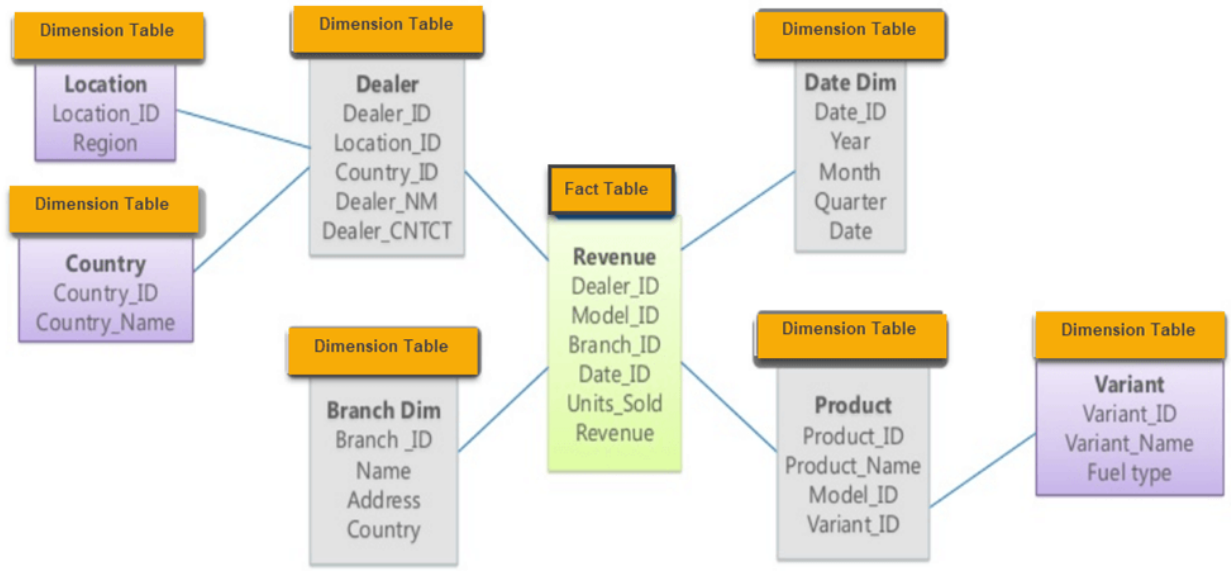


Figure 5: Snowflake Architecture

In this case, each of the dimension tables is further normalized so that there is as few repeated pieces of information as possible.