Week 6: Star Schemas and Query Optimization (with Solutions)

Advanced SQL Concepts

Contents

Row vs. Column Storage in Databases **1**
Database Access Patterns . . . 2
E-Commerce Transactions . . . 2
Analytics Reporting Use Case . . . 5

Indices **6**
Query Plans . . . 6

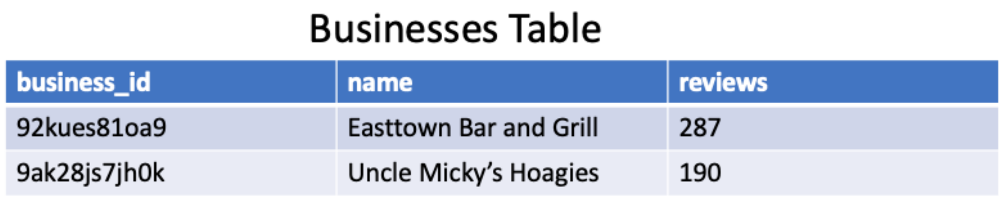
Star Schemas **10**
Advantages of Star Schema . . . 11
Disadvantages of Star Schemas . . . 11
Snowflake Schemas . . . 15

Row vs. Column Storage in Databases

Relational databases typically store information in one of two ways:

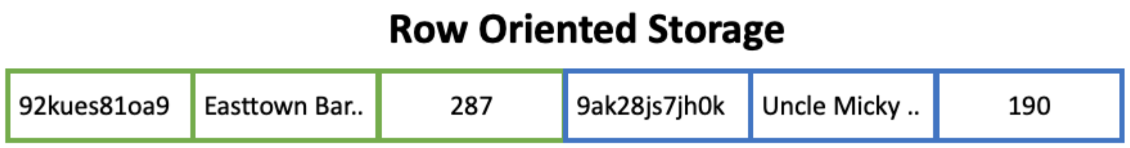
- **row-oriented** (where one row of information is stored together on disk)
- **column-oriented** (where one column of information is stored together on disk)

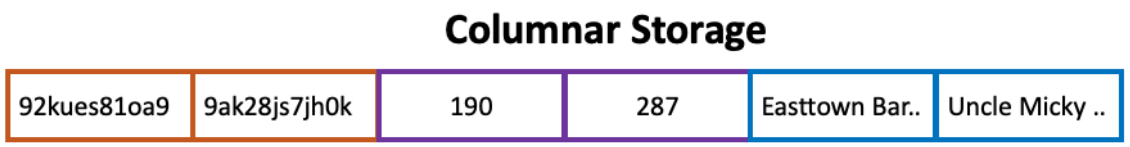
Take for example a `businesses` table with two rows:



Businesses Table

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





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:

```sql
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
```

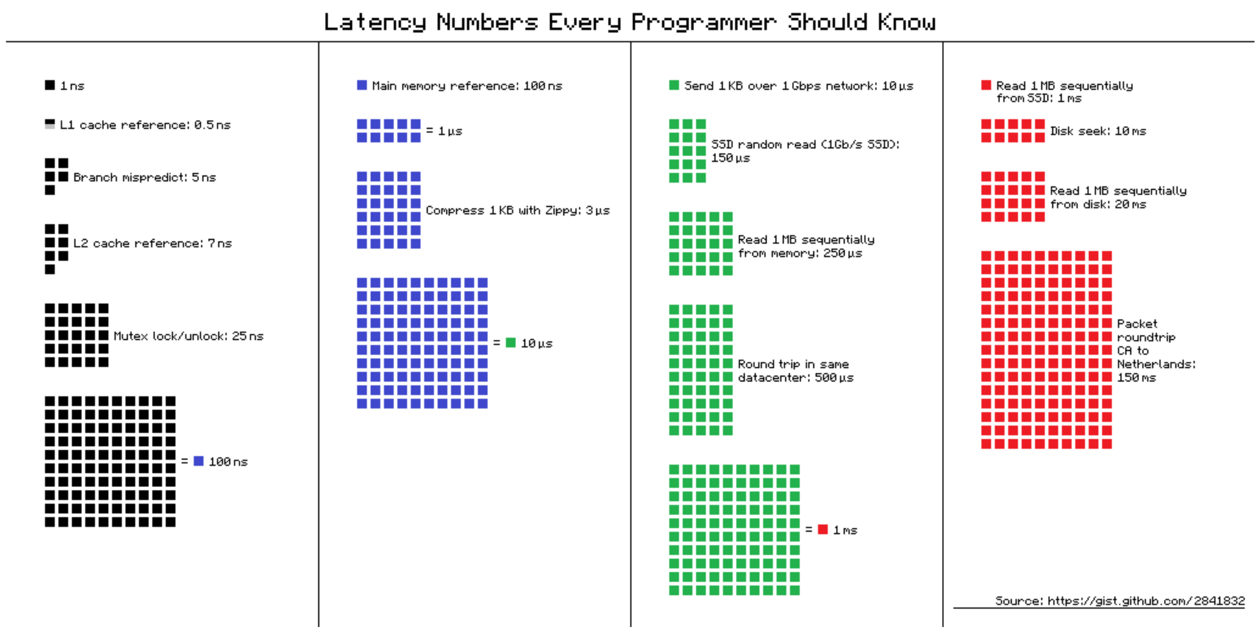


Figure 1: Latencies

```
  '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:

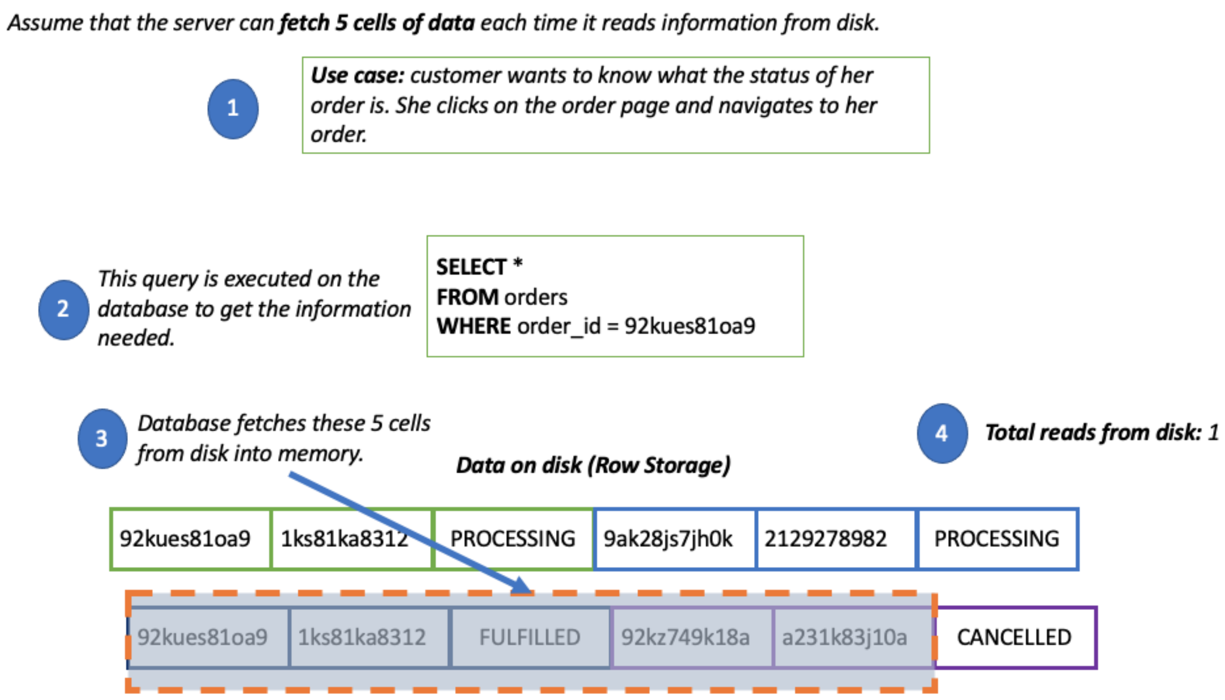
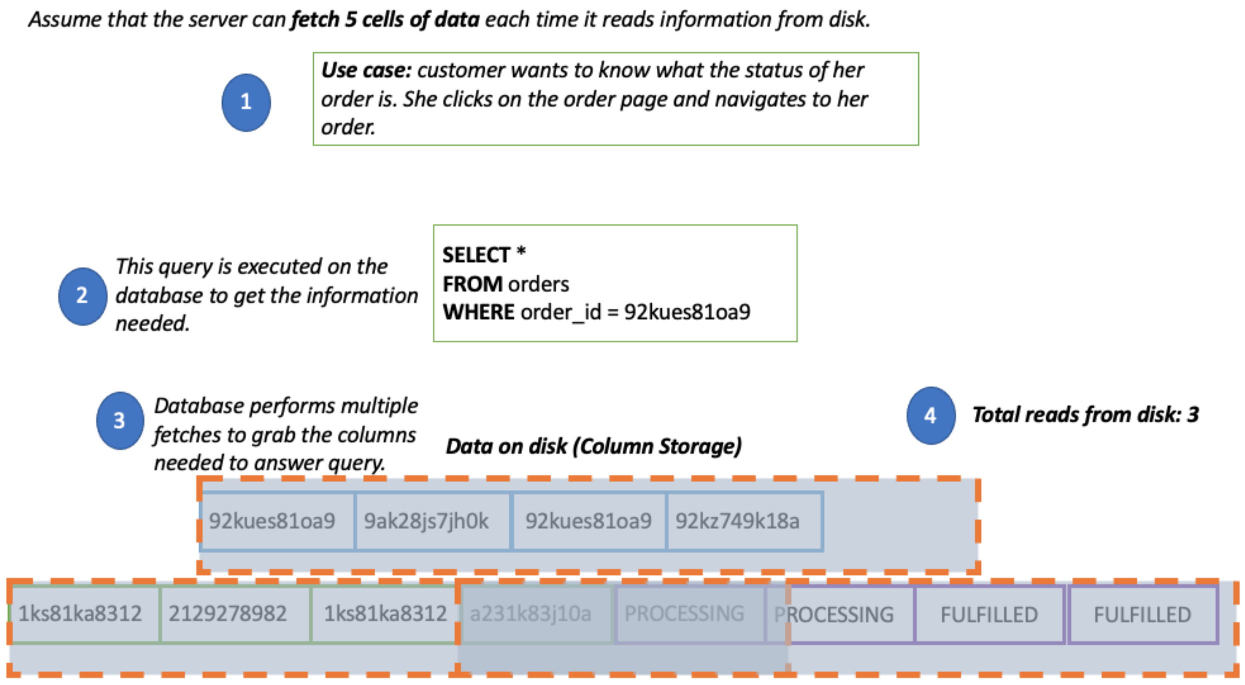

Assume that the server can fetch 5 cells of data each time it reads information from disk.
1
Use case: customer wants to know what the status of her order is. She clicks on the order page and navigates to her order.
2
This query is executed on the database to get the information needed.
SELECT *
FROM orders
WHERE order_id = 92kues81oa9
3
Database fetches these 5 cells from disk into memory.
Data on disk (Row Storage)
4
Total reads from disk: 1
92kues81oa9
1ks81ka8312
PROCESSING
9ak28js7jh0k
2129278982
PROCESSING
92kues81oa9
1ks81ka8312
FULFILLED
92kz749k18a
a231k83j10a
CANCELLED

Figure 2: Fetching Order Information Workflow with Row Level Transactions


Assume that the server can fetch 5 cells of data each time it reads information from disk.
1
Use case: customer wants to know what the status of her order is. She clicks on the order page and navigates to her order.
2
This query is executed on the database to get the information needed.
SELECT *
FROM orders
WHERE order_id = 92kues81oa9
3
Database performs multiple fetches to grab the columns needed to answer query.
Data on disk (Column Storage)
4
Total reads from disk: 3
92kues81oa9
9ak28js7jh0k
92kues81oa9
92kz749k18a
1ks81ka8312
2129278982
1ks81ka8312
a231k83j10a
PROCESSING
PROCESSING
FULFILLED
FULFILLED

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.

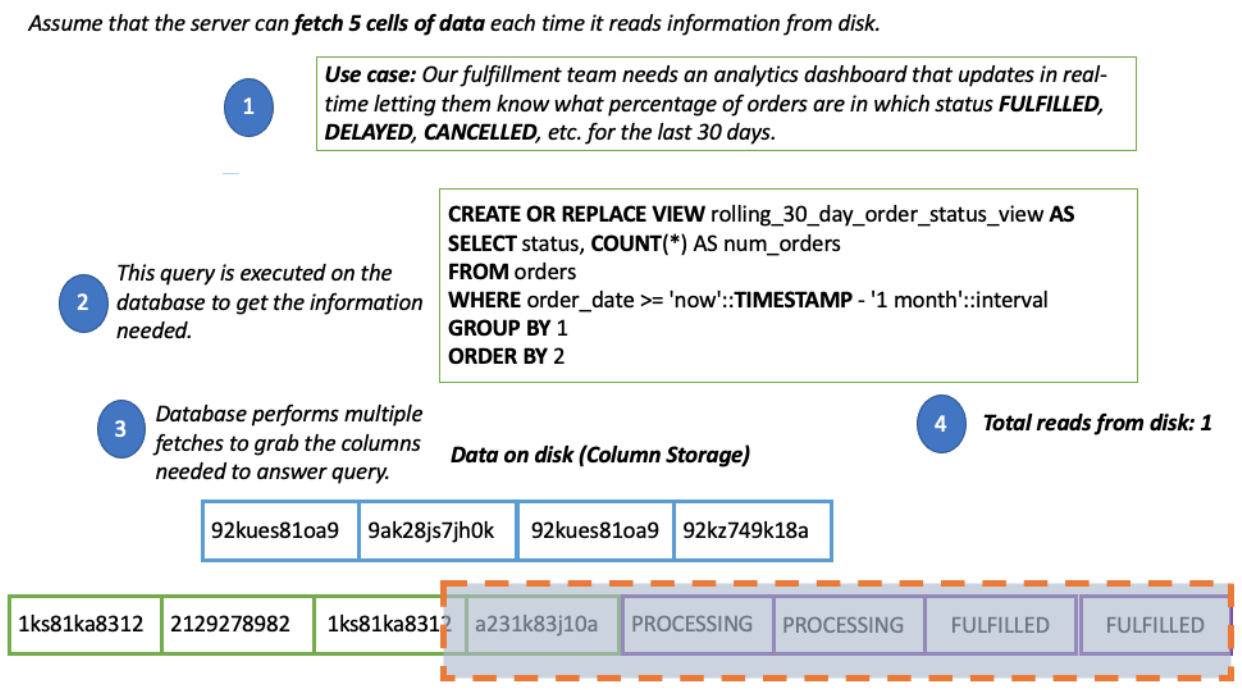
So our view would look something like this:

```sql
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?


*Assume that the server can **fetch 5 cells of data** each time it reads information from disk.*
1
***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.*
2
This query is executed on the database to get the information needed.
3
Database performs multiple fetches to grab the columns needed to answer query.
Data on disk (Column Storage)
4
Total reads from disk: 1
92kues81oa9
9ak28js7jh0k
92kues81oa9
92kz749k18a
1ks81ka8312
2129278982
1ks81ka8312
a231k83j10a
PROCESSING
PROCESSING
FULFILLED
FULFILLED

```
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
```

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:

```sql
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!):

```sql
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)

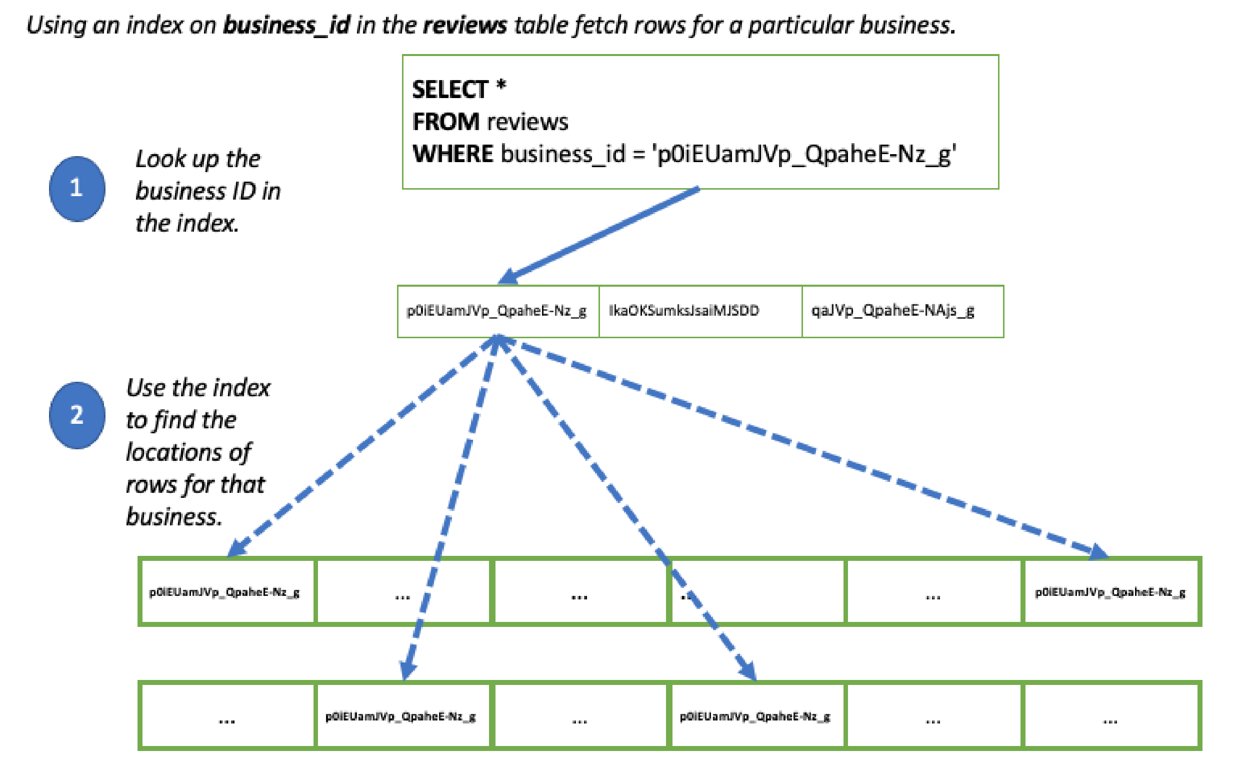
This result bears some explaining:

- 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.


Using an index on business_id in the reviews table fetch rows for a particular business.
SELECT *
FROM reviews
WHERE business_id = 'p0iEUamJVp_QpaheE-Nz_g'
1
Look up the business ID in the index.
p0iEUamJVp_QpaheE-Nz_g
lkaOKSumksJsaiMJSDD
qaJVp_QpaheE-NAjs_g
2
Use the index to find the locations of rows for that business.
p0iEUamJVp_QpaheE-Nz_g
...
p0iEUamJVp_QpaheE-Nz_g
...
p0iEUamJVp_QpaheE-Nz_g
p0iEUamJVp_QpaheE-Nz_g

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

```
SELECT *
FROM pg_indexes
WHERE tablename NOT LIKE 'pg%' LIMIT 10;
```

schemaname	tablename	indexname	tablespace	indexdef
public	businesses	businesses__pkey	NA	CREATE UNIQUE INDEX businesses__pkey ON public.businesses USING btree (business__id)
chenlizi	tips	tips__compliment__count__index	NA	CREATE INDEX tips__compliment__count__index ON chenlizi.tips USING btree (compliment__count)
chenlizi	tips	tips__business__id__index	NA	CREATE INDEX tips__business__id__index ON chenlizi.tips USING btree (business__id)
public	users	users__pkey	NA	CREATE UNIQUE INDEX users__pkey ON public.users USING btree (user__id)
kerenli	businesses	state__index	NA	CREATE INDEX state__index ON kerenli.businesses USING btree (state)
kentzhan	businesses	state__index	NA	CREATE INDEX state__index ON kentzhan.businesses USING btree (state)
ruyuwang	tips	tips__compliment__count__index	NA	CREATE INDEX tips__compliment__count__index ON ruyuwang.tips USING btree (compliment__count)
ruyuwang	tips	tops__business__id__index	NA	CREATE INDEX tops__business__id__index ON ruyuwang.tips USING btree (business__id)
gaoyi	tips	compliment__count__index	NA	CREATE INDEX compliment__count__index ON gaoyi.tips USING btree (compliment__count)
gaoyi	tips	business__id__index	NA	CREATE INDEX business__id__index ON gaoyi.tips USING btree (business__id)

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 business_id, name, address, city, city, state, postal_code, stars, review_count
FROM businesses
WHERE state = 'AZ'
```

business_id	name	address	city	city	state	postal_code	stars	review_count
FxqSR54O6m_HtgD18W8ciw	Riviera Coffee Company	6700 W Chicago St, Ste 10	Chandler	Chandler	AZ	85226	4.5	4
CSc_152hwhSHIzto8tLJ7w	Rodeway Inn at Metro Center	9455 N Black Canyon Hwy	Phoenix	Phoenix	AZ	85021	1.5	5
qRHwb1SDhewRJlWD-4C66Q	Schlotzsky's Deli	820 W Warner Rd, Ste 112	Chandler	Chandler	AZ	85224	2.5	3
y0YoEY7JamQjTWju6OqjkQ	Bentley Carpet Installation and Sales	3131 N 35th Ave, Ste O	Phoenix	Phoenix	AZ	85017	4.5	10
bdBpITXrx5hSvCFFWms78g	The Estate Watch & Jewelry Company	7121 E 5th Ave, Ste 23	Scottsdale	Scottsdale	AZ	85251	4.5	73
-DUbOEoenPbeJu35e9Ukrg	99 Cents Only	7830 N 12th St	Phoenix	Phoenix	AZ	85020	3.5	31
0bF6jv97Z6VbruRpepUOcw	Cholla Prime Steakhouse & Lounge	524 N 92nd St	Scottsdale	Scottsdale	AZ	85256	4.5	290
9w48x4HotTBROt6Jq6wQIA	Static Motorsports	9535 E Double Tree Ranch Rd, Unit 140	Scottsdale	Scottsdale	AZ	85258	4.5	17
JIlybDRCddKZAJ2-Y8SKCg	Starpointe Resident Club	17665 W Elliot Rd	Goodyear	Goodyear	AZ	85338	3.0	10
hz2ak-AmOvqW36F4Yx8Gdg	Vitality Med Spa	4434 E Brown Rd, Ste 102	Mesa	Mesa	AZ	85205	4.5	37
KN5W6mXc6Xzv7UZGNnZGRQ	All About Kitchen and Bath		Phoenix	Phoenix	AZ	85254	4.0	29
XoywjZMSIsAfNX0cPF4gDA	Computer Surgeons	4323 E Broadway Rd, Ste 114	Mesa	Mesa	AZ	85206	2.5	3
ir-NrTl2pCiRQQRbK_VNyA	Under One Woof	9617 N Metro Pkwy W, Ste 1116	Phoenix	Phoenix	AZ	85051	4.0	4
_Vi33ZAf0hdBn5Cu0AnQig	Zoes Kitchen	2985 S Alma School Rd	Chandler	Chandler	AZ	85286	3.5	101
Ntp-xe4x9mDAOEgpbNuPSA	Quick Mobile Repair	5555 E Bell Rd, Ste 24	Scottsdale	Scottsdale	AZ	85254	4.0	87

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

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

We'll create an index using the following command:

```sql
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:

```sql
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)

```sql
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:

```sql
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.

Solution

1. Start by creating the table in your own personal schema:

```
CREATE TABLE ychen220.tips AS
SELECT *
FROM public.tips
```

2. Then, add an index on the `business_id` field of the tips table:

```
CREATE INDEX IF NOT EXISTS tips_business_id_index
    ON ychen220.tips (business_id);
```

The `IF NOT EXISTS` clause makes this statement idempotent - we can run it as many times as we need. If we do not have this statement, then the second time we run `CREATE INDEX`, we'd get an error, since the index already exists.

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.

Solution Since our `WHERE` clause uses `compliment_count`, and we can see that Postgres' query planner is performing a sequential scan on all the rows trying to filter out rows that have `compliment_count = 0`, we can add the following index:

```
CREATE INDEX IF NOT EXISTS tips_compliment_count_index
    ON ychen220.tips (compliment_count);
```

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):

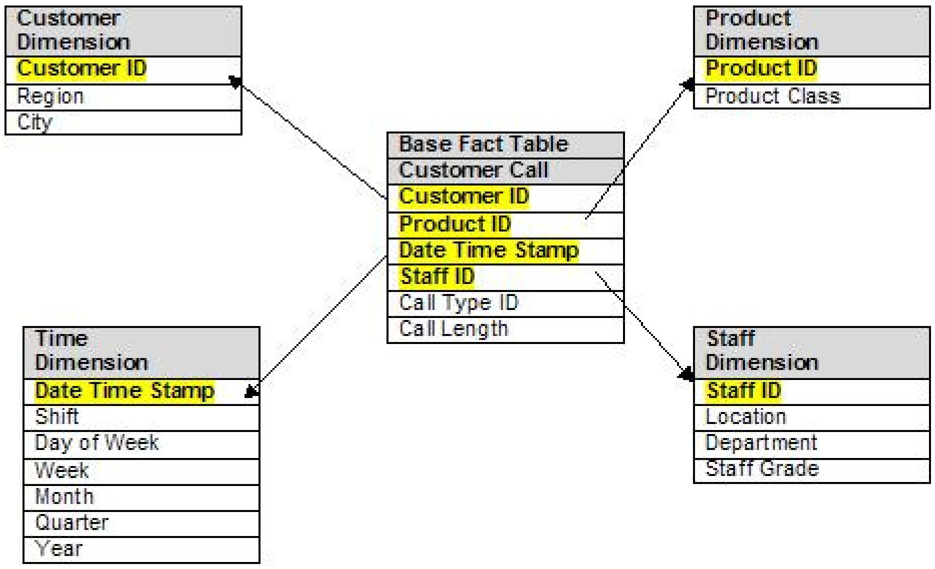


Figure 4: Star Schema Architecture

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.
- **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:

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

You can do so by looking at the result of the following `EXPLAIN` query:

```sql
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)

Solution This question was quite challenging and ended up not being entirely possible for a lot of reasons - after I re-inserted tables and switched to a different size server, the query plan appeared to also have switched.

Therefore - I am going to provide a demonstration of an index being used - but there is no concrete solution for this problem. I will accept any reasonable effort attempt at this problem.

The lab says to add an index to the members table but you will likely find that you still are getting sequential scans. Everything leads you to believe that you should be putting an index on `joindate`. However, remember what I said about very small tables - there's only 33 members in `club` total (a very small amount of data), so there is a decent chance that even if you have added the right index, PostgreSQL will still use a sequential scan (note that an index scan, while efficient, involves two reads from disk often - one read operation to get the index rows, and one read operation to get the actual rows).

If it is a very small table, it may actually be more efficient just to fetch from disk once, and then do a sequential scan, since PostgreSQL can fetch ALL of the rows for that small table in one read operation, and the inefficiency of a sequential scan is offset by not having to do an extra read from disk (which as we learned is quite expensive).

Therefore, the rule of thumb for indices is

> *the larger the table, the greater the likelihood that PostgreSQL will utilize the index, and the larger the optimization gain relative to sequential scans*

All this being said, we can still optimize the query. Look closely at what is the costliest part of the query (where does cost increase the most?). It is likely to be the join between bookings and members, so think about what indices might optimize that particular step. Try next to add an index to either the members or bookings table.

So the first step is to copy over the tables into your personal schema:

```sql
CREATE TABLE ychen220.bookings AS
SELECT *
FROM bookings;

CREATE TABLE ychen220.members AS
SELECT *
FROM ychen220.members;
```

Then you need to add in the existing primary key, foreign key, and indices to these new tables:

```sql
-- add foreign key from bookings to members using memid
ALTER TABLE ychen220.bookings
ADD CONSTRAINT bookings_memid_fk
FOREIGN KEY (memid)
REFERENCES ychen220.members (memid);

-- add the primary key for bookings
ALTER TABLE ychen220.bookings ADD PRIMARY KEY (bookid)

-- add the primary key for members
ALTER TABLE ychen220.members ADD PRIMARY KEY (memid)
```

You also need to add the existing indices:

```sql
CREATE INDEX IF NOT EXISTS "bookings.memid_facid"
    ON public.bookings (memid, facid);

CREATE INDEX IF NOT EXISTS "bookings.facid_memid"
    ON public.bookings (facid, memid);

CREATE INDEX IF NOT EXISTS "bookings.facid_starttime"
    ON public.bookings (facid, starttime);
```

```sql
CREATE INDEX IF NOT EXISTS "bookings.memid_starttime"
    ON public.bookings (memid, starttime);

CREATE INDEX IF NOT EXISTS "bookings.starttime"
    ON public.bookings (starttime);

CREATE INDEX IF NOT EXISTS slots_index
    ON public.bookings (slots);
```

Note, an easy way to do all of this is use the `LIKE` keyword (I should have just introduced this in the lab, I'm sorry!):

```sql
CREATE TABLE ychen220.bookings
(
    LIKE public.bookings INCLUDING INDEXES
);

CREATE TABLE ychen220.members
(
    LIKE public.members INCLUDING INDEXES
);
```

This will copy over all indices and key constraints as well to the new table.

Notice the difference between these two queries. First we filter for a date range that is going to include many, if not most, of the `members` rows:

```sql
EXPLAIN
SELECT *
FROM ychen220.bookings b
JOIN ychen220.members m ON m.memid = b.memid
WHERE joindate > '2012-08-01';
```

QUERY PLAN
Hash Join (cost=10.84..40.75 rows=133 width=1454)
Hash Cond: (b.memid = m.memid)
-> Seq Scan on bookings b (cost=0.00..25.70 rows=1570 width=24)
-> Hash (cost=10.62..10.62 rows=17 width=1430)
-> Seq Scan on members m (cost=0.00..10.62 rows=17 width=1430)
Filter: (joindate > '2012-08-01 00:00:00'::timestamp without time zone)

Now let's filter for a date range that is much smaller, for example, for only one day:

```sql
EXPLAIN
SELECT *
FROM ychen220.bookings b
JOIN ychen220.members m ON m.memid = b.memid
WHERE joindate = '2013-08-01';
```

QUERY PLAN
Nested Loop (cost=4.35..22.61 rows=8 width=1454)
-> Index Scan using joindate_index_members on members m (cost=0.14..8.16 rows=1 width=1430)
Index Cond: (joindate = '2013-08-01 00:00:00'::timestamp without time zone)
-> Bitmap Heap Scan on bookings b (cost=4.21..14.37 rows=8 width=24)
Recheck Cond: (memid = m.memid)
-> Bitmap Index Scan on bookings_memid_starttime_idx (cost=0.00..4.21 rows=8 width=0)
Index Cond: (memid = m.memid)

Notice that using a smaller time window induces Postgres to use the index scan. However, the index used is the `memid_starttime` index. This is actually a totally different index! It is using an index built with the combination of `starttime` and `memid` on the `bookings` table to perform the join from `bookings` to `members`.

Now, let's add an index on `joindate` and observe any effects it has on Postgres' query planning.

```
CREATE INDEX IF NOT EXISTS "joindate_index_members"
    ON ychen220.members (joindate);
```

Let's run the `EXPLAINS` again on our large time range and smaller time range:

```
EXPLAIN
SELECT *
FROM ychen220.bookings b
JOIN ychen220.members m ON m.memid = b.memid
WHERE joindate > '2012-08-01';
```

Notice the `Index Scan` on the join, but the `Seq Scan` on the filter for `joindate`.

QUERY PLAN
Hash Join (cost=10.84..40.75 rows=133 width=1454)
Hash Cond: (b.memid = m.memid)
-> Seq Scan on bookings b (cost=0.00..25.70 rows=1570 width=24)
-> Hash (cost=10.62..10.62 rows=17 width=1430)
-> Seq Scan on members m (cost=0.00..10.62 rows=17 width=1430)
Filter: (joindate > '2012-08-01 00:00:00'::timestamp without time zone)

And then for a smaller time range:

```
EXPLAIN
SELECT *
FROM ychen220.bookings b
JOIN ychen220.members m ON m.memid = b.memid
WHERE joindate = '2013-08-01';
```

This time, we finally get Postgres to use an index when filtering on `joindate`.

QUERY PLAN
Nested Loop (cost=4.35..22.61 rows=8 width=1454)
-> Index Scan using joindate_index_members on members m (cost=0.14..8.16 rows=1 width=1430)
Index Cond: (joindate = '2013-08-01 00:00:00'::timestamp without time zone)
-> Bitmap Heap Scan on bookings b (cost=4.21..14.37 rows=8 width=24)
Recheck Cond: (memid = m.memid)
-> Bitmap Index Scan on bookings_memid_starttime_idx (cost=0.00..4.21 rows=8 width=0)
Index Cond: (memid = m.memid)

In the first example, filtering for a date range that will include most of the rows in `members`, the actual filter on `joindate` operation is still a `Seq Scan`, most likely because there so few rows in members to iterate through that it makes more sense for the query planner to just scan all the rows in one go instead of indirectly getting the index and looking up entries.

In the second example, we finally are filtering for a small enough row count that it makes sense to actually go and look up the date in the `joindate_index_members` index.

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):

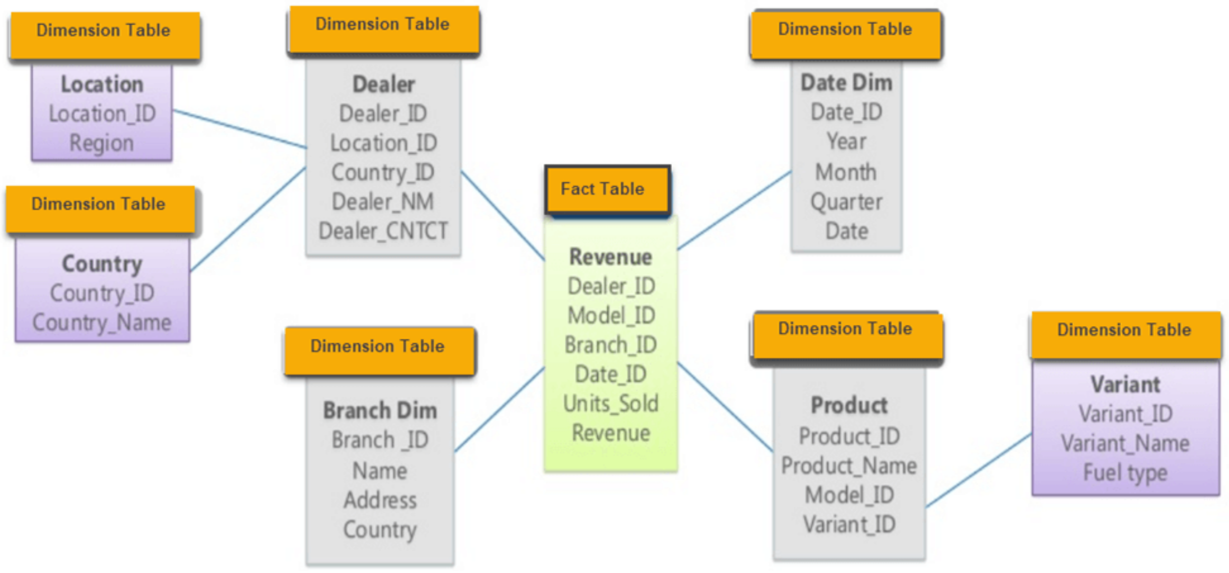


Figure 5: Snowflake Architecture

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