ANALYTICS EXCHANGE NYC

LEARNING SUMMIT 2025

WEDNESDAY, OCTOBER 22- THURSDAY, OCTOBER 23

DATA SCIENCE | DATA GOVERNANCE | DATA ENGINEERING

We'll get started in a few minutes!

ANALYTICS EXCHANGE NYC

LEARNING SUMMIT 2025

WEDNESDAY, OCTOBER 22- THURSDAY, OCTOBER 23

Register for more sessions at bit.ly/anexagenda

WEDNESDAY

9AM - 10AM	**Introduction to Causal Policy Analysis**
10AM - 11AM	**Automating Tasks with GitHub Actions**
11AM - 12PM	**Establishing a Central Data Function**
1PM - 2PM	**Calculating Street Width Right of Way**
2PM - 3PM	**SQL Cheat Codes**

THURSDAY

9AM - 10AM	**Very Effective Data Sharing Agreements**
10AM - 11AM	**Old School SQL Performance Tuning**
11AM - 12PM	**Data Documentation**
1PM - 2PM	**Getting Started with Python and Public APIs**
2PM - 3PM	**Clear and Engaging Data Presentations**

Old School
SQL Performance Tuning

Rebecca Widom
Business Intelligence Developer / Data Engineer
Department of Housing Preservation and Development
Analytics Exchange Learning Summit, October 23, 2025

Agenda

- Intro
- Useful cliches
  - Less is more
  - Once and done
  - Simple is best
- Q & A

Introduction

What is old-school SQL tuning?

Structured Query Language (SQL)

- Relational databases and SQL have been widely used for nearly 50 years
  - Relational model 1970
  - IBM and Oracle develop competing implementations in 70s and early 80s
  - First ISO standards 1987
- The standards are almost never followed 100%
- *What flavor(s) of SQL do you use? What versions?*

Information Retrieval

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

WITHDRAWN

~~ISO 9075:1987~~

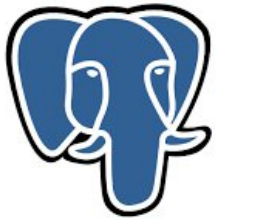
Information processing systems —
Database language — SQL

Withdrawn (Edition 1, 1987)

→ New version available: ISO/IEC 9075:19

Today's Practice Data Environment

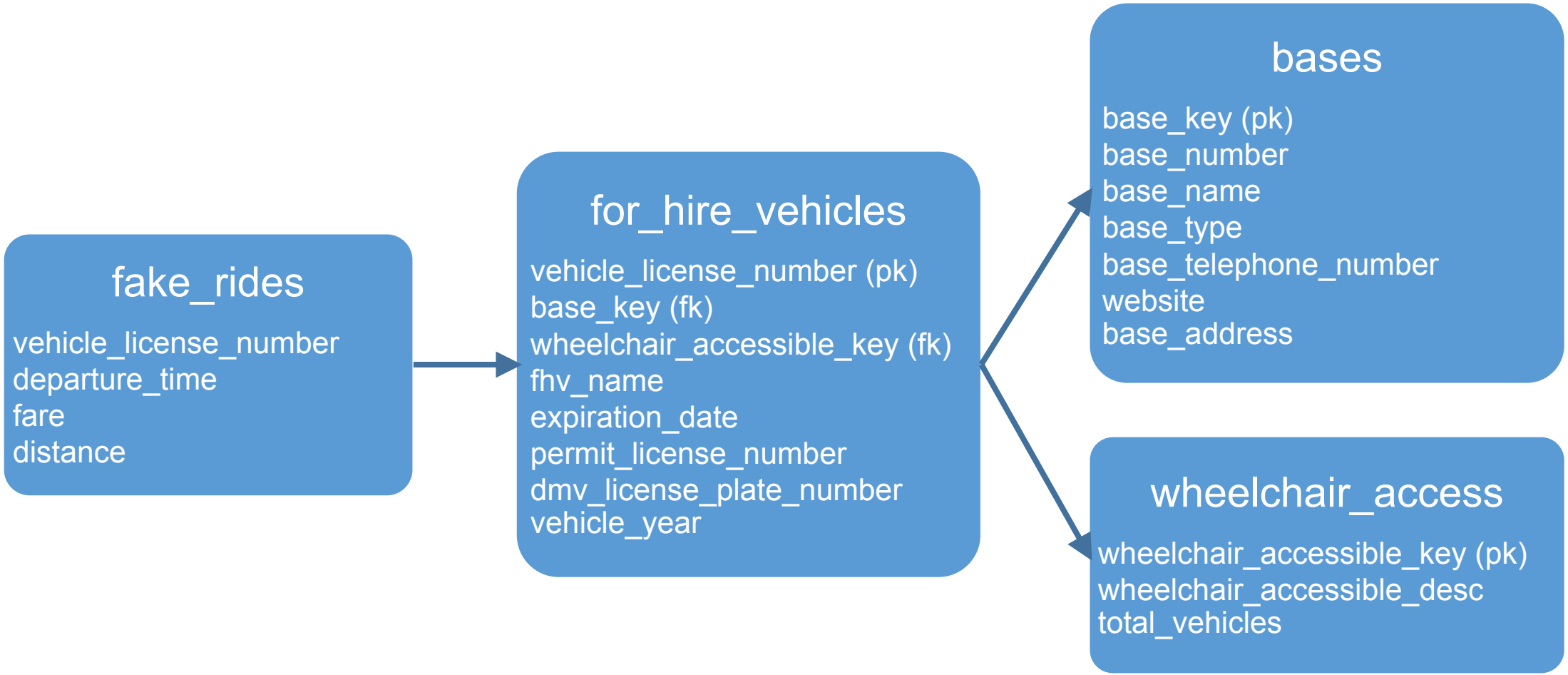
- All of the code and data are on github
  - https://github.com/rwidom/analytics-exchange-sql-tuning
- Some of the data comes from NYC Open Data and some is randomly generated
  - https://data.cityofnewyork.us/Transportation/For-Hire-Vehicles-FHV-Active/8wbx-tsch/about_data
- We'll be using PostgreSQL, because it's free.
  - Not a free trial, just Free and Open Source (FOSS).
  - https://www.postgresql.org/docs/13/index.html
  - Docker (the local virtual machine engine) is not FOSS, but it is free for personal / training use.
  - https://hub.docker.com/layers/library/postgres/13-alpine3.21/



Today’s goal: Explore broad principles

- Different implementations of SQL provide different bells and whistles, particularly when it comes to query performance tuning.
- Performance tuning is about nudging the database to get the same results a little faster.
  - If there were a simple obvious solution, it would probably be built into the query optimizer.
  - So we have to look at trade-offs.
  - But doing it strategically and well can save real time and money.
- We’ll cover some basics that can point to further research in your setting.

Practice Data Model



Less is More

Or at least faster

Basic retrieval times, one table at a time

Table Name	Length (records)	select count(*)	~Width (bytes)	select * limit 10
fake_rides	222,512,920	27,021.725 ms	44	17.404 ms
for_hire_vehicles	104,860	142.584 ms	65	6.299 ms
bases	812	20.614 ms	106	16.465 ms
wheelchair_access	3	13.802 ms	17	3.325 ms

Logical order of operations in SQL

1 FROM

2 WHERE (including Oracle rownum filters)

3 GROUP BY

4 HAVING

5 SELECT

6 DISTINCT

7 ORDER BY

8 LIMIT / FETCH / TOP etc

Database Optimizer Plans

```
training=# explain (select * from fake_rides limit 10);
                                  QUERY PLAN
 Limit  (cost=0.00..0.16 rows=10 width=23)
   ->  Seq Scan on fake_rides  (cost=0.00..3642409.28 rows=222512928 width=23)
(2 rows)

training=# explain (select count(*) from fake_rides);
                                              QUERY PLAN
 Finalize Aggregate  (cost=2577201.71..2577201.72 rows=1 width=8)
   ->  Gather  (cost=2577201.50..2577201.71 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=2576201.50..2576201.51 rows=1 width=8)
               ->  Parallel Seq Scan on fake_rides  (cost=0.00..2344417.20
rows=92713720 width=0)
```

Logical Order of Execution

```
select
    distinct 6
    wheelchair_accessible_key, wheelchair_accessible_desc, 5
    count(*) records
from for_hire_vehicles 1
    join bases using (base_key)
    join wheelchair_access using (wheelchair_accessible_key)
where bases.base_type = 'BLACK-CAR' 2
group by wheelchair_accessible_key, wheelchair_accessible_desc 3
having count(*)>100 4
order by wheelchair_accessible_key 7
limit 10; 8
```

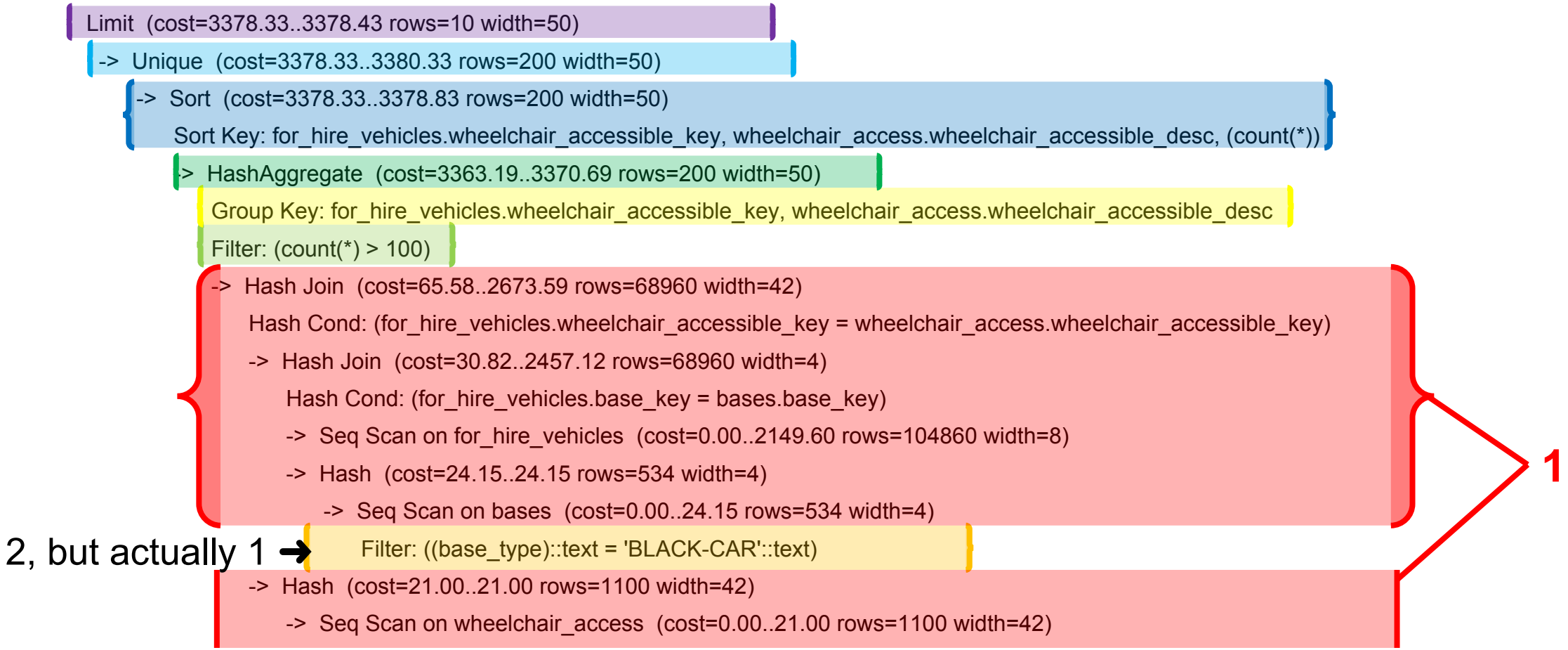
Logical Order of Execution (inside first)

Step 2 comes first here, because it's inside.

```
select
    distinct
    wheelchair_accessible_key, wheelchair_accessible_desc,
    count(*) records
from for_hire_vehicles
    join (select * from bases where base_type = 'BLACK-CAR') bases
        using (base_key)
    join wheelchair_access using (wheelchair_accessible_key)
group by wheelchair_accessible_key, wheelchair_accessible_desc
having count(*)>100
order by wheelchair_accessible_key
limit 10;
```

Both versions use the same plan

The optimizer “plan” is how the database actually goes to retrieve the data for your query results.



When can’t the optimizer apply the filter first?

```
select wheelchair_accessible_key, wheelchair_accessible_desc,
    count(*) pilot_ride_count
from fake_rides
    join for_hire_vehicles using (vehicle_license_number)
    join wheelchair_access using (wheelchair_accessible_key)
where wheelchair_accessible_desc='PILOT'
    and fare<wheelchair_accessible_key
group by wheelchair_accessible_key,
    wheelchair_accessible_desc;
```

Comparing Results from Two Tables with < or >

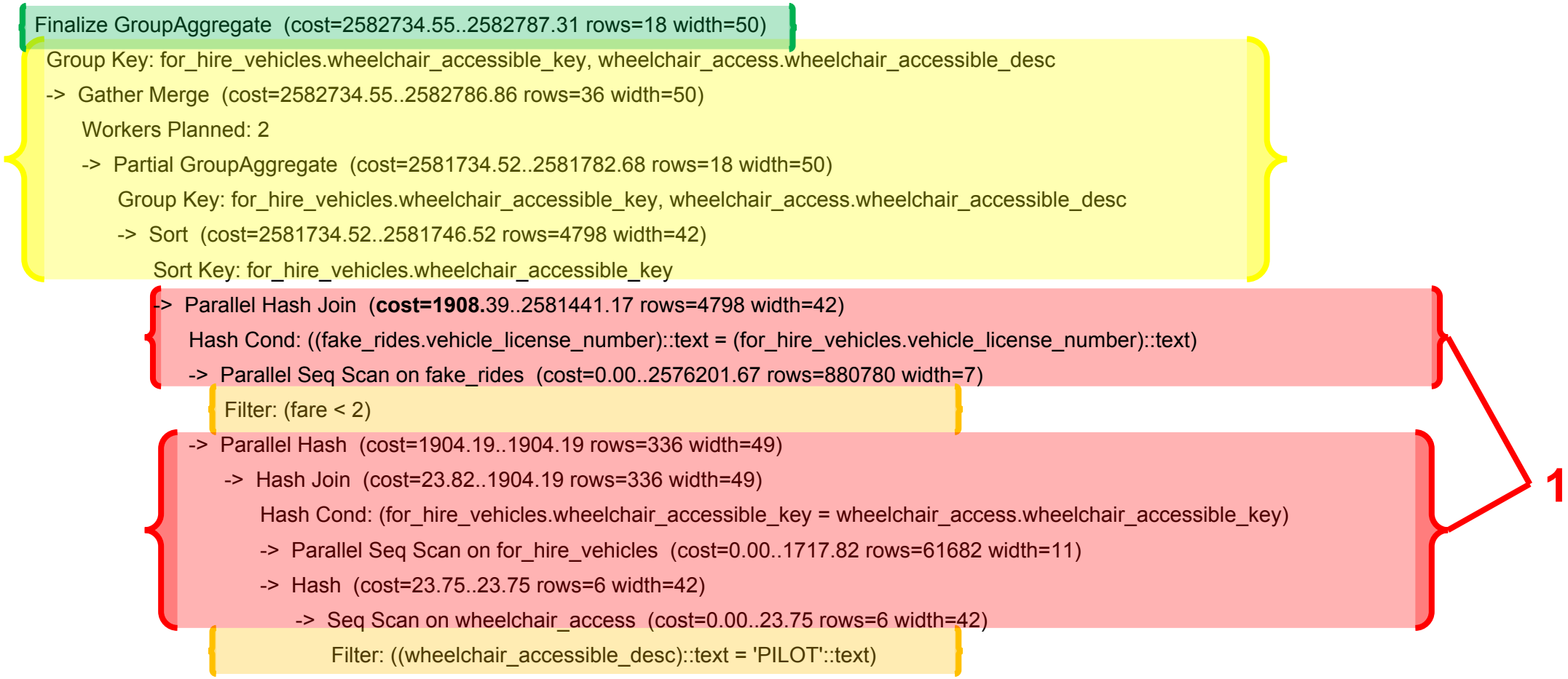
```
Finalize GroupAggregate  (cost=2068411.81..2069275.56 rows=18 width=50)
  Group Key: for_hire_vehicles.wheelchair_accessible_key, wheelchair_access.wheelchair_accessible_desc
  ->  Gather Merge  (cost=2068411.81..2069275.11 rows=36 width=50)
        Workers Planned: 2
        ->  Partial GroupAggregate  (cost=2067411.79..2068270.93 rows=18 width=50)
              Group Key: for_hire_vehicles.wheelchair_accessible_key, wheelchair_access.wheelchair_accessible_desc
              ->  Sort  (cost=2067411.79..2067626.53 rows=85896 width=42)
                    Sort Key: for_hire_vehicles.wheelchair_accessible_key
                    ->  Hash Join  (cost=2512.67..2057729.98 rows=85896 width=42)
                          Hash Cond: (for_hire_vehicles.wheelchair_accessible_key = wheelchair_access.wheelchair_accessible_key)
                          ->  Parallel Hash Join  (cost=2488.85..2016209.56 rows=15747555 width=4)
                                Hash Cond: ((fake_rides.vehicle_license_number)::text = (for_hire_vehicles.vehicle_license_number)::text)
                                Join Filter: (fake_rides.fare < for_hire_vehicles.wheelchair_accessible_key)
                                ->  Parallel Seq Scan on fake_rides  (cost=0.00..1889706.67 rows=47242667 width=62)
                                ->  Parallel Hash  (cost=1717.82..1717.82 rows=61682 width=11)
                                      ->  Parallel Seq Scan on for_hire_vehicles  (cost=0.00..1717.82 rows=61682 width=11)
                          ->  Hash  (cost=23.75..23.75 rows=6 width=42)
                                ->  Seq Scan on wheelchair_access  (cost=0.00..23.75 rows=6 width=42)
                                      Filter: ((wheelchair_accessible_desc)::text = 'PILOT'::text)
```

1

Is it possible to filter on a constant (2) instead?

```
select wheelchair_accessible_key,
    wheelchair_accessible_desc,
    count(*) pilot_ride_count
from fake_rides
    join for_hire_vehicles using (vehicle_license_number)
    join wheelchair_access using (wheelchair_accessible_key)
where wheelchair_accessible_desc='PILOT' and fare<2
group by wheelchair_accessible_key,
wheelchair_accessible_desc;
```

Filtering on a constant here means we don't have to join 99% of the records, which saves time for the same results.



Less is More

- Just like for humans, databases take more time when they have to do more stuff.
- To the extent that we can help the database filter out records earlier in the process, we can help the query go faster.
- Exactly how to do that will depend on the specific database you're using, but there should be a version of "explain plan" that you can use to get more information about how the database is handling your sql.

Once and Done

Doing the same thing over and over takes longer than doing it once.

Preparing for a Needle in a Haystack with Indexes

- Scanning all records in a large table takes time.
- A database index is like a book index.
  - It's less pages to sort through.
  - It helps you find what you want faster.
  - It does not have the level of detail of the book.
  - It takes time to create and maintain.
- Here, creating an index took ~2 minutes.
- **Using the index reduced query time from 26 seconds to <1 second.**

```
select count(*) from fake_rides where
departure_time >= '2025-09-01';
--    count
--  5452720
-- Time: 26447.360 ms (00:26.447)
create index ind_departure_time on
fake_rides (departure_time);
-- Time: 108513.257 ms (01:48.513)
select count(*) from fake_rides where
departure_time >= '2025-09-01';
--    count
--  5452720
-- Time: 992.404 ms
```

Optimizer Plan Without and With an Index

Finalize Aggregate (**cost=2,583,271.57**..2583271.58 rows=1 width=8)
-> Gather (cost=2583271.35..2583271.56 rows=2 width=8)
Workers Planned: 2
-> Partial Aggregate (cost=2582271.35..2582271.36 rows=1 width=8)
-> Parallel **Seq Scan on fake_rides** (cost=0.00..2576201.67 rows=2427874 width=0)
Filter: (departure_time >= '2025-09-01 00:00:00'::timestamp without time zone)

Finalize Aggregate (**cost=138,958.92**..138958.93 rows=1 width=8)
-> Gather (cost=138958.71..138958.92 rows=2 width=8)
Workers Planned: 2
-> Partial Aggregate (cost=137958.71..137958.72 rows=1 width=8)
-> Parallel **Index Only Scan using ind_departure_time** on fake_rides (cost=0.57..131889.02 rows=2427873 width=0)
Index Cond: (departure_time >= '2025-09-01 00:00:00'::timestamp without time zone)

Wow! 26 times faster! What's the catch?

A few caveats to keep in mind...

- Indexes take time to create and maintain, and they affect more than just your individual query. If you ask for one, and get a "no", be curious, not offended.
- In this particular case, the query itself was just for a count matching index criteria, so the database did not even need to look at the table itself. ("index only scan") The benefit of creating an index is not always as dramatic in more complex queries.
- And, there *may* already be an index on another field you can use.
  - For example, maybe you're looking for records on a particular program that started this year.
  - Maybe there isn't an index on program name, but there is a date index that you could use by adding a filter to your where clause.
  - In those cases, you can add a filter that does not change your results, but does give the database optimizer more information to finish your query more quickly.

Correlated Sub-queries

```
1.select fhv_name,
2.    vehicle_license_number,
3.    wheelchair_accessible_desc
4.from for_hire_vehicles v
5.    join wheelchair_access using
  (wheelchair_accessible_key)
6.where base_key in (
7.    select base_key
8.    from bases b
9.    where b.base_type = 'BLACK-CAR'
10.        and v.fhv_name = b.base_name
11.)
12.order by fhv_name;
```

- Lines 6-11 have a sub-query.
- The way it is written, it cannot be executed once and done.
- Line 10 requires comparing results from the inner and outer queries.
- So, the database has to run the sub-query over and over for each and every record in the outer query.

Saving Time by Uncorrelating the Sub-query

```
1.select fhv_name,
2.    vehicle_license_number,
3.    wheelchair_accessible_desc
4.from for_hire_vehicles
5.    join wheelchair_access using
 (wheelchair_accessible_key)
6.where fhv_name in (
7.    select DISTINCT BASE_NAME
8.    from bases
9.    where base_type = 'BLACK-CAR'
10.)
11.order by fhv_name;
```

- Lines 6-10 have a sub-query.
- The comparison of fhv_name to base_name happens ***outside*** of the subquery.
  - Here it's in the where clause
  - It could be part of a join instead
  - Or use "exists" syntax if you don't want to change the subquery.
- This took about 30 milliseconds to complete, because it didn't have to run the same query over and over.

Common Table Expressions (CTEs)

Advantages

- Make more complicated queries easier to read by breaking them down into steps.
- If you have a bunch of needles from a large haystack or a summary of a large table that you will reuse in multiple ways, better to do that work once.

Caveats / Pitfalls

- Although the database will often save a CTE as if it were a regular table, it won't create any indexes on the CTE, so you need to balance the complexity of calculations and filters against the benefits of indexes and other performance infrastructure in the original table.

Once and Done

- You can save time by preparing the data base to execute your query as efficiently as possible, but with preparation there are often trade-offs.
- There are options for how to avoid correlated sub-queries, pick one!
- Indexes and common table expressions can be very helpful, but they are not one-size-fits all solutions.

Simple is best

Also known as give the optimizer all possible information and then trust it.

Don’t use count distinct if you don’t have to. Count is much easier on the database.

```
select
    wheelchair_accessible_key,
    wheelchair_accessible_desc,
    count(distinct vehicle_license_number) veh
from fake_rides
    join for_hire_vehicles using (vehicle_license_number)
    join wheelchair_access using (wheelchair_accessible_key)
group by wheelchair_accessible_key, wheelchair_accessible_desc;
```

- Get a count of vehicles with rides in fake_rides by accessibility status.
- The joins here require count distinct because a single car can have multiple rides.
- But all we really need to know is whether there are *any* rides for the car, and this query took almost 10 minutes to run.

Exists is your friend.

```
 select wheelchair_accessible_key, wheelchair_accessible_desc, count(*) veh
from for_hire_vehicles v
    join wheelchair_access using (wheelchair_accessible_key)
where exists(
    select 1 from fake_rides r
    where r.vehicle_license_number = v.vehicle_license_number
 )
group by wheelchair_accessible_key, wheelchair_accessible_desc;
```

- Exists tells you only what you need to know here: whether there are *any* rides for the car, and it tells the optimizer not to do a correlated sub-query.
- This version produces the same results in about a minute and a half.

Know your dataset vs. the devil is in the details.

```
select * from wheelchair_access;
```

- This is a training dataset, and I cannot promise that there will always be the exact metric you're looking for ready and waiting for you to use it.
- But one of the jobs of a well-designed analytics system (aka data warehouse, data mart, etc) is to look for commonly used calculations and take care of them in regular updates so that they do not have to be calculated on the fly.
- In this case, the difference between 10 minutes and 15 milliseconds, tells me that it's worth checking out the documentation about your dataset and asking the question.
- And if there isn't a good data dictionary with meaningful descriptions, you can start creating one!

Does the query planner have the best possible info?

- Handling tables with 2 records vs 200,000,000 records.
- How and when does your database collect tables statistics?
- **Get the optimizer the best information and then trust it!**
- Relevant info:
  - table statistics
  - business / data rules and relationships

Keep it simple

- There may be ways to use table structures and to simplify necessary calculations and save time.
- Focus on making sure that the query planner has the best information, not micromanaging it.

Questions & Answers

Helpful cliches: Less is more, Once and done, Simple is best.