Query lifecycle and the planner

IMPROVING QUERY PERFORMANCE IN POSTGRESQL

Amy McCarty
Instructor

Basic query lifecycle

	System	Front end steps	Back end processes
1	**Parser**	Send query to database	Checks syntax. Translates SQL into more computer friendly syntax based on system stored rules.
2	**Planner & Optimizer**	Assess and optimize query tasks	Uses database stats to create query plan. Calculates costs and chooses the best plan.
3	**Executor**	Return query results	Follows the query plan to execute the query.

Query planner and optimizer

Responsive to SQL structure changes

- Generates plan trees
  - Nodes corresponding to steps
  - Visualize with EXPLAIN
- Estimate cost of each tree
  - Statistics from pg_tables
  - Time based optimization

[1] Plan tree: https://www.postgresql.org/docs/current/querytree.html

Statistics from pg_tables

```
SELECT * FROM pg_class
WHERE relname = 'mytable'
```

```
-- sample of output columns
| relname | relhasindex |
```

```
SELECT * FROM pg_stats
WHERE tablename = 'mytable'
```

```
-- sample of output columns
null_frac | avg_width | n_distinct |
```

- Column indexes
- Count null values
- Column width
- Distinct values

EXPLAIN

- Window into query plan
- Steps and cost **estimates**
  - Does not run query

```
EXPLAIN
SELECT * FROM cheeses
```

- Sequential scan of cheeses table
- Cost and size estimates

```
Seq Scan on cheeses
(cost=0.00..10.50 rows=5725 width=296)
```

EXPLAIN: Scan

- Query plan step
- Returns rows

```
Seq Scan on cheeses (cost=0.00..10.50 rows=5725 width=296)
```

- **Seq Scan** : scan of all the rows in table

EXPLAIN: Cost

- Dimensionless
- Compare structures with same output
  - Should **not** compare queries with different output

Seq Scan on cheeses (**cost=0.00..10.50** rows=5725 width=296)

- **0.00..** : start up time
- **..10.50** : total time
- total time = start up + run time

EXPLAIN: Size

- Size estimates

Seq Scan on cheeses (cost=0.00..10.50 **rows=5725 width=296**)

- **rows** : rows query needs to examine to run
- **width** : byte width of rows

EXPLAIN with a WHERE clause

```
EXPLAIN
SELECT * FROM cheeses WHERE species IN ('goat','sheep')
```

```
Seq Scan on cheeses (cost=0.00..378.90 rows=3 width=118)
 -> Filter: (species = ANY ('{"goat","sheep"}'::text[]))
```

- From bottom to top
  - Step 1: Filter
  - Step 2: Sequential scan
- WHERE clause
  - Decrease rows to scan and increases total cost

EXPLAIN with an index

```
EXPLAIN
SELECT * FROM cheeses WHERE species IN ('goat','sheep') -- index on species column
```

```
Bitmap Index Scan using species_idx on cheeses (cost=0.29..12.66 rows=3 width=118)
  Index Cond: (species = ANY ('{"goat","sheep"}'::text[]))
```

- Step 1: Bitmap Index Scan
  - Index Cond explains the scan step
- INDEX
  - Start up cost increased from 0
  - Overall cost decreased from 379

Let's practice!

IMPROVING QUERY PERFORMANCE IN POSTGRESQL

A deeper dive into EXPLAIN

IMPROVING QUERY PERFORMANCE IN POSTGRESQL

Amy McCarty
Instructor

EXPLAIN optional parameters

VERBOSE

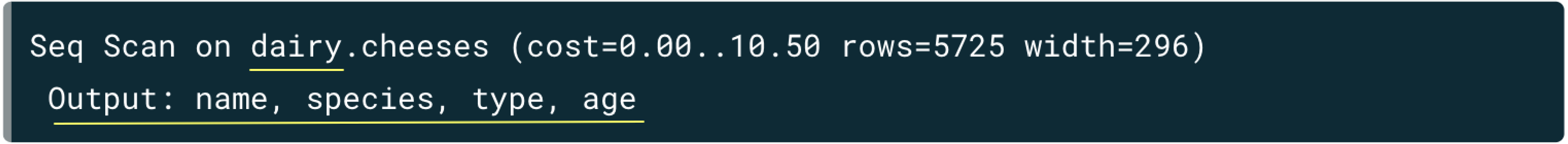
- Columns for each plan node
- Shows table schema and aliases

ANALYZE

- Runs the query
- Actual run times in milliseconds

VERBOSE

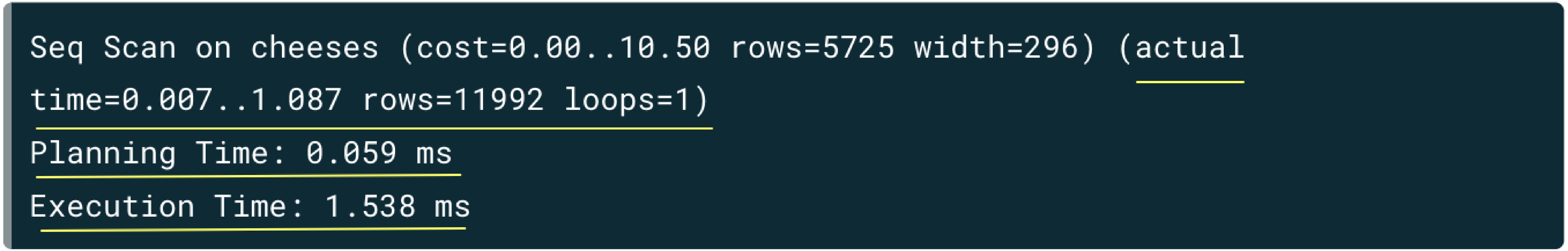
```
EXPLAIN VERBOSE
SELECT * FROM cheeses
```



```
Seq Scan on dairy.cheeses (cost=0.00..10.50 rows=5725 width=296)
 Output: name, species, type, age
```

ANALYZE

```
EXPLAIN ANALYZE
SELECT * FROM cheeses
```



```
Seq Scan on cheeses (cost=0.00..10.50 rows=5725 width=296) (actual
time=0.007..1.087 rows=11992 loops=1)
Planning Time: 0.059 ms
Execution Time: 1.538 ms
```

- Most useful to minimize run time

Query plan - aggregations

```
EXPLAIN ANALYZE
SELECT type, AVG(age) AS avg_age
FROM cheeses
GROUP BY type -- hard or soft cheese
```

```
HashAggregate  (cost=314.88..317.38 rows=200 width=40)(actual time = 4.973..4.975
                rows=2 loops=1)
  Group Key: type
  ->  Seq Scan on cheeses  (cost=0.00..286.25 rows=5725 width=10)(actual time =
                            0.016..2.546 rows = 11992 loops=1)
Planning Time: 12.891 ms
Execution Time: 5.074 ms
```

Query plan - sort

```
EXPLAIN ANALYZE
SELECT name, age
FROM cheeses
ORDER BY age DESC
```

```
Sort  (cost=1161.37..1191.35 rows=11992 width=20)(actual time = 4.281..5.331
       rows=11992 loops=1)
  Sort Key: age DESC
  Sort Method: quicksort Memory: 1216kB
  ->  Seq Scan on cheeses  (cost=0.00..348.92 rows=11992 width=20)(actual time =
                            0.0007..1.799 rows = 11992 loops=1)
Planning Time: 0.131 ms
Execution Time: 5.870 ms
```

```sql
EXPLAIN ANALYZE
SELECT name, age FROM cheeses
INNER JOIN animals ON cheeses.species = animals.species
```

```
Hash Join  (cost=182.97..4339.35 rows=335776 width=145)(actual time=2.755..138.418
            rows=335776 loops=1)
  Hash Cond: (cheeses.species = animals.species)
  ->  Seq Scan on cheeses  (cost=0.00..348.92 rows=11992 width=118) (actual
                            time=0.010..2.271 rows=11992 loops=1)
  ->  Hash  (cost=106.32..106.32 rows=6132 width=27) (actual time=2.725..2.725 rows=6132
             loops=1)
        Buckets: 8192  Batches: 1  Memory Usage: 439kB
        ->  Seq Scan on animals  (cost=0.00..106.32 rows=6132 width=27) (actual
                                  time=0.009..1.008 rows=6132 loops=1)
Planning Time: 0.379 ms
Execution Time: 161.918 ms
```

Let's practice!

IMPROVING QUERY PERFORMANCE IN POSTGRESQL

Query structure and query execution

IMPROVING QUERY PERFORMANCE IN POSTGRESQL

Amy McCarty
Instructor

Subqueries and joins

```
-- SUBQUERY
SELECT COUNT(athlete_id)
FROM athletes
WHERE country IN
  (SELECT country FROM climate
    WHERE temp_annual > 22)
```

```
-- JOIN
SELECT COUNT(athlete_id)
FROM athletes a
INNER JOIN climate c
  ON a.country = c.country
  AND c.temp_annual > 22
```

Query plan

```
Aggregate  ()
  ->  Hash Join  ()
        Hash Cond: (athletes.country = climate.country)
        ->  Seq Scan on athletes  ()
        ->  Hash  ()
              ->  Seq Scan on climate  ()
                    Filter: (temp_annual > '22'::numeric)
```

Common table expressions and temporary tables

```sql
-- CTE
WITH celsius AS
(
  SELECT country
  FROM climate
  WHERE temp_annual > 22 -- Celsius
)
SELECT count(athlete_id)
FROM athletes a
INNER JOIN celsius c
  ON a.country = c.country
```

```sql
-- TEMP TABLE
CREATE TEMPORARY TABLE celsius AS
  SELECT country
  FROM climate
  WHERE temp_annual > 22; -- Celsius

SELECT count(athlete_id)
FROM athletes a
INNER JOIN celsius c
  ON a.country = c.country
```

Query plan

```
Aggregate  ()
CTE celsius
  ->  Seq Scan on climate  ()
        Filter: (temp_annual > '22'::numeric)
->  Hash Join  ()
      Hash Cond: (a.country_code = c.country_code)
      ->  Seq Scan on athletes a  ()
      ->  Hash  ()
            ->  CTE Scan on celsius c  ()
```

Limiting the data

```sql
SELECT country_code
, COUNT(athlete_id) as athletes
FROM athletes
WHERE year in (2014, 2010) -- Indexed column
GROUP BY country_code
```

Limiting the data

```
SELECT country_code
, COUNT(athlete_id) as athletes
FROM athletes
WHERE year in (2014, 2010) -- Indexed column
GROUP BY country_code
```

No Index	Index
Planning Time: 3.370 ms	Planning Time: 0.163 ms
Execution Time: 0.143 ms	Execution Time: 0.062 ms

Aggregations - different granularities

```
SELECT r.country
  , COUNT(a.athlete_id) as athletes
FROM regions r -- country level
INNER JOIN athletes a -- athletes level
  ON r.country = a.country
GROUP BY r.country
```

- Execution Time : 0.267 ms

Aggregations - changing the granularity

```
WITH olympians AS ( -- country level
  SELECT country
  , COUNT(athlete_id) as athletes
  FROM athletes -- athletes level
  GROUP BY country
)
SELECT country, athletes
FROM regions r -- country level
INNER JOIN olympians o
  ON r.country = o.country
```

	Execution Time
Join 1st	0.267 ms
Aggregate 1st	0.192 ms

Let's practice!

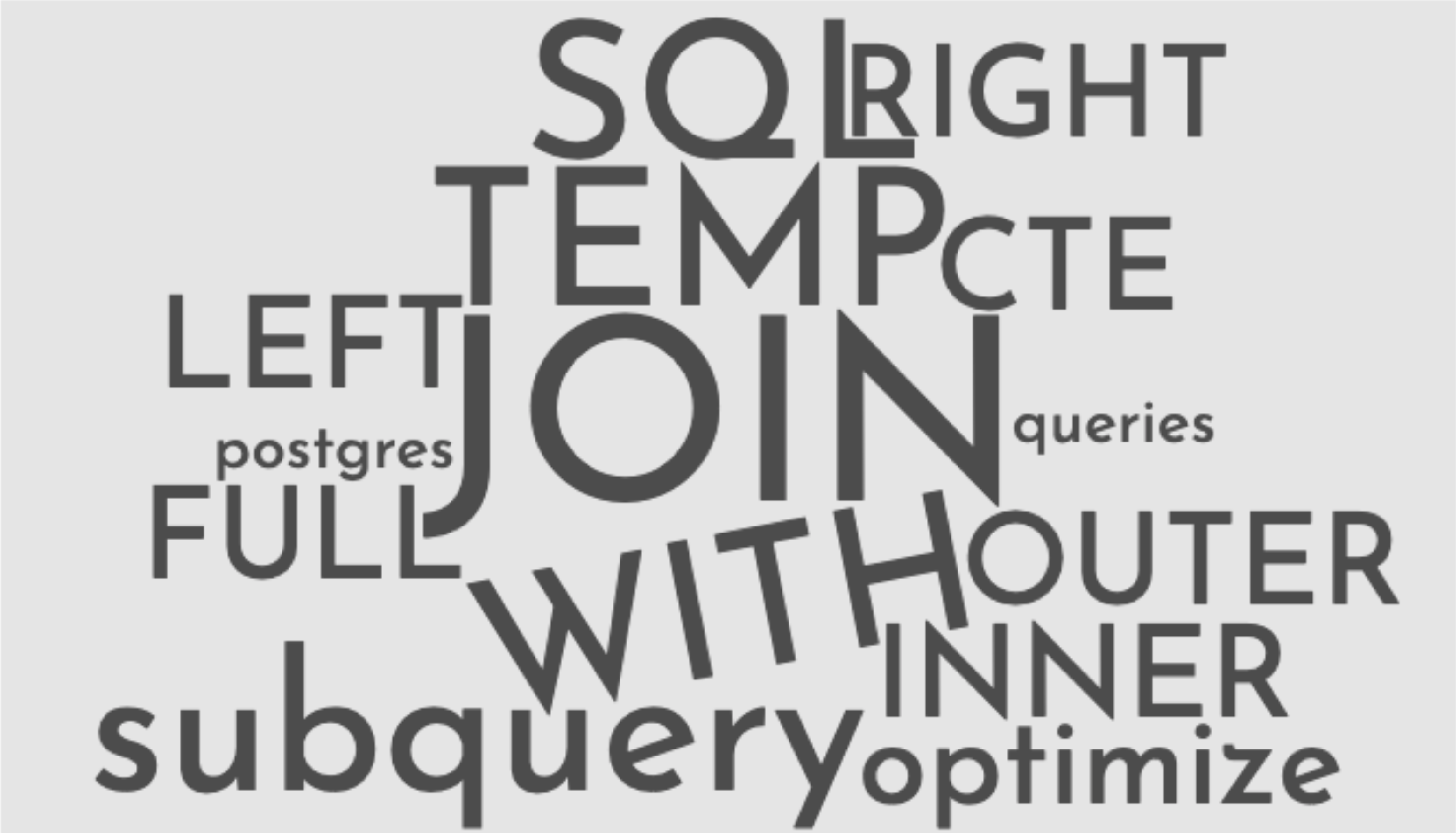
IMPROVING QUERY PERFORMANCE IN POSTGRESQL

Congratulations

IMPROVING QUERY PERFORMANCE IN POSTGRESQL

Amy McCarty
Instructor

Chapter 1: ways to combine data

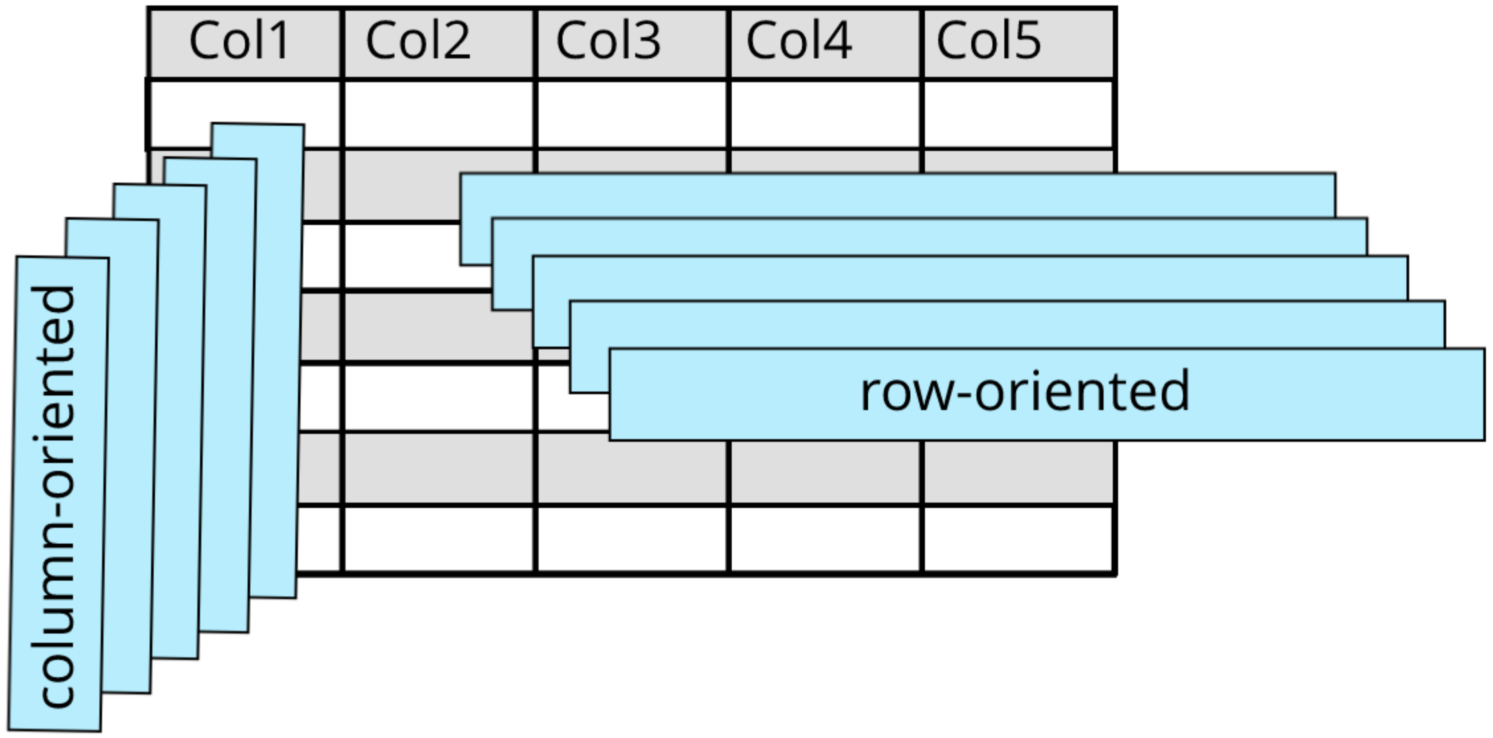


Chapter 2: limiting the results

- Order of operations
- Filtering in the WHERE
- Filtering with an INNER JOIN
- Joining different data granularities

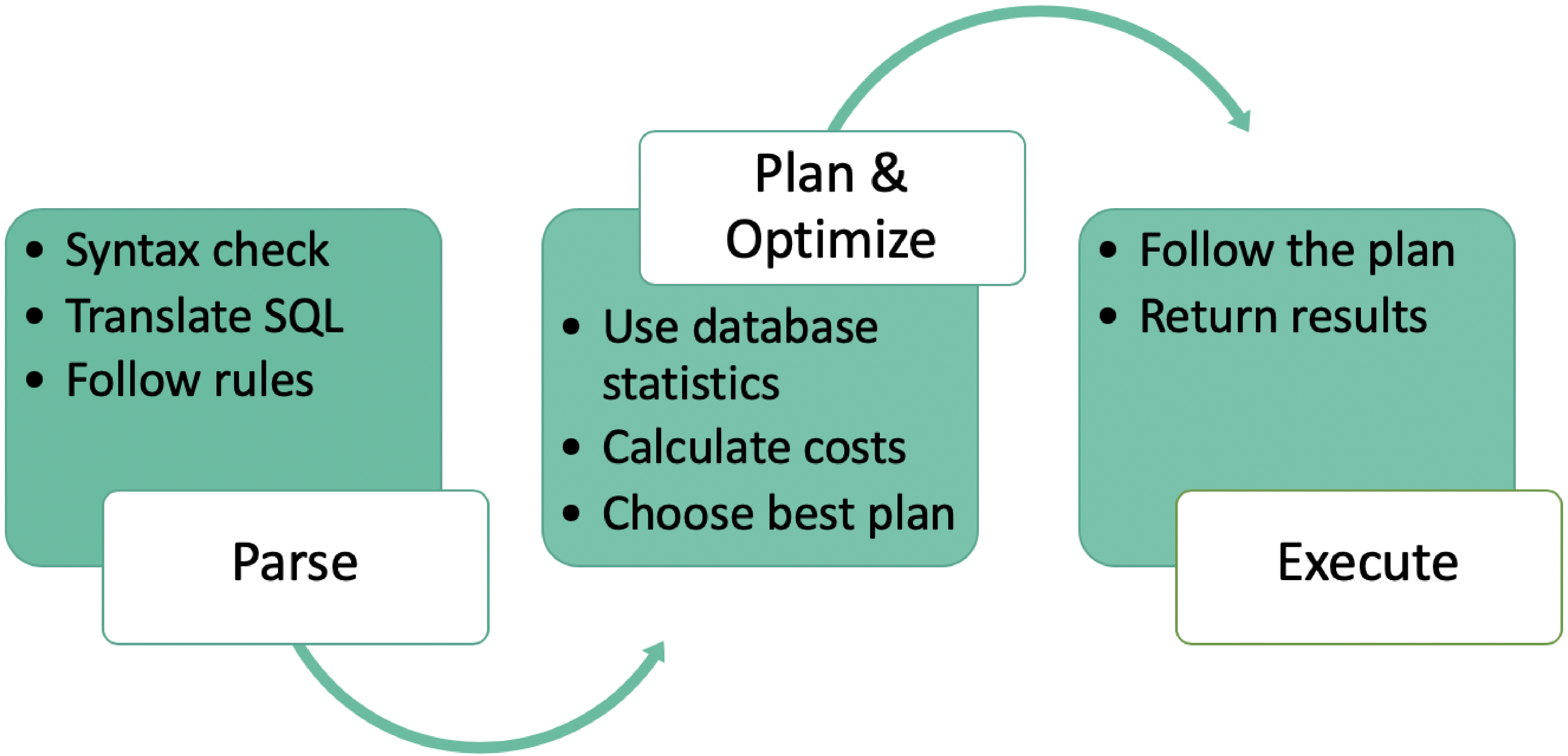
Chapter 3: learning the database

- pg_tables and information_schema
- Tables and views



- Indexes and partitions

Chapter 4: using the query planner



- Syntax check
- Translate SQL
- Follow rules

Parse

Plan & Optimize

- Use database statistics
- Calculate costs
- Choose best plan

- Follow the plan
- Return results

Execute

- EXPLAIN

Thank you!

IMPROVING QUERY PERFORMANCE IN POSTGRESQL