Performance issues

Indexes

- Indexes are a common way to **enhance database performance**.
  - An index allows the database server to find and retrieve specific rows **much faster** than it could do without an index.
  - But indexes also **add overhead to the database system** as a whole, so they should be used sensibly

Create index

- **CREATE INDEX** test1_id_index **ON** test1 (id);
- **CREATE INDEX** test1_id_index **ON** test1 **USING** btree (id);

- **CREATE INDEX** test1_id_index **ON** test1 [USING btree] (id) **WHERE** <condition>;

➔ Partial index

Index types in PostgreSQL

- PostgreSQL provides several index types: B-tree, Hash, GiST, SP-GiST, GIN, BRIN
- Each index type uses a different algorithm that is best suited to different types of queries.

By default, the CREATE INDEX command creates **B-tree** indexes, which **fit the most common situations**

Index types in PostgreSQL

- **B-Tree** (default)
  - handle equality and range queries on data that can be sorted into some ordering.
  - Operators: <, ≤, = , ≥, > , LIKE (col LIKE **'foo%'** but not col LIKE '%bar')
  - Sorted output
- **Hash index**: can only handle simple equality comparisons
- **GiST index**: for several two-dimensional geometric data types,
  - not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented
- **GIN index**
  - inverted indexes which can handle values that contain more than one key, arrays for example

Index types in PostgreSQL

- **spgist index**: …………
- **Brin:** …..

Multicolumn index

- **CREATE INDEX** test2_mm_idx **ON** test2 (major, minor);

- B-Tree
- GiST index
- GIN index

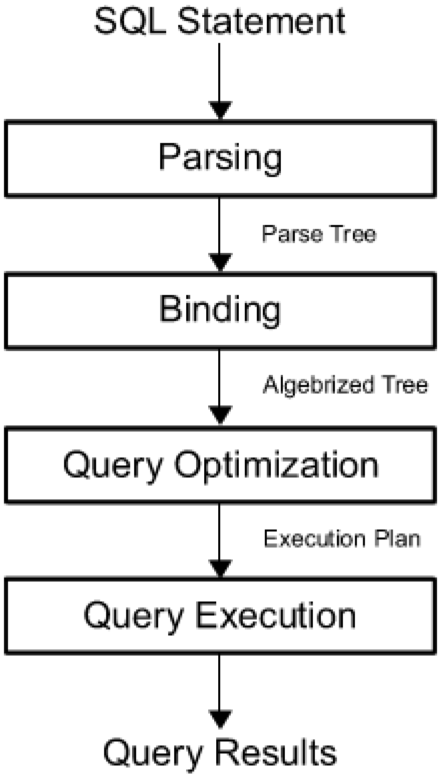
https://www.postgresql.org/docs/10/sql-createindex.html

https://www.postgresql.org/docs/10/indexes.html

Examining index usage

- EXPLAIN [ANALYZE] [VERBOSE] *statement*
  - EXPLAIN *statement:* displays the execution plan that the PostgreSQL planner generates for the supplied statement.

    Actually two numbers are shown: the start-up cost before the first row can be returned, and the total cost to return all the rows.
  - **VERBOSE** option: displays additional information regarding the plan (output column list, table and function names, …)



Examining index usage

- EXPLAIN [ANALYZE] [VERBOSE] *statement*
  - **ANALYZE** option: causes the statement to be actually executed, not only planned, actual runtime statistics are added to the display
- **Important:** If you wish to use EXPLAIN ANALYZE on an INSERT, UPDATE, DELETE, CREATE TABLE AS, or EXECUTE statement without letting the command affect your data, use this approach:

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

View table indexes

- \d table_name
- Ex.: \d customers

Tips

- Select fewer columns to improve hash join performance
- Index the *independent* **where** predicates to improve hash join performance

Tips

- Having a **WHERE / HAVING** clause in your queries does not necessarily means that it is a bad query
- Only retrieve the data you need
  - remove unnecessary columns from SELECT
  - Inner join vs. exists (with subqueries)
  - Select **DISTINCT** : try to avoid if you can
  - **LIKE** operator: the index isn't used if the pattern starts with % or _

- Limit your results : LIMIT, TOP
- Don’t Make Queries More Complex Than They Need To Be
  - OR / IN / UNION ?
  - OR operator : index is not used except composite index → IN/UNION/OUTER JOIN
  - NOT operator: index is not used => avoid
  - AND vs BETWEEN
  - ANY / ALL: index not used => max , min ,…
  - Isolate columns in Condition : age + 7 < 20 → age < 13

- Limit your results : **LIMIT, TOP**

You can add the LIMIT or TOP clauses to your queries to set a maximum number of rows for the result set.

```
SELECT TOP 3 *
FROM customers;

SELECT  *
FROM customers
LIMIT 3;
```

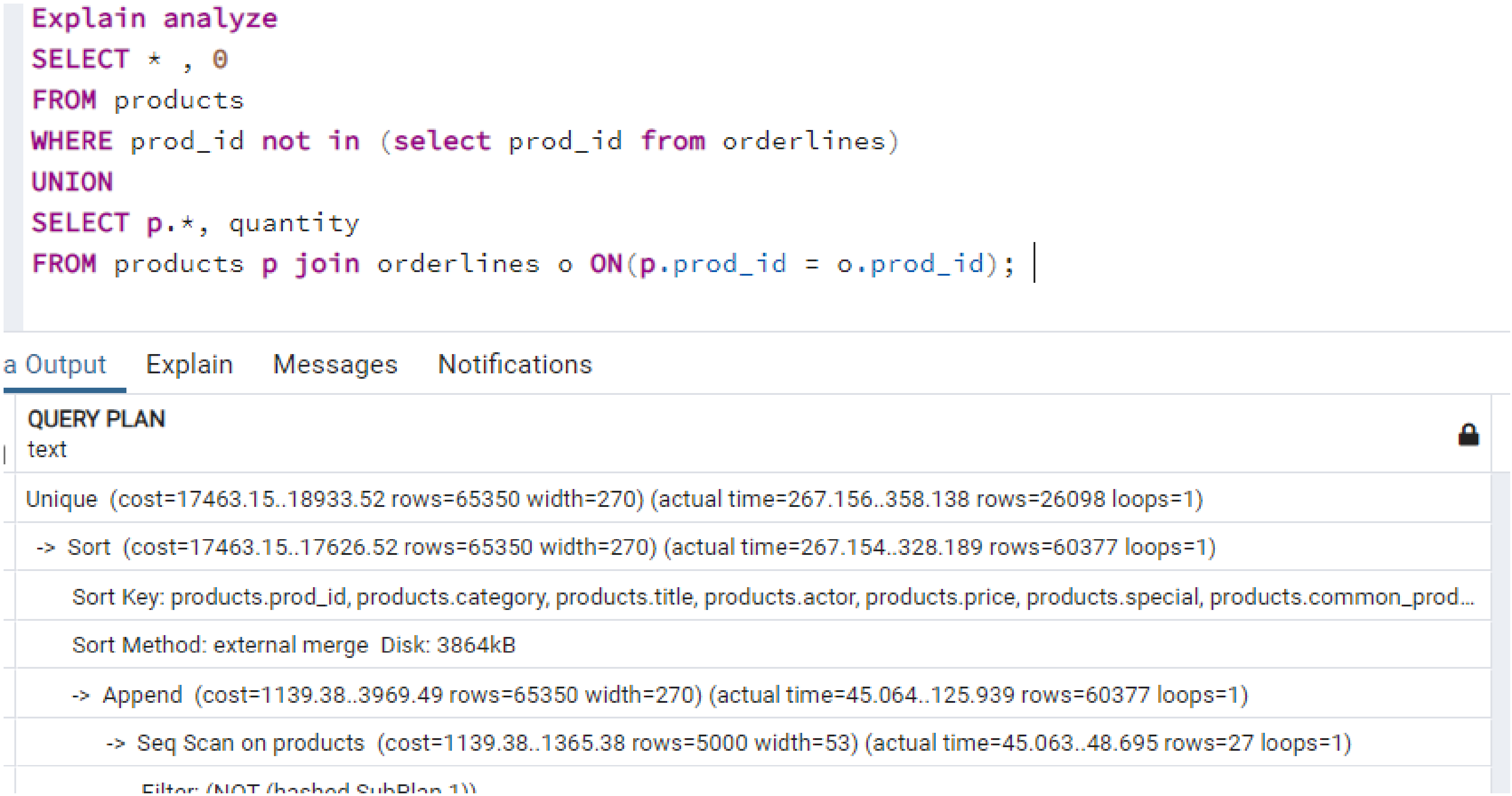
- Don’t Make Queries More Complex Than They Need To Be
  - OR / IN / UNION ?
  - OR operator : index is not used except composite index ➔ IN/UNION/OUTER JOIN

    ➔ Using a condition with IN or UNION:

```
SELECT * FROM orderlines
WHERE orderid = 1 OR orderid = 5000;
-- (first cost: 8 - total cost: 47).
Actual time = 50.82..50.83

SELECT * FROM orderlines
WHERE orderid IN (1,5000);
-- (0.29 - 30), actual time = 0.028..0.039

SELECT * FROM orderlines
WHERE orderid = 1
UNION
SELECT * FROM orderlines
WHERE orderid = 5000;
-- (30 - 31) – actual time: 0.053..0.056
```

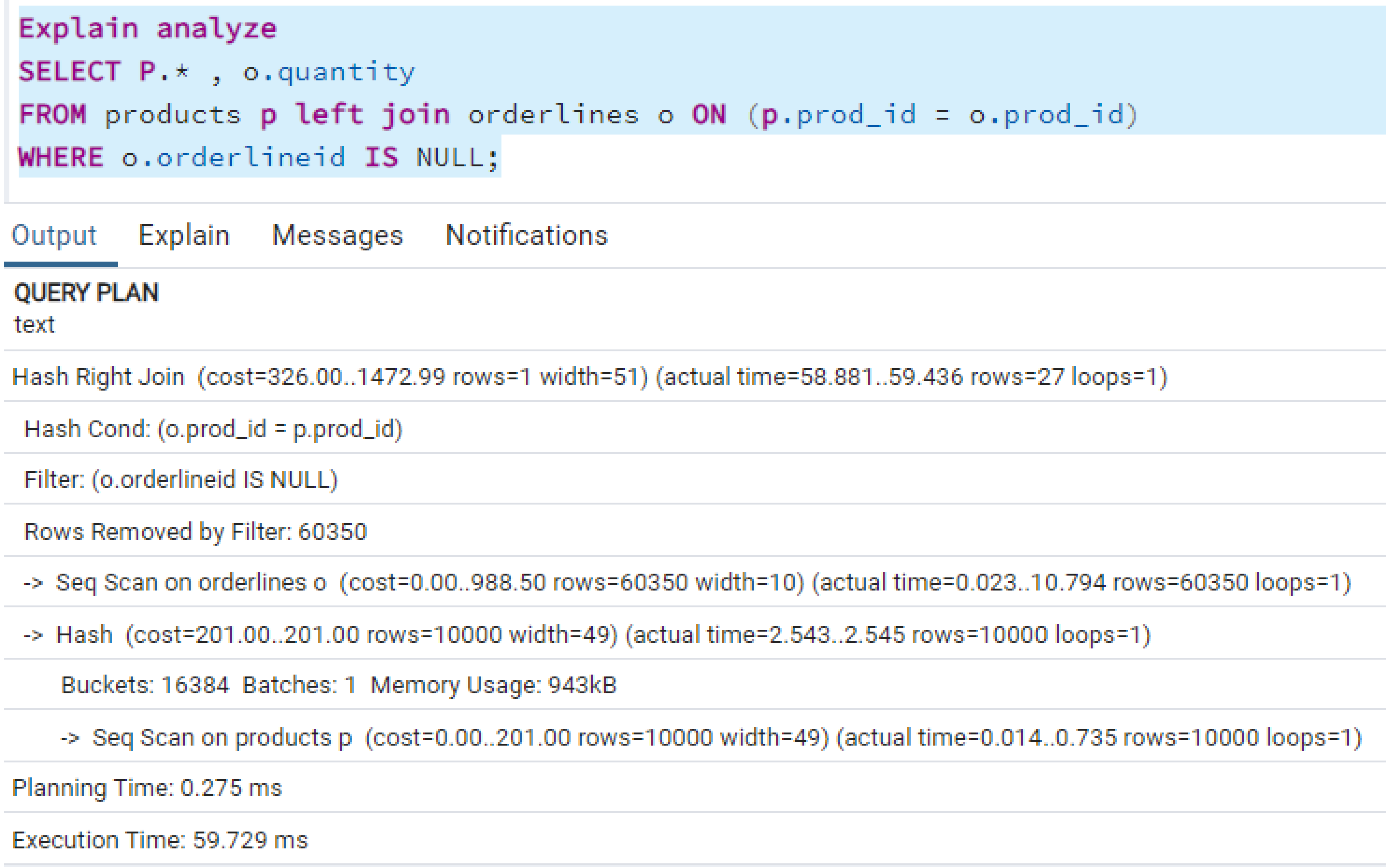
- Don’t Make Queries More Complex Than They Need To Be
  - To be careful not to unnecessarily use the UNION operation because you go through the same table multiple times ➔ use a UNION in your query, the execution time will increase.
  - Alternatives to the UNION operation are: reformulating the query in such a way that all conditions are placed in one SELECT instruction, or using an OUTER JOIN instead of UNION.

```sql
SELECT P.* , o.quantity
FROM products p left join orderlines o ON(p.prod_id =
o.prod_id) -- (326 - 2076), ~500ms
WHERE o.orderlineid IS NULL; -- (326 - 2076), 162ms

SELECT * , 0
FROM products
WHERE prod_id not in (select prod_id from orderlines)
UNION
SELECT p.*, quantity
FROM products p join orderlines o ON(p.prod_id = o.prod_id);
-- (17 780 - 19 210) 864 ms
```



```
Explain analyze
SELECT * , 0
FROM products
WHERE prod_id not in (select prod_id from orderlines)
UNION
SELECT p.*, quantity
FROM products p join orderlines o ON(p.prod_id = o.prod_id);
```

```
Unique  (cost=17463.15..18933.52 rows=65350 width=270) (actual time=267.156..358.138 rows=26098 loops=1)
  ->  Sort  (cost=17463.15..17626.52 rows=65350 width=270) (actual time=267.154..328.189 rows=60377 loops=1)
        Sort Key: products.prod_id, products.category, products.title, products.actor, products.price, products.special, products.common_prod...
        Sort Method: external merge  Disk: 3864kB
        ->  Append  (cost=1139.38..3969.49 rows=65350 width=270) (actual time=45.064..125.939 rows=60377 loops=1)
              ->  Seq Scan on products  (cost=1139.38..1365.38 rows=5000 width=53) (actual time=45.063..48.695 rows=27 loops=1)
                    Filter: (NOT (hashed SubPlan 1))
```



```sql
Explain analyze
SELECT P.* , o.quantity
FROM products p left join orderlines o ON (p.prod_id = o.prod_id)
WHERE o.orderlineid IS NULL;
```

```
QUERY PLAN
text
Hash Right Join  (cost=326.00..1472.99 rows=1 width=51) (actual time=58.881..59.436 rows=27 loops=1)
  Hash Cond: (o.prod_id = p.prod_id)
  Filter: (o.orderlineid IS NULL)
  Rows Removed by Filter: 60350
  ->  Seq Scan on orderlines o  (cost=0.00..988.50 rows=60350 width=10) (actual time=0.023..10.794 rows=60350 loops=1)
  ->  Hash  (cost=201.00..201.00 rows=10000 width=49) (actual time=2.543..2.545 rows=10000 loops=1)
        Buckets: 16384  Batches: 1  Memory Usage: 943kB
        ->  Seq Scan on products p  (cost=0.00..201.00 rows=10000 width=49) (actual time=0.014..0.735 rows=10000 loops=1)
Planning Time: 0.275 ms
Execution Time: 59.729 ms
```

- NOT operator: index is not used => avoid

```
select * from customers
where customerid != 5000;

select * from customers
where customerid = 5000;
```

- ANY / ALL: index not used => max , min ,…

- Isolate columns in Condition :

  age + 7 < 20 ➔ age < 13

- No Brute force
  - JOIN clause:
    - Order of tables => biggest table: placed last in join
    - No redundant conditions on joins
  - Having clause:
    - Used only if needed
    - Not to replace WHERE => WHERE help to limit the intermediate number of records

➔ Need smart indexing, smart using

Other index types

- Geometric type :
  - https://www.postgresql.org/docs/10/datatype-geometric.html
  - https://www.postgresql.org/docs/10/functions-geometry.html
- GiST: https://www.postgresql.org/docs/10/indexes-types.html