Database Tuning

https://soc-n.us/dbproject

Model

SQL Tuning II

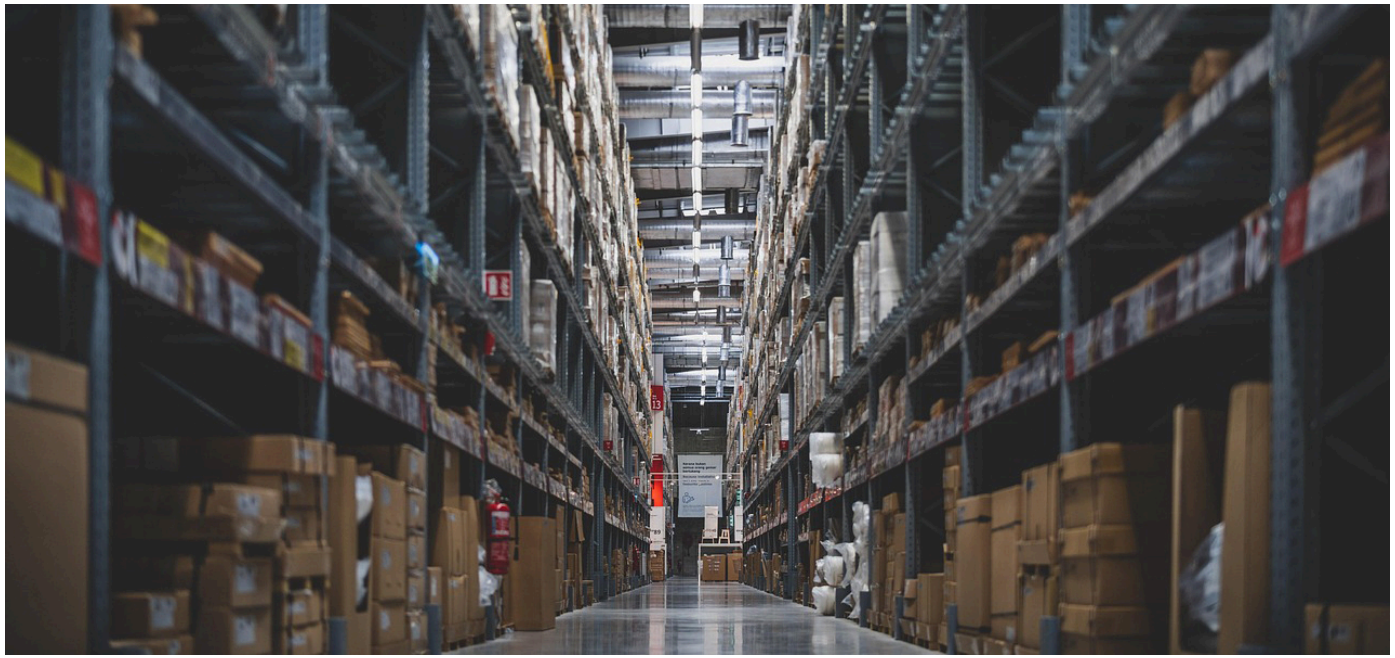
TPC-C

Partial Schema

The TPC-C Schema (partial)

A wholesale supplier operates out of several warehouses. The warehouse maintain stocks for the items sold by the company. We record the quantity in stock for each item available in each warehouse.

Warehouses have a unique identifier, a name, and a location. A location is given by a street, city, and country. Items have a unique identifier, a unique image identifier, a name, and a price.



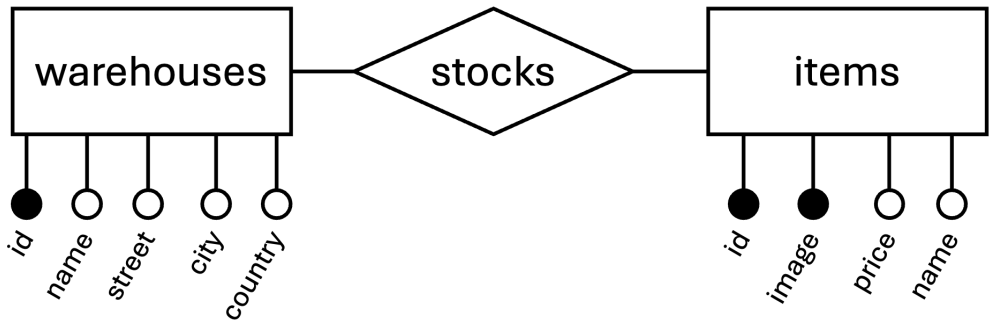
TPC-C

Entity-Relationship Diagram

The TPC-C Schema (partial)

A wholesale supplier operates out of several warehouses. The warehouse maintain stocks for the items sold by the company. We record the quantity in stock for each item available in each warehouse.

Warehouses have a unique identifier, a name, and a location. A location is given by a street, city, and country. Items have a unique identifier, a unique image identifier, a name, and a price.



Indexes

table_name	index_name	is_unique	is_primary	column_names
items	items_pkey	true	true	i_id
items	items_i_im_id_key	true	false	i_im_id
items	items_i_price	false	false	i_price
warehouses	warehouses_pkey	true	true	w_id
warehouses	warehouses_w_city	false	false	w_city
stocks	stocks_pkey	true	true	w_id
stocks	stocks_s_qty	false	false	s_qty

Nested Loop Join

Question

Find the names of the warehouses in Singapore that have stocks for item 33.

SQL

```
SELECT w.w_name
FROM warehouses w NATURAL JOIN stocks s
WHERE w.w_city = 'Singapore' AND s.i_id = 33;
```

Result

w_name
Crescent Oaks
Schemedeman
Namekagon
Briar Crest

4 rows

How to Join?

How do we join two tables? What is the efficient way to join?

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Nested Loop Join

Question

Find the names of the warehouses in Singapore that have stocks for item 33.

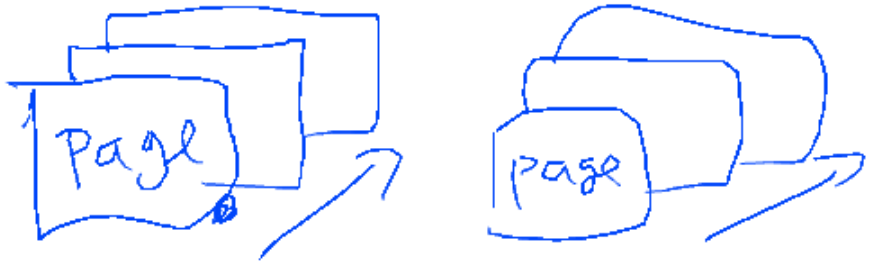
SQL

```
EXPLAIN SELECT w.w_name
FROM warehouses w NATURAL JOIN stocks s
WHERE w.w_city = 'Singapore' AND s.i_id = 33;
```

Query Plan
Nested Loop (cost=0.56..49.96 rows=2 width=7)
-> Index Scan using warehouses_w_city on warehouses w (cost=0.28..8.36 rows=5 width=11)
Index Cond: ((w_city)::text = 'Singapore'::text)
-> Index Only Scan using stocks_pkey on stocks s (cost=0.29..8.31 rows=1 width=4)
Index Cond: ((w_id = w.w_id) AND (i_id = 33))

Simplification?

Currently, the outer/inner tables are simpler due to index. Let us create a table without index.



Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

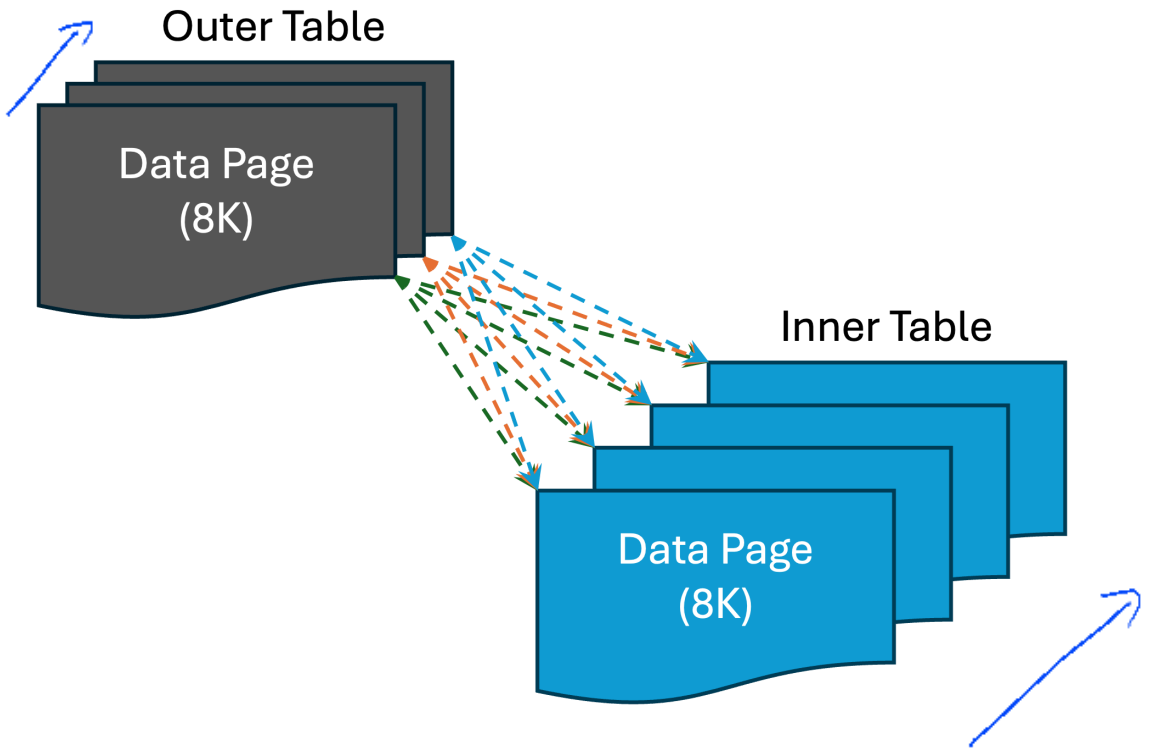
Nested Loop Join

Question

Find the names of the warehouses in Singapore that have stocks for item 33.

SQL

```
EXPLAIN SELECT w.w_name
FROM warehouses w NATURAL JOIN stocks s
WHERE w.w_city = 'Singapore' AND s.i_id = 33;
```



Simplification?

Currently, the outer/inner tables are simpler due to index. Let us create a table without index.

Temporary Table

Plain Table

We make temporary copies of **stocks** and **warehouses**. The copies do not have indexes. PostgreSQL does not create statistics for temporary tables unless told to do so.

Stocks

```
CREATE TEMPORARY TABLE stocks1 AS
  SELECT * FROM stocks;
```

Warehouses

```
CREATE TEMPORARY TABLE warehouses1 AS
  SELECT * FROM warehouses;
```

No Index

```
SELECT * FROM indexinfo
WHERE table_name IN ('warehouses1', 'stocks1');
```

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Nested Loop Join with Inner Sequential Scan

Question

Find the names of the warehouses in Singapore that have stocks for item 33.

SQL

```
EXPLAIN (ANALYZE) SELECT w.w_name
FROM warehouses1 w NATURAL JOIN stocks1 s
WHERE w.w_city = 'Singapore' AND s.i_id = 33;
```

PostgreSQL 16?

In PostgreSQL16, you may need to disable materialization, hash, and merge joins.

```
SET enable_material=off;
SET enable_hashjoin=off;
SET enable_mergejoin=off;
```

Query Plan
Nested Loop (cost=0.00..1909.85 rows=3 width=50) (actual time=7.172..33.408 rows=4 loops=1)
Join Filter: (w.w_id = s.w_id) Rows Removed by Join Filter: 1501
-> Seq Scan on warehouses1 w (cost=0.00..21.40 rows=2 width=54) (actual time=0.195..0.228 rows=5 loops=1)
Filter: ((w_city)::text = 'Singapore'::text) Rows Removed by Join Filter: 1000
-> Seq Scan on stocks1 s (cost=0.00..940.80 rows=274 width=4) (actual time=0.005..6.592 rows=301 loops=5)
Filter: (i_ud = 33) Rows Removed by Join Filter: 44611
Planning Time: 0.195 ms Execution Time: 33.450 ms

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Nested Loop Join with Inner Sequential Scan

Question

Find the names of the warehouses in Singapore that have stocks for item 33.

SQL

```
EXPLAIN (ANALYZE) SELECT w.w_name
FROM warehouses1 w NATURAL JOIN stocks1 s
WHERE w.w_city = 'Singapore' AND s.i_id = 33;
```

Outer Table?

The outer table is usually the **smallest** *(w.r.t. fit into memory)*.

Query Plan

```
Nested Loop (cost=0.00..1909.85 rows=3 width=50)
(actual time=7.172..33.408 rows=4 loops=1)
  Join Filter: (w.w_id = s.w_id)
  Rows Removed by Join Filter: 1501
  -> Seq Scan on warehouses1 w
     (cost=0.00..21.40 rows=2 width=54)
     (actual time=0.195..0.228 rows=5 loops=1)
     Filter: ((w_city)::text = 'Singapore'::text)
     Rows Removed by Join Filter: 1000
  -> Seq Scan on stocks1 s
     (cost=0.00..940.80 rows=274 width=4)
     (actual time=0.005..6.592 rows=301 loops=5)
     Filter: (i_ud = 33)
     Rows Removed by Join Filter: 44611
Planning Time: 0.195 ms
Execution Time: 33.450 ms
```

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Nested Loop Join with Materialized Inner Sequential Scan

Question

Find the names of the warehouses in Singapore that have stocks for item 33.

SQL

```
EXPLAIN (ANALYZE) SELECT w.w_name
FROM warehouses1 w NATURAL JOIN stocks1 s
WHERE w.w_city = 'Singapore' AND s.i_id = 33;
```

Materialize?

The inner table may be **materialized** if we enable it. The table order may also be reordered.

```
RESET enable_material;
```

Query Plan

```
Nested Loop (cost=0.00..970.43 rows=3 width=50)
(actual time=0.398..7.513 rows=4 loops=1)
  Join Filter: (w.w_id = s.w_id)
  Rows Removed by Join Filter: 1501
  -> Seq Scan on stocks1 s
     (cost=0.00..940.80 rows=274 width=4)
     (actual time=0.034..6.808 rows=301 loops=1)
                                   :
  -> Materialize (cost=0.00..21.41 rows=2 width=54)
     (actual time=0.001..0.001 rows=5 loops=301)
        -> Seq Scan on warehouses1 w
           (cost=0.00..21.40 rows=2 width=54)
           (actual time=0.232..0.285 rows=5 loops=1)
                                   :
Planning Time: 0.704 ms
 Execution Time: 5.130 ms
```

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Nested Loop Join with Inner (Bitmap) Index Scan

Question

Find the identifier of the items and their individual quantity in stock in warehouses called 'Agimba'.

SQL

```
SELECT s.i_id, s.s_qty
FROM warehouses w, stocks s
WHERE w.w_id = s.w_id
  AND w.w_name = 'Agimba';
```

Result

i_id	s_qty
2	5
33	7
103	14
298	14
:	:

444 rows

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Nested Loop Join with Inner (Bitmap) Index Scan

Question

Find the identifier of the items and their individual quantity in stock in warehouses called 'Agimba'.

SQL

```
EXPLAIN (ANALYZE) SELECT s.i_id, s.s_qty
FROM warehouses w, stocks s
WHERE w.w_id = s.w_id
  AND w.w_name = 'Agimba';
```

Query Plan

```
Nested Loop
  → Seq Scan on warehouses w
  → Bitmap Heap Scan on stocks s
    → Bitmap Index Scan on stocks_pkey
```

Index

If an index is available, then the optimizer may choose a **Nested Loop Join** with an inner **Index Scan**.

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

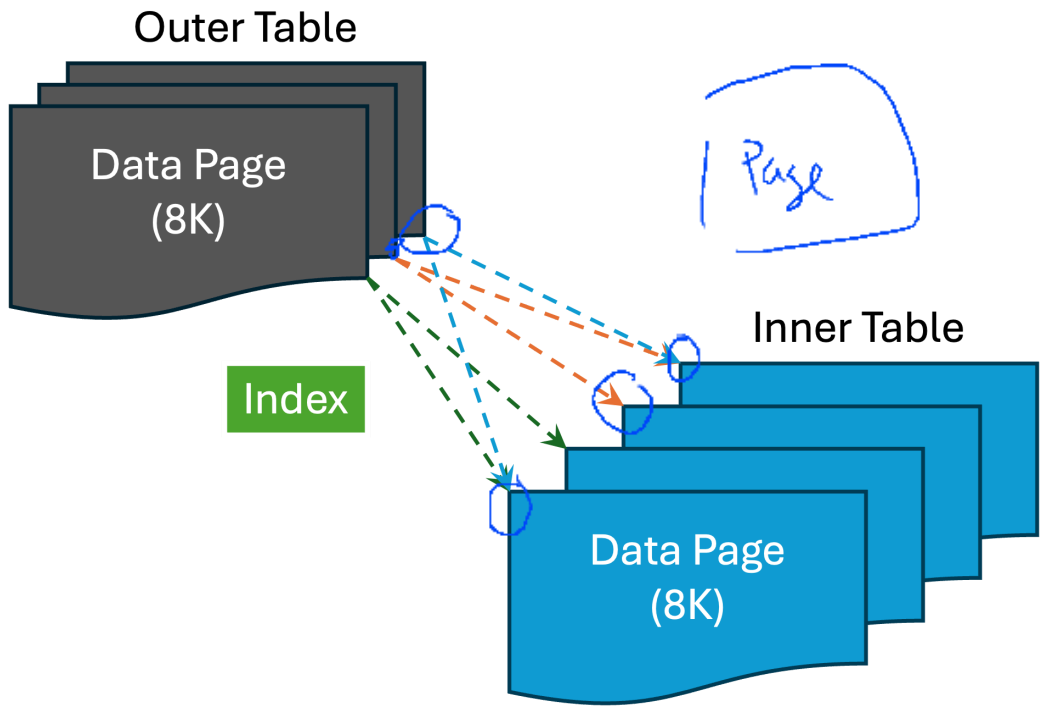
Nested Loop Join with Inner (Bitmap) Index Scan

Question

Find the identifier of the items and their individual quantity in stock in warehouses called 'Agimba'.

SQL

```
EXPLAIN (ANALYZE) SELECT s.i_id, s.s_qty
FROM warehouses w, stocks s
WHERE w.w_id = s.w_id
  AND w.w_name = 'Agimba';
```



Index

If an index is available, then the optimizer may choose a **Nested Loop Join** with an inner **Index Scan**.

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Nested Loop Join with Memoized Inner Index Scan

Question

Find the name of warehouse, item id, and quantity in stock in the warehouse for every item.

SQL

```
EXPLAIN (ANALYZE)
  SELECT w.w_name, s.i_id, s.s_qty
  FROM warehouses w, stocks s
  WHERE w.w_id = s.w_id;
```

Query Plan

```
Nested Loop
  → Seq Scan on stocks s
  → Memoize
    Cache Key: s.w_id
    Cache Mode: logical
    Hits: 43907 Misses: 1005
    Evictions: 0 Overflows: 0
    Memory Usage: 112kB
    → Index Scan using warehouses_pkey
      on warehouses w
```

Memoization

The scan to the underlying plans can be skipped when the results for the current parameters are already in the cache.

This was added in PostgreSQL 16.

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Hash Join

Question

Find the name of warehouse, item id, and quantity in stock in the warehouse for every item.

SQL

```
EXPLAIN (ANALYZE)
  SELECT w.w_name, s.i_id, s.s_qty
  FROM warehouses w, stocks s
  WHERE w.w_id = s.w_id;
```

Query Plan

```
Hash Join
Hash Cond: (s.w_id = w.w_id)
  → Seq Scan on stocks s
  → Hash
    Buckets: 1024 Batches: 1
    Memory Usage: 53kB
    → Seq Scan on warehouses w
```

Enable Hash

We may have to re-enable hash join.

```
RESET enable_hashjoin;
```

In general, the optimizer will choose Hash Join.

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

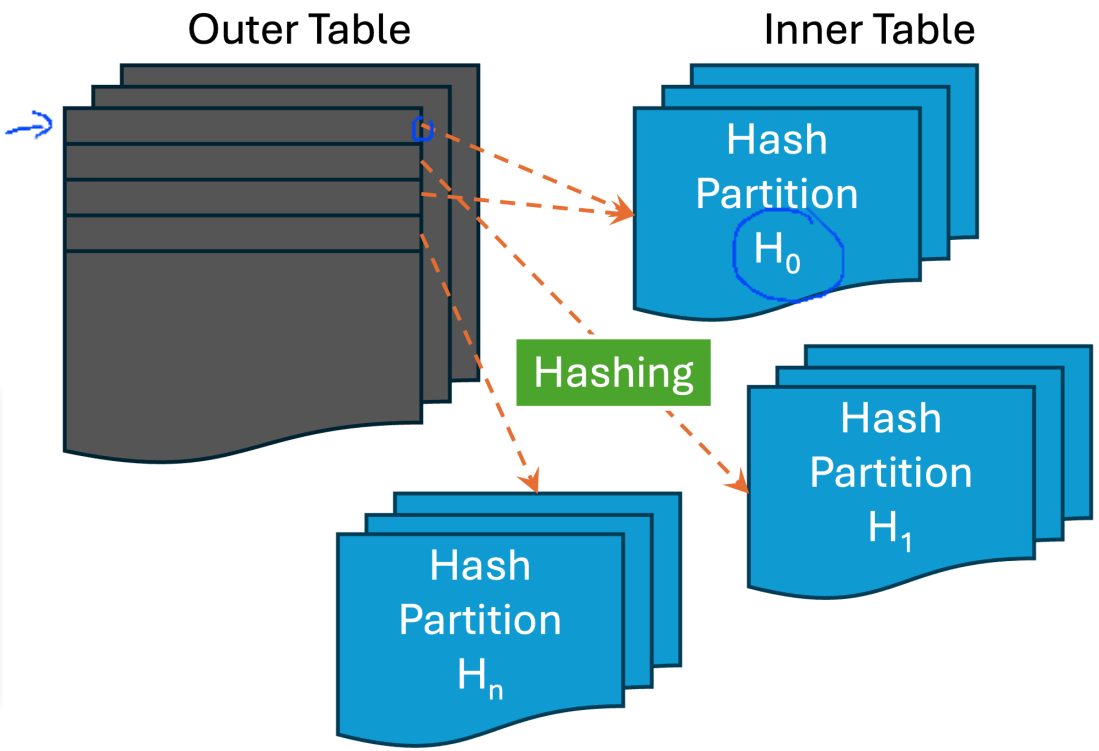
Hash Join

Question

Find the name of warehouse, item id, and quantity in stock in the warehouse for every item.

SQL

```
EXPLAIN (ANALYZE)
  SELECT w.w_name, s.i_id, s.s_qty
  FROM warehouses w, stocks s
  WHERE w.w_id = s.w_id;
```



Enable Hash

We may have to re-enable hash join.

```
RESET enable_hashjoin;
```

In general, the optimizer will choose Hash Join.

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Hash Join

Question

... this is just an illustrative example ...

SQL

```
EXPLAIN (ANALYZE) SELECT w1.w_name
FROM warehouses1 w1, warehouses1 w2
WHERE w1.w_name = w2.w_name;
```

Statistics

Let us check the statistics.

```
SELECT * FROM pg_statistic s, pg_class t
WHERE s.starelid = t.oid
  AND t.relname = 'warehouses1';
```

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Hash Join

Question

... this is just an illustrative example ...

SQL

```
EXPLAIN (ANALYZE) SELECT w1.w_name
FROM warehouses1 w1, warehouses1 w2
WHERE w1.w_name = w2.w_name;
```

Query Plan

```
Hash Join
Hash Cond:
((w1.w_name)::text = (w2.w_name)::text)
  → Seq Scan on warehouses1 w1
  → Hash
    Buckets: 1024 Batches: 1
    Memory Usage: 48kB
    → Seq Scan on warehouses1 w2
```

Statistics

In the absence statistics, PostgreSQL may also choose Hash Join.

This is true even if we re-enable merge join.

```
RESET enable_mergejoin;
```

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Merge Join

Question

... this is just an illustrative example ...

SQL

```
EXPLAIN (ANALYZE) SELECT w1.w_name
FROM warehouses1 w1, warehouses1 w2
WHERE w1.w_name = w2.w_name
ORDER BY w1.w_name;
```

Join then Sort

When a join is followed by sorting, PostgreSQL may combine both into a **Merge Join**.

Query Plan

```
Merge Join
Merge Cond:
((w1.w_name)::text = (w2.w_name)::text)
  → Sort
    Sort Key: w1.w_name
    Sort Method: quicksort
    Memory: 43kB
    → Seq Scan on warehouses1 w1

  → Sort
    Sort Key: w1.w_name
    Sort Method: quicksort
    Memory: 43kB
    → Seq Scan on warehouses1 w2
```

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

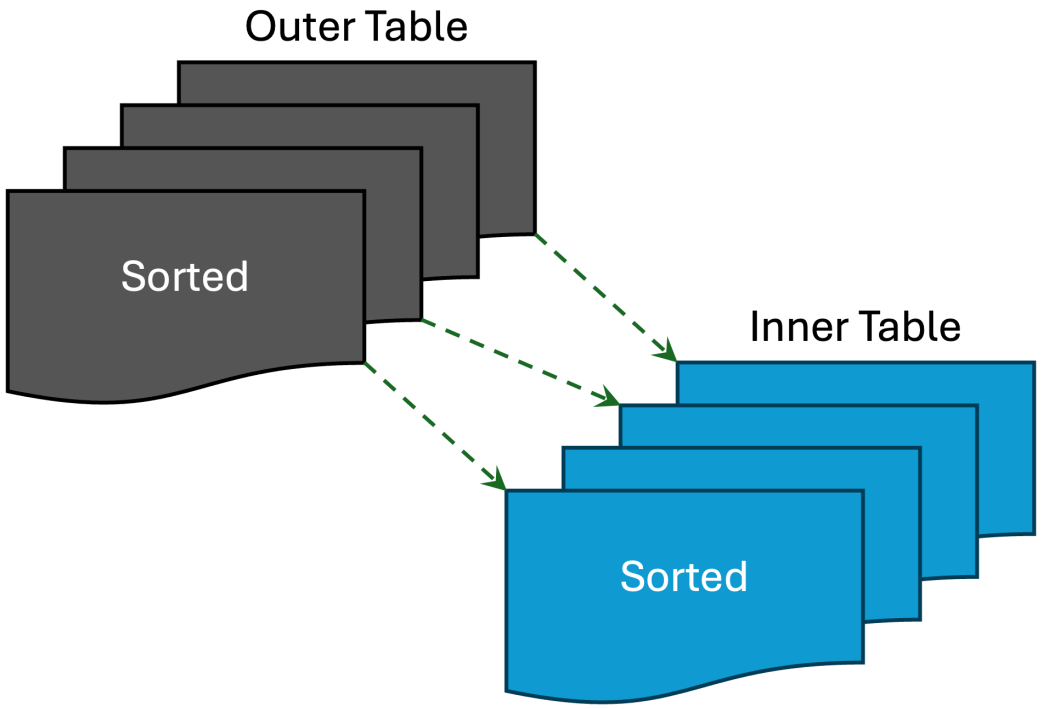
Merge Join

Question

... this is just an illustrative example ...

SQL

```
EXPLAIN (ANALYZE) SELECT w1.w_name
FROM warehouses1 w1, warehouses1 w2
WHERE w1.w_name = w2.w_name
ORDER BY w1.w_name;
```



Join then Sort

When a join is followed by sorting, PostgreSQL may combine both into a **Merge Join**.

↳ like a merge sort

Join 1 2 3 4 5 6 7 8 | Pages and Stats

Join on Ordered Table

Ordered Table

Even on an ordered table, a **Hash Join** may still be preferred over **Merge Join**.

SQL

```
EXPLAIN (ANALYZE) SELECT w1.w_name
FROM warehouses2 w1, warehouses2 w2
WHERE w1.w_name = w2.w_name;
```

```
CREATE TABLE warehouses2 AS
  SELECT * FROM warehouses
  ORDER BY w_name;
```

Query Plan

```
 Hash Join
Hash Cond:
((w1.w_name)::text = (w2.w_name)::text)
  → Seq Scan on warehouses2 w1
  → Hash
    Buckets: 1024 Batches: 1
    Memory Usage: 48kB
    → Seq Scan on warehouses2 w2
```

Join ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ | Pages and Stats

Merge Join with Index

Index

If there is an available index, then PostgreSQL may switch to Merge Join.

SQL

```
EXPLAIN (ANALYZE) SELECT w1.w_name
FROM warehouses2 w1, warehouses2 w2
WHERE w1.w_name = w2.w_name;
```

```
CREATE INDEX w2_w_name ON warehouses2(w_name);
```

Query Plan

```
Merge Join
Merge Cond:
((w1.w_name)::text = (w2.w_name)::text)
  → Sort
    Sort Key: w1.w_name
    Sort Method: quicksort
    Memory: 43kB
    → Seq Scan on warehouses2 w1

  → Sort
    Sort Key: w1.w_name
    Sort Method: quicksort
    Memory: 43kB
    → Seq Scan on warehouses2 w2
```

Join | Pages and Stats

Vacuum

Vacuum and Vacuum Full

The commands VACUUM and VACUUM FULL recover or reuse disk space occupied by *updated* or *deleted* rows, update data statistics used by the PostgreSQL query planner, identify opportunities for index-only scans, and protect against loss of very old data.

```
VACUUM;
VACUUM FULL;
VACUUM warehouses2;
VACUUM FULL warehouses2;
```

Warning

Each of the operation above cannot be executed on a transaction.

```
ANALYZE;
ANALYZE warehouses2;
VACUUM FULL ANALYZE;
VACUUM ANALYZE warehouses2;
```

Analyze

ANALYZE gathers and updates statistics used by the PostgreSQL query planner.

PostgreSQL also has auto-vacuum daemon that can issue VACUUM and ANALYZE commands *adaptively*.

*The reason for issues with update and delete is Multiversion Concurrency Control *(MVCC)*. This allows multiple transactions to read/write data simultaneously without blocking each other. This is done by maintaining multiple versions of the data.

Pratik : 2, 8, 9

Biswadeep : 4, 6, 11

Puru : 1, 10, 7

Adi : 3, 5, 12, 13

Break

Back by 20:18

$$\delta[c](R_1 \times R_2) \equiv R_1 \bowtie_{[c]} R_2$$

$$R_1 \times R_2 \equiv R_1 \bowtie_{[T]} R_2$$

Semi ● ○ ○ ○ | Anti | Join Order

Semi-Join

Question

Find the name of the warehouses with at least one item.

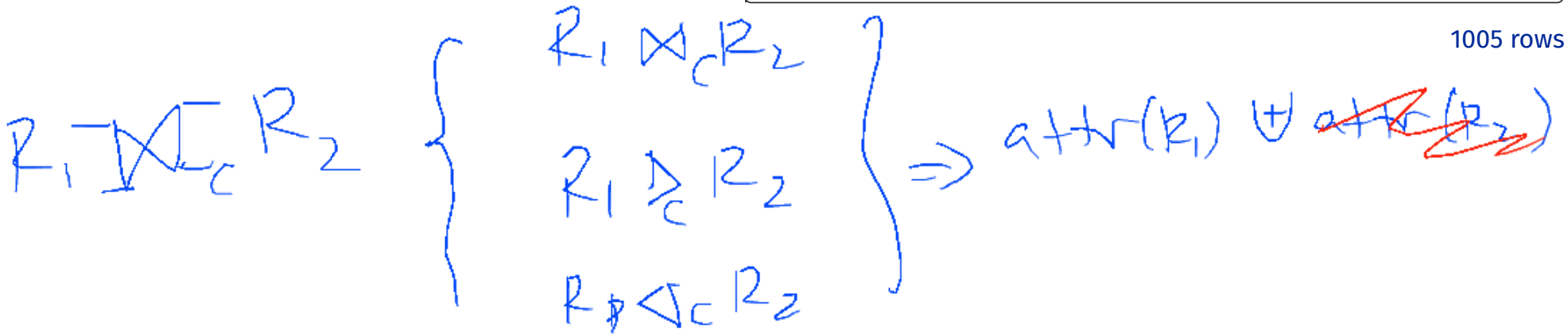
Query

```
SELECT w.w_name FROM warehouses w
WHERE EXISTS (
  SELECT * FROM stocks s
  WHERE s.w_id = w.w_id
);
```

Result

w_name
Linktype
Photolist
Thoughtlist
Browsedrive
:

1005 rows



Semi

1 2 4 5 | Anti | Join Order

Semi-Join

Question

Find the name of the warehouses with at least one item.

Query

```
EXPLAIN (ANALYZE)
  SELECT w.w_name FROM warehouses w
  WHERE EXISTS (
    SELECT * FROM stocks s
    WHERE s.w_id = w.w_id
  );
```

Query Plan

```
Nested Loop Semi Join
  → Seq Scan on warehouses w
  → Index Only Scan using stocks_pkey
    on stocks s
    Index Cond: (w_id = w.w_id)
    Heap Fetches: 0
```

Semi-Join

When the inner table is only used for including results, PostgreSQL can use Semi-Join.

Semi

1 2 4 5 | Anti | Join Order

Semi-Join

Question

Find the name of the warehouses with at least one item.

Query

```
EXPLAIN (ANALYZE)
  SELECT w.w_name FROM warehouses w
  WHERE EXISTS (
    SELECT * FROM stocks s
    WHERE s.w_id = w.w_id
  );
```

Query Plan

```
Nested Loop Semi Join
  → Seq Scan on warehouses w
  → Index Only Scan using stocks_pkey
    on stocks s
    Index Cond: (w_id = w.w_id)
    Heap Fetches: 0
```

Follow Up Question

What are other ways to perform a Semi-Join?

Semi ❶ ❷ ❸ ❹ | Anti | Join Order

Semi-Join

Question

Find the name of the warehouses with at least one item.

Query

```
EXPLAIN (ANALYZE)
  SELECT w.w_name FROM warehouses w
  WHERE w.w_id IN (
    SELECT s.w_id FROM stocks s
  );
```

Query Plan

```
Nested Loop Semi Join
  → Seq Scan on warehouses w
  → Index Only Scan using stocks_pkey
    on stocks s
    Index Cond: (w_id = w.w_id)
    Heap Fetches: 0
```

Semi ❶ ❷ ❸ ❹ ❺ | Anti | Join Order

Semi-Join

Question

Find the name of the warehouses with at least one item.

Query

```
EXPLAIN (ANALYZE)
  SELECT w.w_name FROM warehouses w
  WHERE w.w_id = ANY (
    SELECT s.w_id FROM stocks s
  );
```

Query Plan

```
Nested Loop Semi Join
  → Seq Scan on warehouses w
  → Index Only Scan using stocks_pkey
    on stocks s
    Index Cond: (w_id = w.w_id)
    Heap Fetches: 0
```

Anti (Semi) Join

Question

Find the name of the warehouses without any item.

Query

```
EXPLAIN (ANALYZE)
  SELECT w.w_name FROM warehouses w
  WHERE NOT EXISTS (
    SELECT * FROM stocks s
    WHERE s.w_id = w.w_id
  );
```

Query Plan

```
Nested Loop Anti Join
  → Seq Scan on warehouses w
  → Index Only Scan using stocks_pkey
    on stocks s
    Index Cond: (w_id = w.w_id)
    Heap Fetches: 0
```

Anti-Join

When the inner table is only used for excluding results, PostgreSQL can use **Anti-Join**.

We call this **Anti-Semi-Join** *(i.e., opposite of Semi-Join)*.

Anti (Semi) Join

Question

Find the name of the warehouses without any item.

Query

```
EXPLAIN (ANALYZE)
  SELECT w.w_name FROM warehouses w
  WHERE NOT EXISTS (
    SELECT * FROM stocks s
    WHERE s.w_id = w.w_id
  );
```

Query Plan

```
Nested Loop Anti Join
  → Seq Scan on warehouses w
  → Index Only Scan using stocks_pkey
    on stocks s
    Index Cond: (w_id = w.w_id)
    Heap Fetches: 0
```

Follow Up Question

What are other ways to perform an **Anti-Join**?

Anti (Semi) Join

Question

Find the name of the warehouses without any item.

Query

```
EXPLAIN (ANALYZE)
  SELECT w.w_name
  FROM warehouses w
    LEFT JOIN stocks s
    ON w.w_id = s.w_id
  WHERE s.w_id IS NULL;
```

Query Plan

```
Nested Loop Anti Join
  → Seq Scan on warehouses w
  → Index Only Scan using stocks_pkey
    on stocks s
    Index Cond: (w_id = w.w_id)
    Heap Fetches: 0
```

Follow Up Question

What are other ways to perform an **Anti-Join**?

Anti (Semi) Join

Question

Find the name of the warehouses without any item.

Query

```
EXPLAIN (ANALYZE)
  SELECT w.w_name FROM warehouses w
  WHERE w.w_id NOT IN (
    SELECT s.w_id FROM stocks s
  );
```

Query Plan

```
 Seq Scan on warehouses w
Filter: (NOT (hashed SubPlan 1))
  SubPlan 1
  → Seq Scan on stocks s
```

Warning

PostgreSQL does not do well with `NOT IN`.

Anti (Semi) Join

Question

Find the name of the warehouses without any item.

Query

```
EXPLAIN (ANALYZE)
  SELECT w.w_name FROM warehouses w
  WHERE w.w_id <> ALL (
    SELECT s.w_id FROM stocks s
  );
```

Query Plan

```
 Seq Scan on warehouses w
Filter: (SubPlan 1)
  SubPlan 1
  → Materialze
    → Seq Scan on stocks s
```

Warning

PostgreSQL does not do well with **ALL**.

Semi | Anti | Join Order ❶ ❷ ❸ ❹ ❺ ❻

Outer Join

Question

Find the warehouses with a stock of Meclizine Hydrochloride.

Query

```
EXPLAIN (ANALYZE) SELECT i.i_name, s.w_id
FROM items i LEFT JOIN stocks s
  ON s.i_id = i.i_id
WHERE i.i_name = 'MECLIZIE HYDROCHLORIDE';
```

Query Plan

```
Hash Right Join
Hash Cond: (s.i_id = i.i_id)
  → Seq Scan on stocks s
  → Hash
    → Seq Scan on items i
```

Left/Right

PostgreSQL implements *(Left/Right)* **Outer Join**.

PostgreSQL may replace **Left Join** with **Right Join**.

Semi | Anti | Join Order ❶ ❷ ❸ ❹ ❺ ❻

Cross Join

Question

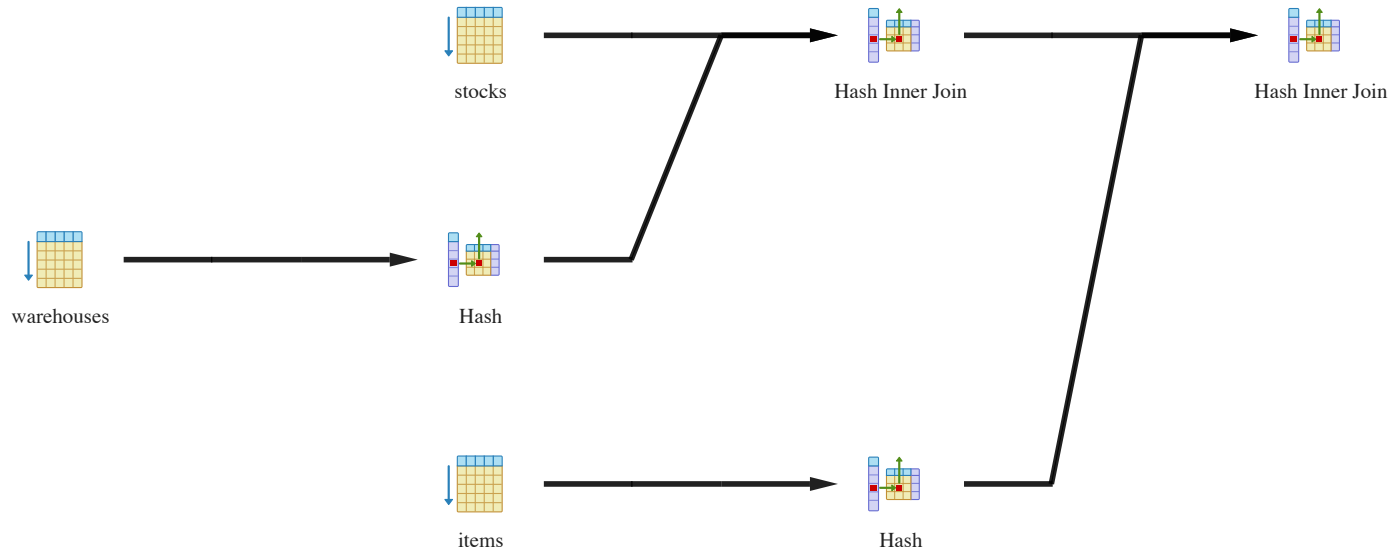
Find warehouse, item, and quantity in the database.

Query

```
EXPLAIN (ANALYZE) SELECT w.w_name, i.i_name, s.s_qty
FROM warehouses w, stocks s, items i WHERE w.w_id = s.w_id AND i.i_id = s.i_id;
```

Join Order

The optimizer chooses one amongst several possible join order.



Semi | Anti | Join Order ❶ ❷ ❸ ❹ ❺ ❻

Cross Join

Question

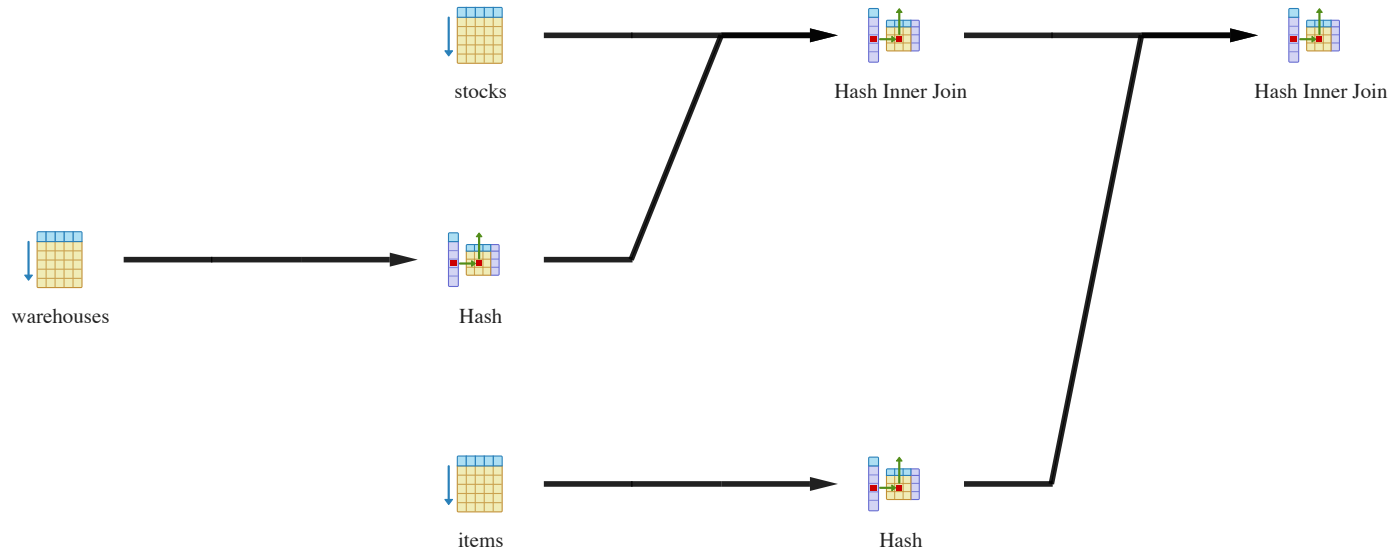
Find warehouse, item, and quantity in the database.

Query

```
EXPLAIN (ANALYZE) SELECT w.w_name, i.i_name, s.s_qty
FROM stocks s, warehouses w, items i WHERE w.w_id = s.w_id AND i.i_id = s.i_id;
```

Join Order

It will try to find an optimized query plan. Hence, reordering typically does not change the query plan.



Semi | Anti | Join Order ❶ ❷ ❸ ❹ ❺ ❻

Cross Join

Question

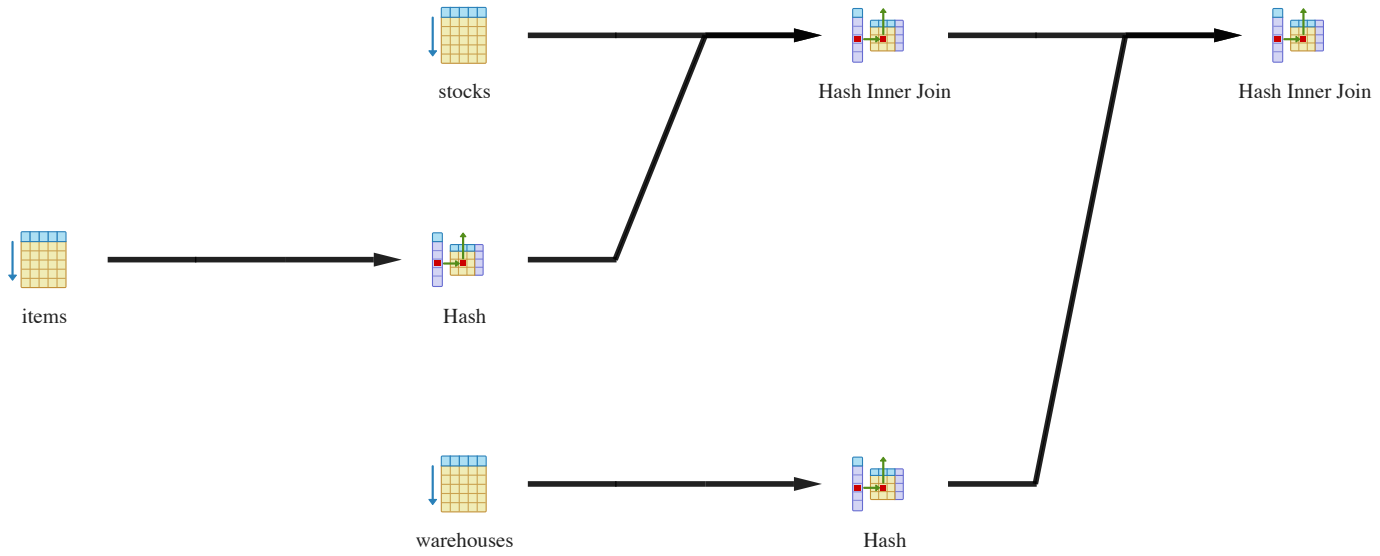
Find warehouse, item, and quantity in the database.

Query

```
EXPLAIN (ANALYZE) SELECT w.w_name, i.i_name, s.s_qty
FROM items i, stocks s, warehouses w WHERE w.w_id = s.w_id AND i.i_id = s.i_id;
```

Join Order

In some cases, it can be *(indirectly)* be forced to choose one preferred join orders.



Semi | Anti | Join Order ❶ ❷ ❸ ❹ ❺ ❻

Natural Join

Question

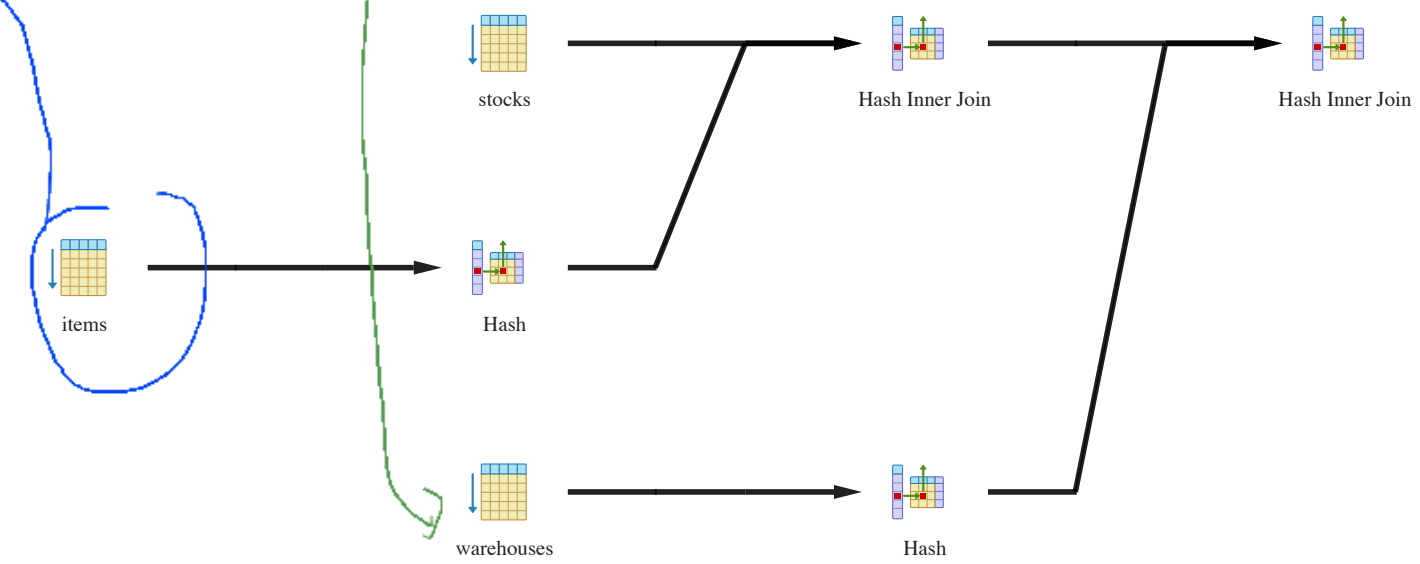
Find warehouse, item, and quantity in the database.

Query

```
EXPLAIN (ANALYZE) SELECT w.w_name, i.i_name, s.s_qty
FROM items i NATURAL JOIN stocks s NATURAL JOIN warehouses w;
```

Natural Join Order

Natural Join order can also be *(indirectly)* coerced.



Semi | Anti | Join Order ❶ ❷ ❸ ❹ ❺ ❻

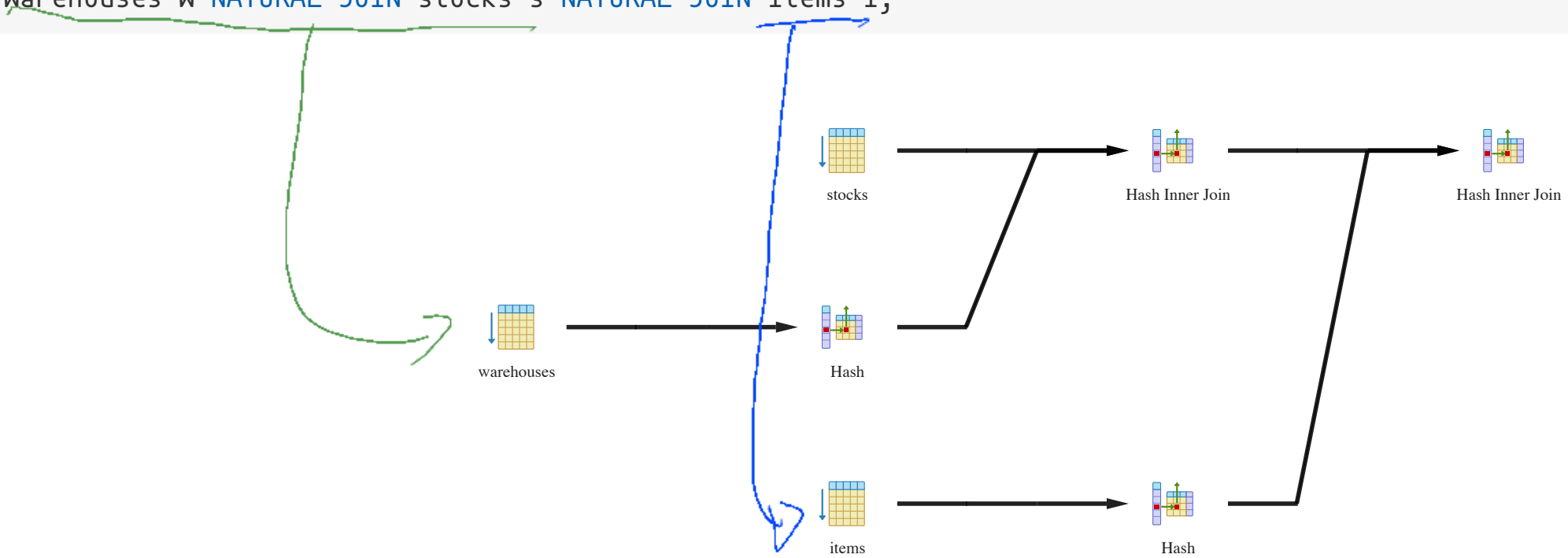
Natural Join

Question

Find warehouse, item, and quantity in the database.

Query

```
EXPLAIN (ANALYZE) SELECT w.w_name, i.i_name, s.s_qty
FROM warehouses w NATURAL JOIN stocks s NATURAL JOIN items i;
```



Normalized Schema

Effect of Normalization

A **normalized schema** requires us to join tables on the ***equality*** of their primary and foreign keys.

Joins can be expensive.

Denormalized Schema

We can **denormalize** the schema by joining back some tables together.

Insertions, deletions, and updates are **more complicated** and they are risky since some constraints cannot be maintained.

They are more costly because of manual propagation *(e.g., using triggers)*. Some but not all queries are typically faster.

Database Views

Views

We can create views, but they are for convenience only. They do not change the performance from that of the underlying normalized schema.

The `VIEW` definition is used by the optimizer as would a **subquery**.

Materialized Views

We can create materialized views. They are **middle ground** between a normalized and a denormalized schema.

Insertions, deletions, and updates are **more costly** *(currently manual with triggers and refresh)*. Postgres may not use the materialized view definition to optimize the query.

We can refresh a materialized views using the following command.

```
REFRESH MATERIALIZED VIEW mvall;
```

Materialized views can also be **indexed**.

Arguably Good or Bad Things to Do

PREPARE

When the PREPARE statement is executed, the specified query is

- parsed,
- analyzed, and
- rewritten.

When an EXECUTE command is subsequently inssued, the prepared query is

- planned, and
- executed.

Potential Issues?

If a prepared statement is executed enough times, the server may eventually decide to save and reuse a generic plan rather than re-planning each time.

Arguably Good or Bad Things to Do

Prepare

```
PREPARE q AS
  SELECT s.i_id
  FROM stocks s
  WHERE s.s_qty > 500;
```

```
EXPLAIN (ANALYZE) EXECUTE q;
```

Query Plan

```
Index Scan using stocks_s_qty
on stocks s
Index Cond: (s_qty > 500)
```

Deallocate

```
DEALLOCATE q;
```

Arguably Good or Bad Things to Do

Planner Method Configuration

It is not recommended to configure the optimizer by turning off some methods.

See *PostgreSQL documentation* for more information.

```
SET enable_seqscan=off;
SET enable_bitmapscan=off;
SET enable_hashjoin=off;
-- etc
```

on true
off false
RESET

Arguably Good or Bad Things to Do

Hints

Several systems *(e.g., MariaDB)* allows the designer and the programmer to give hints to the optimizer.

Potential Issues?

It is not recommended to use hints unless you are confident that the statistics will never change. You also need to be confident that the plan the optimizer find with your hints will always be the optimal plan.

MariaDB

```sql
SELECT
  /*+ INDEX (items_i_price) */
  i.i_name
FROM warehouses w, stocks s, items i
WHERE w.w_id = s.w_id
  AND i.i_id = s.i_id
  AND i.i_price < 100;
```

```sql
SELECT
  /*+ NO_INDEX (items_i_price) */
  i.i_name
FROM warehouses w, stocks s, items i
WHERE w.w_id = s.w_id
  AND i.i_id = s.i_id
  AND i.i_price < 100;
```

*May be available in PostgreSQL with extension.

Arguably Good or Bad Things to Do

Hints

Several systems *(e.g., MariaDB)* allows the designer and the programmer to give hints to the optimizer.

Potential Issues?

It is not recommended to use hints unless you are confident that the statistics will never change. You also need to be confident that the plan the optimizer find with your hints will always be the optimal plan.

Wise Words

> *... Forcing postgres to do it your way usually means you've done it wrong. 9/10 Times the planner will beat anything you can come up with. The other 1 time its because you made it wrong.*

Kent Fredric at *StackOverflow*
Nov 21, 2008 at 19:31

What to Keep in Mind

Why are Queries Slow?

- Wrong design *(normalized vs denormalized)* ;
- Poor configuration *(increase `work_mem`)* ;
- Tuples are scattered, tables, and indexes are **bloated** *(`VACUUM`, `CLUSTER`, `VACUUM FULL`, reindexing, etc)* ;
- Missing indexes *(`CREATE INDEX`)* ;
- PostgreSQL does not choose the best plan *(`ANALYZE`)*.

In Conclusion

- Understand the optimizer ;
- Tune the data *(normalize, denormalize, index, views, materialized, etc)* for everyone ;
- Help the system maintain good statistics ;
- Hard-tune the queries as a **last resort** and at every users' *(current and future)* risk.

Acknowledgement

Materials

The material in this lecture is adapted from the lecture by Prof. Stéphane Bressan.

In turn, the material was adapted from Bruce Momjian's online tutorials

- on "Explaining PostgreSQL Query Optimizer", *PDF*
- on "PostgreSQL Internals Through Pictures", *PDF*

Additional information can be found on *PostgreSQL online documentation* as well as the book "Database Tuning" by Dennis Shasha and Philippe Bonnet.

```
postgres=# exit
Press any key to continue . . .
```