CS4221
Relational Databases II. Turning Strategies B

Yao LU

2024 Semester 2

National University of Singapore
School of Computing

Agenda

- Index scans
- Multicolumn index
- Joins
- Denormalisation
- Views
- Materialized views

Recall from last lecture

- Queries (TPCC)
- Index scans, bitmap heap scans

When does it happen?

- Query

Find the items and warehouses where the item quantity is below 100.

```
1 SELECT *
2 FROM stocks s
3 WHERE s.s_qty < 100;
```

When does it happen?

- Sequential Scan

Find the items and warehouses where the item quantity is below 100.

```sql
EXPLAIN ANALYZE SELECT *
FROM stock s
WHERE s.s_qty < 100;
```

Query Plan
Seq Scan on stock s (cost=0.00..804.40 rows=6706 width=10) (actual time=0.022..19.820 rows=6750 loops=1)
Filter: (s_qty < 100)
Rows Removed by Filter: 38162
Planning time: 0.347 ms
Execution time: 20.608 ms

When does it happen?

- Creating an Index

Create a B+Tree index (default) on the s_qty attribute of stock.

```
1 CREATE INDEX i_s_qty ON stocks(s_qty)
;
```

When does it happen?

- Bitmap Heap Scan

With an index and sufficient statistics the optimizer can choose using bitmaps constructed from the index.

```
1 EXPLAIN ANALYZE SELECT *
2 FROM stocks s
3 WHERE s.s_qty < 100;
```

When does it happen?

```sql
EXPLAIN ANALYZE SELECT *
FROM stock s
WHERE s.s_qty < 100;
```

Query Plan
Bitmap Heap Scan on stock s (cost=128.26..455.09 rows=6706 width=10) (actual time=1.992..3.667 rows=6750 loops=1)
Recheck Cond: (s_qty < 100)
Heap Blocks: exact=243
-> Bitmap Index Scan on i_s_qty (cost=0.00..126.58 rows=6706 width=0) (actual time=1.932..1.932 rows=6750 loops=1)
Index Cond: (s_qty < 100)
Planning time: 0.653 ms
Execution time: 4.081 ms

When does it happen?

- Sequential Scan

Even with an index and sufficient statistics the optimizer can choose a sequential scan.

```
1 EXPLAIN ANALYZE SELECT *
2 FROM stock s
3 WHERE s.s_qty >= 100;
```

Query Plan
Seq Scan on stock s (cost=0.00..804.40 rows=38206 width=10) (actual time=0.022..17.914 rows=38162 loops=1)
Filter: (s_qty >= 100)
Rows Removed by Filter: 6750
Planning time: 0.114 ms
Execution time: 21.138 ms

When does it happen?

- Index Scan

With an index and sufficient statistics the optimizer can choose using the index.

```
1 EXPLAIN ANALYZE SELECT *
2 FROM stock s
3 WHERE s.s_qty >= 10000;
```

Query Plan
Index Scan using i_s_qty on stock s (cost=0.29..8.26 rows=1 width=10) (actual time=0.043..0.089 rows=37 loops=1)
Index Cond: (s_qty >= 1000)
Planning time: 1.035 ms
Execution time: 0.119 ms

When does it happen?

- The Condition is not Selective

`s.s_qty` >= 100:38222 of 44912 rows (85%) are estimated to match the condition. The optimizer chooses a sequential scan.

- The Condition is Moderately Selective

`s.s_qty` < 100: 6690 of 44912 rows (15%) are estimated to match the condition. The optimizer chooses a bitmap heap scan.

- The Condition is very Selective

`s.s_qty` >= 100: 37 of 44912 rows (less than 0.001%) are estimated to match the condition. The optimizer chooses an index scan.

<u>These exact numbers will change in different PostgreSQL versions. You get the idea.</u>

When does it happen?

qty	stat	test
0	100	Seq Scan on stock
100	85	Seq Scan on stock
200	76	Seq Scan on stock
300	66	Seq Scan on stock
400	57	Seq Scan on stock
500	48	Seq Scan on stock
600	38	Bitmap Heap Scan on stock
700	28	Bitmap Heap Scan on stock
800	19	Bitmap Heap Scan on stock
900	10	Bitmap Heap Scan on stock
1000	0	Index Scan using i_s_qty on stock

Agenda

- Index scans
- Multicolumn index
- Joins
- Denormalisation
- Views
- Materialized views

Multicolumn Index

- Multicolumn Index

A multicolumn index can be used for index scan.

```
1 EXPLAIN SELECT s.s_qty
2 FROM stocks s
3 WHERE s.w_id='123' AND s.i_id='7';
```

Query Plan
Index Scan using stock_pkey on public.stock as s (cost=0.29..8.31 rows=1 width=2)
Index Cond: ((s.w_id = 123) AND (s.i_id = 7))

Multicolumn Index

- Multicolumn Index

A multicolumn index can be used for index scan even with a partial condition.

```
1 EXPLAIN SELECT s.s_qty
2 FROM stocks s
3 WHERE s.w_id='123';
```

Query Plan
Index Scan using stock_pkey on stock s (cost=0.29..74.95 rows=35 width=2)
Index Cond: (w_id = 123)

Multicolumn Index

- Multicolumn Index

This only works if the condition involves the prefix of the multicolumn index (attributes from left to right).

```
1 EXPLAIN SELECT s.s_qty
2 FROM stock s
3 WHERE s.i_id='7';
```

Query Plan
Seq Scan on stock s (cost=0.00..804.40 rows=90 width=2)
Filter: (i_id = 7)

Multicolumn Index

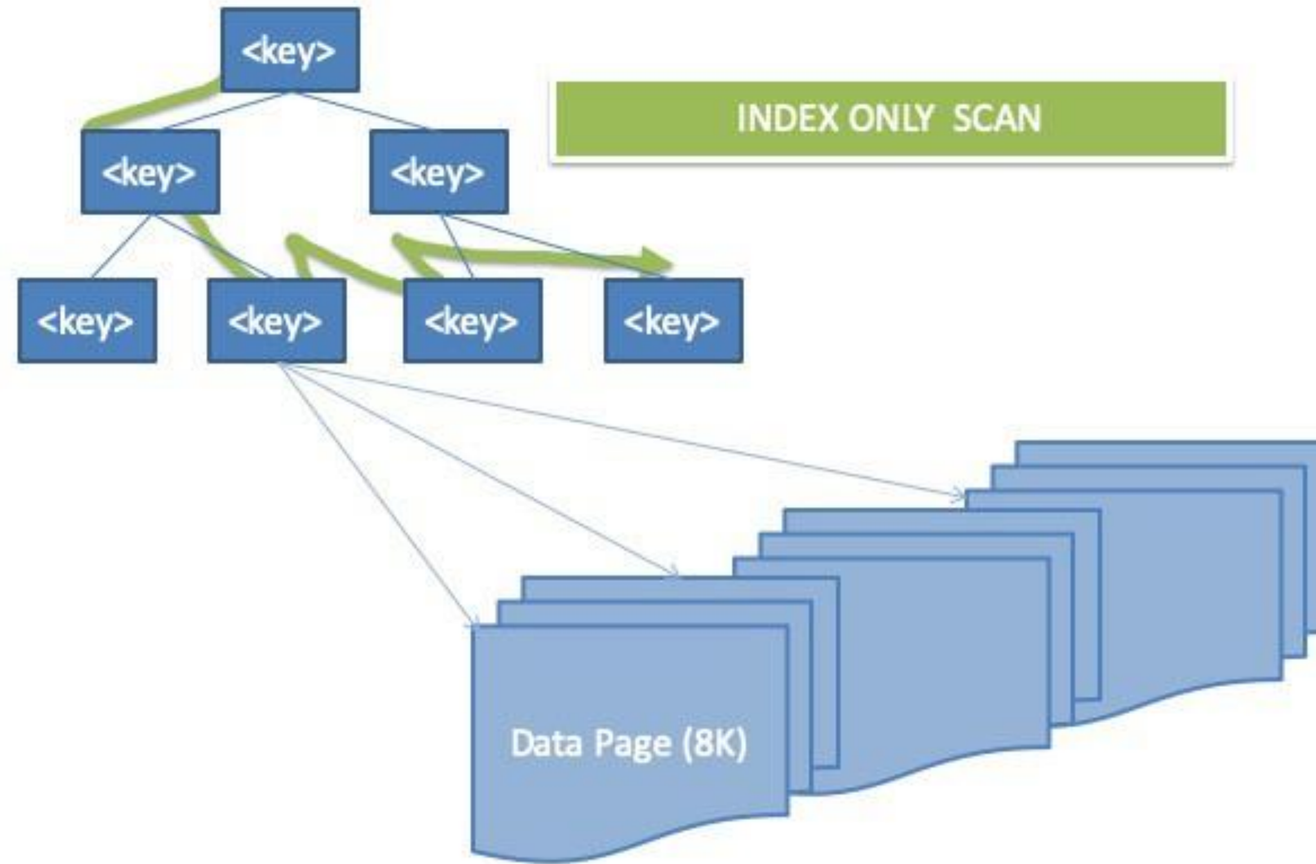
- Index Only Scan

Depending on the query, the scan can be done within the index only, without accessing the data.

```
1 EXPLAIN SELECT s.i_id
2 FROM stocks s
3 WHERE s.w_id='123';
```

Query Plan
Index Only Scan using stock_pkey on stock s (cost=0.29..74.95 rows=35 width=4)
Index Cond: (w_id = 123)

Multicolumn Index



Multicolumn Index

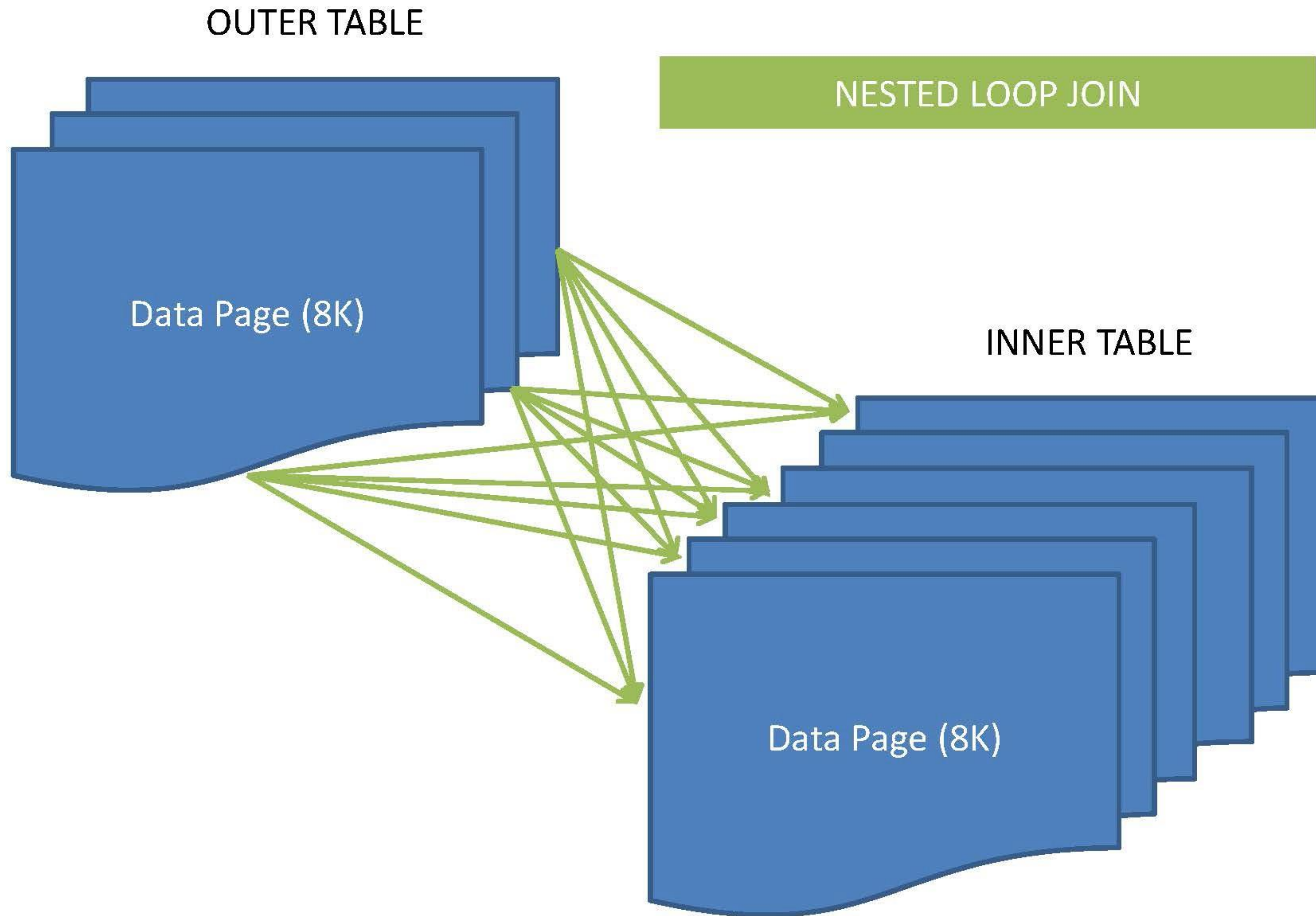
- Index Only Scan

Again, for a multicolumn index, it depends on the prefix.

```
1 EXPLAIN SELECT s.w_id
2 FROM stocks s
3 WHERE s.i_id='7';
```

Query Plan
Seq Scan on stock s (cost=0.00..804.40 rows=90 width=4)
Filter: (i_id = 7)

Nested Loop



Agenda

- Index scans
- Multicolumn index
- Joins
- Denormalisation
- Views
- Materialized views

Nested Loop

- Nested Loop Join With Inner Sequential Scan

We make temporary copies of `stock` and warehouse. The copies do not have indexes. PostgreSQL does not create statistics for temporary tables unless told to do so.

```
1 CREATE  TEMPORARY TABLE stock1 AS
2 SELECT * FROM stocks s;
3
4 CREATE  TEMPORARY TABLE warehouse1 AS
5 SELECT * FROM warehouses w;
```

Nested Loop

- Query

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

```
1 SELECT w.w_name
2 FROM   warehouse1 w NATURAL JOIN stock1 s
3 WHERE w.w_city='Singapore' AND s.i_id=33;
```

w_name
"Schmedeman"
"Crescent Oaks"
"Namekagon"
"Briar Crest"

Nested Loop

- Nested Loop Join with Inner sequential Scan

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

```sql
EXPLAIN ANALYZE SELECT w.w_name
FROM   warehouse1 w NATURAL JOIN stock1 s
WHERE w.w_city='Singapore' AND s.i_id=33;
```

Nested Loop

Query Plan
Nested Loop (cost=0.00..908.28 rows=1 width=118) (actual time=0.171..55.095 rows=4 loops=1)
Join Filter: (w.w_id = s.w_id)
-> Seq Scan on warehouse1 w (cost=0.00..12.00 rows=1 width=122) (actual time=0.019..0.283 rows=5 loops=1)
Filter: ((w_city)::text = 'Singapore'::text)
Rows Removed by Filter: 1000
-> Seq Scan on stock1 s (cost=0.00..893.03 rows=260 width=4) (actual time=0.121..10.903 rows=301 loops=5)
Filter: (i_id = 33)
Rows Removed by Filter: 44611
Planning time: 0.173 ms
Execution time: 55.135 ms

Nested Loop

- Nested Loop Join With Materialised Inner Sequential Scan

The filtered inner table may be materialised if there are several iterations of the inner loop.

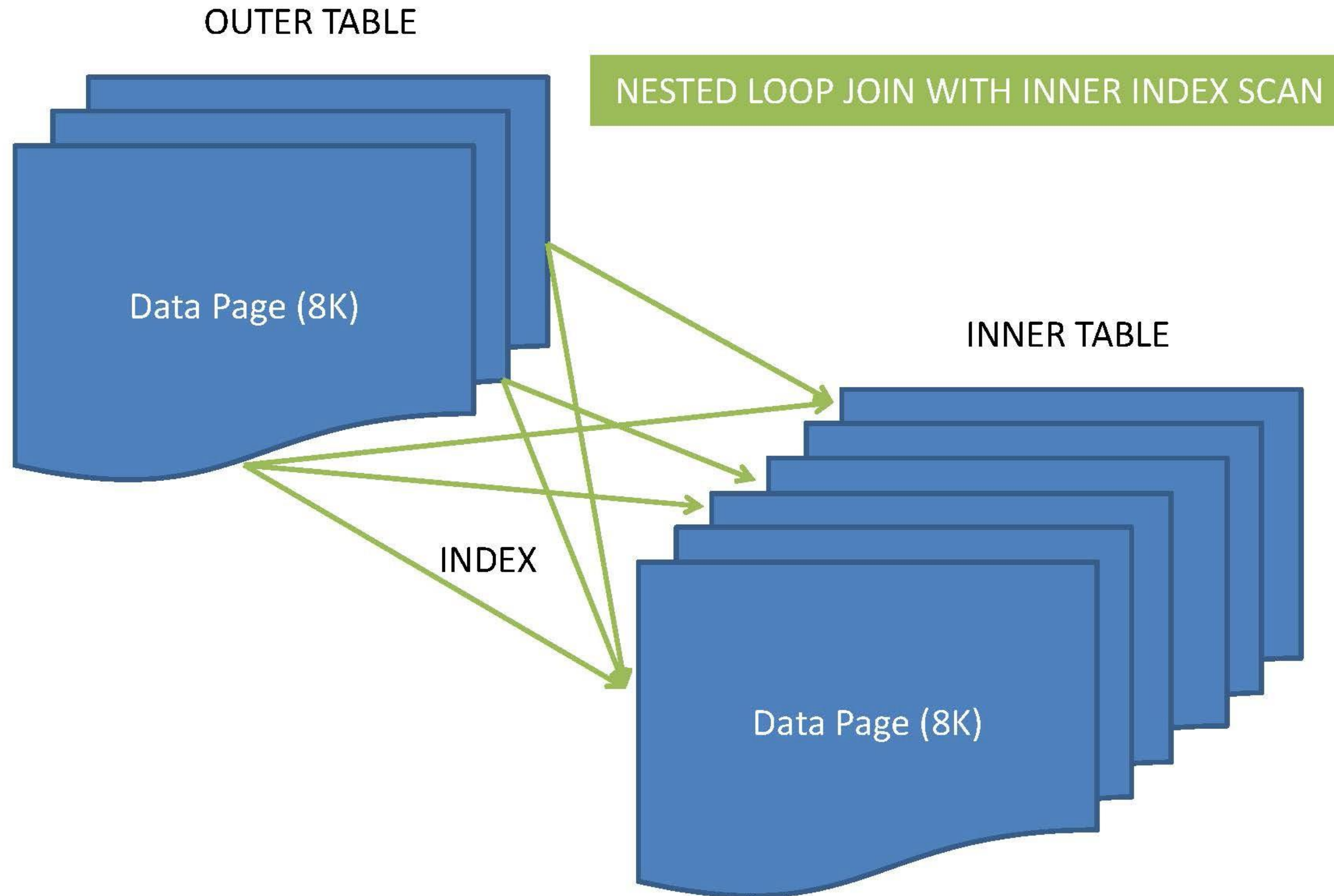
Let us artificially inflate the warehouse1 table size.

```
1 INSERT INTO warehouse1 SELECT * FROM warehouses;
2 INSERT INTO warehouse1 SELECT * FROM warehouses;
3
4 EXPLAIN ANALYZE SELECT w.w_name
5 FROM   warehouse1 w NATURAL JOIN stock1 s
6 WHERE w.w_city='Singapore' AND s.i_id=33;
```

Nested Loop

Query Plan
Nested Loop (cost=0.00..933.23 rows=3 width=118) (actual time=0.134..13.890 rows=12 loops=1)
Join Filter: (w.w_id = s.w_id) Rows Removed by Join Filter: 4503
-> Seq Scan on stock1 s (cost=0.00..893.03 rows=260 width=4) (actual time=0.118..11.527 rows=301 loops=1)
Filter: (i_id = 33)
Rows Removed by Filter: 44611
-> Materialize (cost=0.00..32.41 rows=2 width=122) (actual time=0.000..0.005 rows=15 loops=301)
-> Seq Scan on warehouse1 w (cost=0.00..32.40 rows=2 width=122) (actual time=0.010..1.080 rows=15 loops=1)
Filter: ((w_city)::text = 'Singapore'::text)
Rows Removed by Filter: 3000
Planning time: 0.132 ms
Execution time: 13.983 ms

Nested Loop



Nested Loop

- Query

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

```
1 SELECT s.i_id , s.s_qty
2 FROM warehouse w JOIN stock s ON w.w_id=s.w_id
3 WHERE w.w_name='Agimba';
```

Nested Loop

- Nested Loop Join With Inner Index Scan

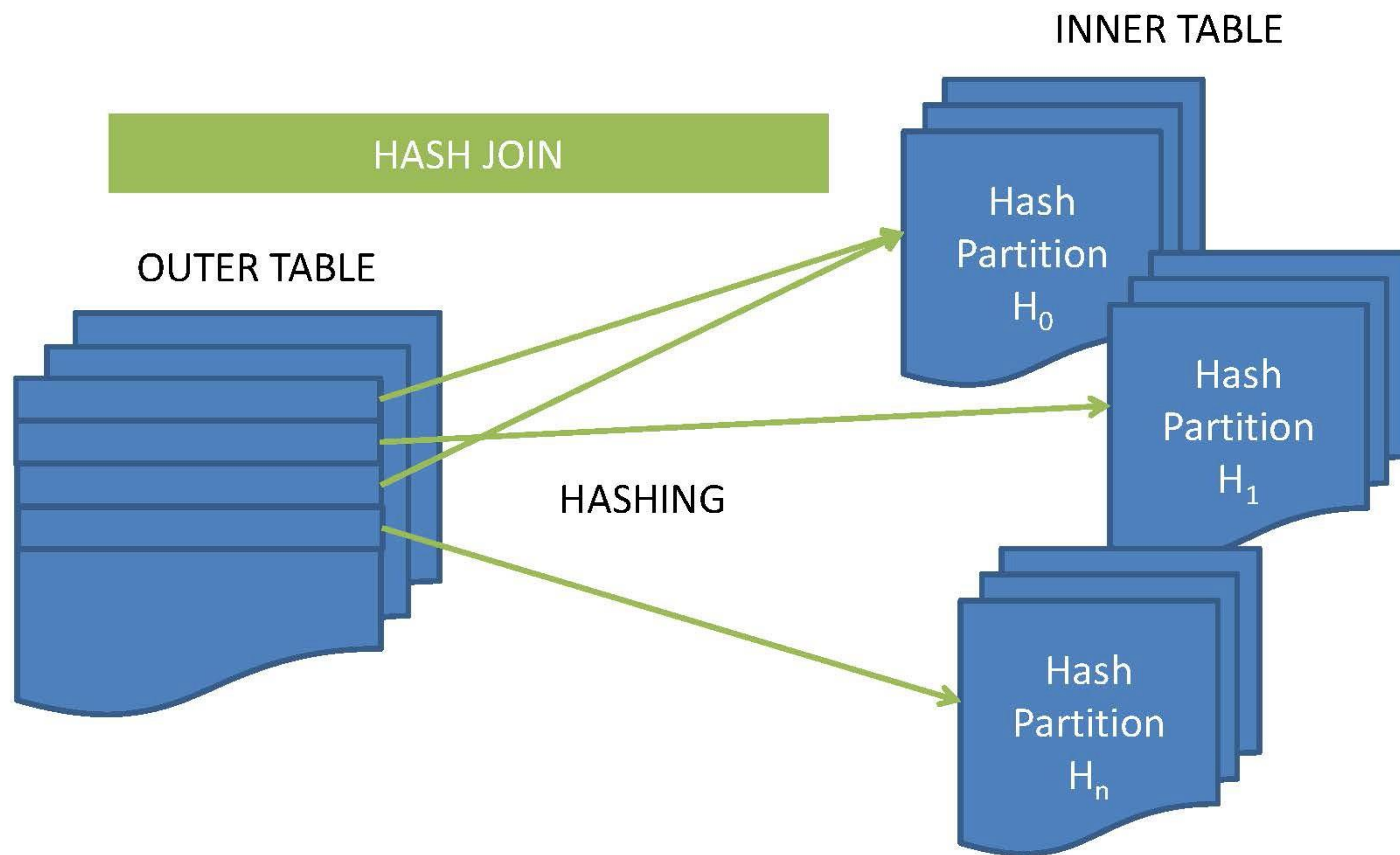
If an index is available, then the optimizer may choose a Nested Loop Join with an inner index scan.

```
1 EXPLAIN ANALYZE SELECT s.i_id, s.s_qty
2 FROM warehouses w JOIN stocks s ON w.w_id=s.w_id
3 WHERE w.w_name='Agimba';
```

Nested Loop

Query Plan
Nested Loop (cost=0.29..203.42 rows=89 width=6) (actual time=0.050..0.585 rows=444 loops=1)
-> Seq Scan on warehouse w (cost=0.00..21.56 rows=2 width=4) (actual time=0.031..0.274 rows=3 loops=1)
Filter: ((w_name)::text = 'Agimba'::text)
Rows Removed by Filter: 1002
-> Index Scan using stock_pkey on stock s (cost=0.29..90.45 rows=48 width=10) (actual time=0.014..0.062 rows=148 loops=3)
Index Cond: (w_id = w.w_id)
Planning time: 0.481 ms
Execution time: 0.659 ms

Hash Join



Hash Join

- Query

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

```
1 SELECT w.w_name, s.i_id , s.s_qty
2 FROM warehouses w, stocks s
3 WHERE w.w_id=s.w_id;
```

Hash Join

In general, the optimizer will choose a Hash Join

```
1 EXPLAIN ANALYZE SELECT w.w_name, s.i_id ,  s.s_qty
2 FROM warehouses w, stocks s
3 WHERE w.w_id=s.w_id;
```

Hash Join

Query Plan
Hash Join (cost=31.61..1341.27 rows=44912 width=13) (actual time=0.811..37.777 rows=44912 loops=1)
Hash Cond: (s.w_id = w.w_id)
-> Seq Scan on stock s (cost=0.00..692.12 rows=44912 width=10) (actual time=0.017..7.814 rows=44912 loops=1)
-> Hash (cost=19.05..19.05 rows=1005 width=11) (actual time=0.769..0.769 rows=1005 loops=1)
Buckets: 1024 Batches: 1 Memory Usage: 53kB
-> Seq Scan on warehouse w (cost=0.00..19.05 rows=1005 width=11) (actual time=0.024..0.353 rows=1005 loops=1)
Planning time: 0.573 ms
Execution time: 40.879 ms

Hash Join

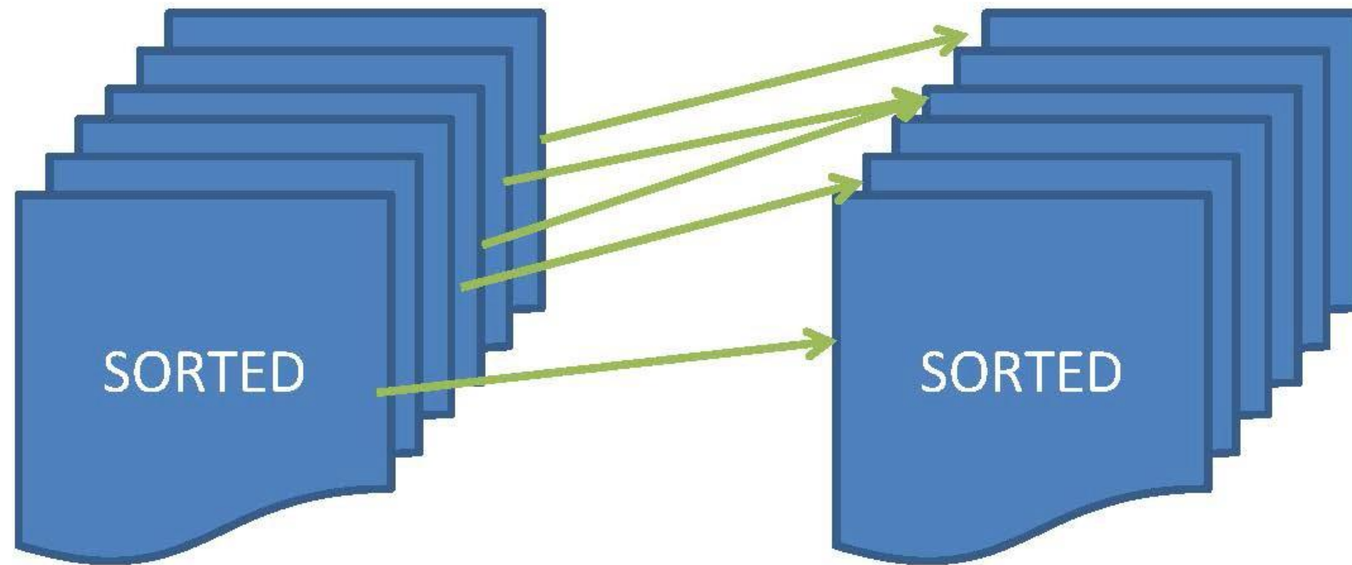
The plan is generally, but not always, the same regardless of the order of tables in the FROM clause.

```
1 EXPLAIN SELECT w.w_name, s.i_id, s.s_qty
2 FROM stocks s, warehouses w
3 WHERE w.w_id=s.w_id;
```

Hash Join

Query Plan
Hash Join (cost=31.61..1341.27 rows=44912 width=13)
Hash Cond: (s.w_id = w.w_id)
-> Seq Scan on stock s (cost=0.00..692.12 rows=44912 width=10)
-> Hash (cost=19.05..19.05 rows=1005 width=11)
-> Seq Scan on warehouse w (cost=0.00..19.05 rows=1005 width=11)

Merge Join



Merge Join

In the absence of statistics, PostgreSQL may prefer a Merge Join.

```
1 CREATE TABLE warehouse2 AS
2 SELECT * FROM warehouses;
```

Merge Join

- Hash Join

With minimum statistics PostgreSQL chooses a Hash Join.

```
1 EXPLAIN SELECT w1.w_name
2 FROM warehouse2 w1, warehouse2 w2
3 WHERE w1.w_name=w2.w_name;
```

Query Plan
Hash Join (cost=13.60..27.40 rows=160 width=118)
Hash Cond: ((w1.w_name)::text = (w2.w_name)::text)
-> Seq Scan on warehouse2 w1 (cost=0.00..11.60 rows=160 width=118)
-> Hash (cost=11.60..11.60 rows=160 width=118)"
-> Seq Scan on warehouse2 w2 (cost=0.00..11.60 rows=160 width=118)

Merge Join

We remove the statistics and PostgreSQL chooses a Merge Join.

```sql
SELECT * FROM pg_statistic s, pg_class t
WHERE s.starelid = t.oid
AND   t.relname='warehouse2';
```

```sql
DELETE FROM pg_statistic s
WHERE s.starelid = ANY (
  SELECT t.oid
  FROM pg_class t
  WHERE t.relname='warehouse2');
```

```sql
SELECT * FROM pg_statistic s, pg_class t
WHERE s.starelid = t.oid
AND t.relname='warehouse2';
```

Merge Join

```sql
EXPLAIN SELECT w1.w_name
FROM warehouse2 w1, warehouse2 w2
WHERE w1.w_name=w2.w_name;
```

Query Plan
Merge Join (cost=138.33..219.10 rows=5050 width=118)
Merge Cond: ((w1.w_name)::text = (w2.w_name)::text)
-> Sort (cost=69.16..71.68 rows=1005 width=118)
Sort Key: w1.w_name
-> Seq Scan on warehouse2 w1 (cost=0.00..19.05 rows=1005 width=118)
-> Sort (cost=69.16..71.68 rows=1005 width=118)
Sort Key: w2.w_name
-> Seq Scan on warehouse2 w2 (cost=0.00..19.05 rows=1005 width=118)

Merge Join

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

```sql
VACUUM;
VACUUM FULL;
VACUUM warehouse2;
VACUUM FULL warehouse2;
```

Merge Join

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

```
1 ANALYZE warehouse2;
2 ANALYZE;
3 VACUUM ANALYZE;
4 VACUUM ANALYZE warehouse2;
5 VACUUM FULL ANALYZE;
6 VACUUM ANALYZE warehouse2;
```

Merge Join

```
1 ANALYZE warehouse2;
```

```
1 SELECT * FROM pg_statistic s, pg_class t
2 WHERE s.starelid = t.oid
3 AND  t.relname='warehouse2';
```

```
1 EXPLAIN SELECT w1.w_name
2 FROM warehouse2 w1, warehouse2 w2
3 WHERE w1.w_name=w2.w_name;
```

Query Plan
Hash Join (cost=13.60..27.40 rows=160 width=118)
Hash Cond: ((w1.w_name)::text = (w2.w_name)::text)
-> Seq Scan on warehouse2 w1 (cost=0.00..11.60 rows=160 width=118)
-> Hash (cost=11.60..11.60 rows=160 width=118)"
-> Seq Scan on warehouse2 w2 (cost=0.00..11.60 rows=160 width=118)

Merge Join

PostgreSQL has an autovaccum daemon that can issue vacuum and analyse commands adaptively.

Merge Join

- Semi-Join

When the inner table is only used for filtering PostgreSQL can use a Semi-Join.

```
1 EXPLAIN SELECT w.w_name
2 FROM warehouses w
3 WHERE EXISTS (
4 SELECT *
5 FROM stocks s
6 WHERE s.w_id = w.w_id);
```

Query Plan
Nested Loop Semi Join (cost=0.29..435.16 rows=923 width=7)
-> Seq Scan on warehouse w (cost=0.00..19.05 rows=1005 width=11)
-> Index Only Scan using stock_pkey on stock s (cost=0.29..3.03 rows=49 width=4)
Index Cond: (w_id = w.w_id)

Merge Join

- Anti-Join

When the inner table is only used for excluding results PostgreSQL can use an Anti–Join.

```
1 EXPLAIN SELECT w.w_name
2 FROM warehouses w
3 WHERE NOT EXISTS (
4   SELECT *
5   FROM stock s
6   WHERE s.w_id = w.w_id);
```

Query Plan
Nested Loop Anti Join (cost=0.29..435.16 rows=82 width=7)
-> Seq Scan on warehouse w (cost=0.00..19.05 rows=1005 width=11)
-> Index Only Scan using stock_pkey on stock s (cost=0.29..3.03 rows=49 width=4)
Index Cond: (w_id = w.w_id)

Merge Join

- Anti-Join

When the inner table is only used for excluding results PostgreSQL can use an Anti–Join.

```
1 EXPLAIN SELECT w.w_name
2 FROM warehouses w LEFT OUTER JOIN stocks s ON w.w_id = s.w_id
3 WHERE s.w_id IS NULL;
```

Query Plan
Nested Loop Anti Join (cost=0.29..435.16 rows=82 width=7)
-> Seq Scan on warehouse w (cost=0.00..19.05 rows=1005 width=11)
-> Index Only Scan using stock_pkey on stock s (cost=0.29..3.03 rows=49 width=4)
Index Cond: (w_id = w.w_id)

Merge Join

- Don’t Use NOT IN

PostgreSQL does not really do well with NOT IN...

```
1 EXPLAIN SELECT w.w_name
2 FROM warehouses w
3 WHERE w.w_id NOT IN (
4   SELECT s.w_id
5   FROM stock s);
```

Query Plan
Seq Scan on warehouse w (cost=804.40..825.96 rows=502 width=7)
Filter: (NOT (hashed SubPlan 1))
SubPlan 1
-> Seq Scan on stock s (cost=0.00..692.12 rows=44912 width=4)

Merge Join

- Outer Join

And, of course, PostgreSQL implements OUTER JOINS.

```
1 EXPLAIN SELECT i.i_name, s.w_id
2 FROM items i  LEFT OUTER JOIN  stocks s ON s.i_id = i.i_id
3 WHERE i.i_name = 'MECLIZINE HYDROCHLORIDE'
```

Query Plan
Hash Right Join (cost=11.05..872.52 rows=93 width=22)
Hash Cond: (s.i_id = i.i_id)
-> Seq Scan on stock s (cost=0.00..692.12 rows=44912 width=8
-> Hash (cost=11.04..11.04 rows=1 width=22)
-> Seq Scan on item i (cost=0.00..11.04 rows=1 width=22)
Filter: ((i_name)::text = 'MECLIZINE HYDROCHLORIDE'::text)

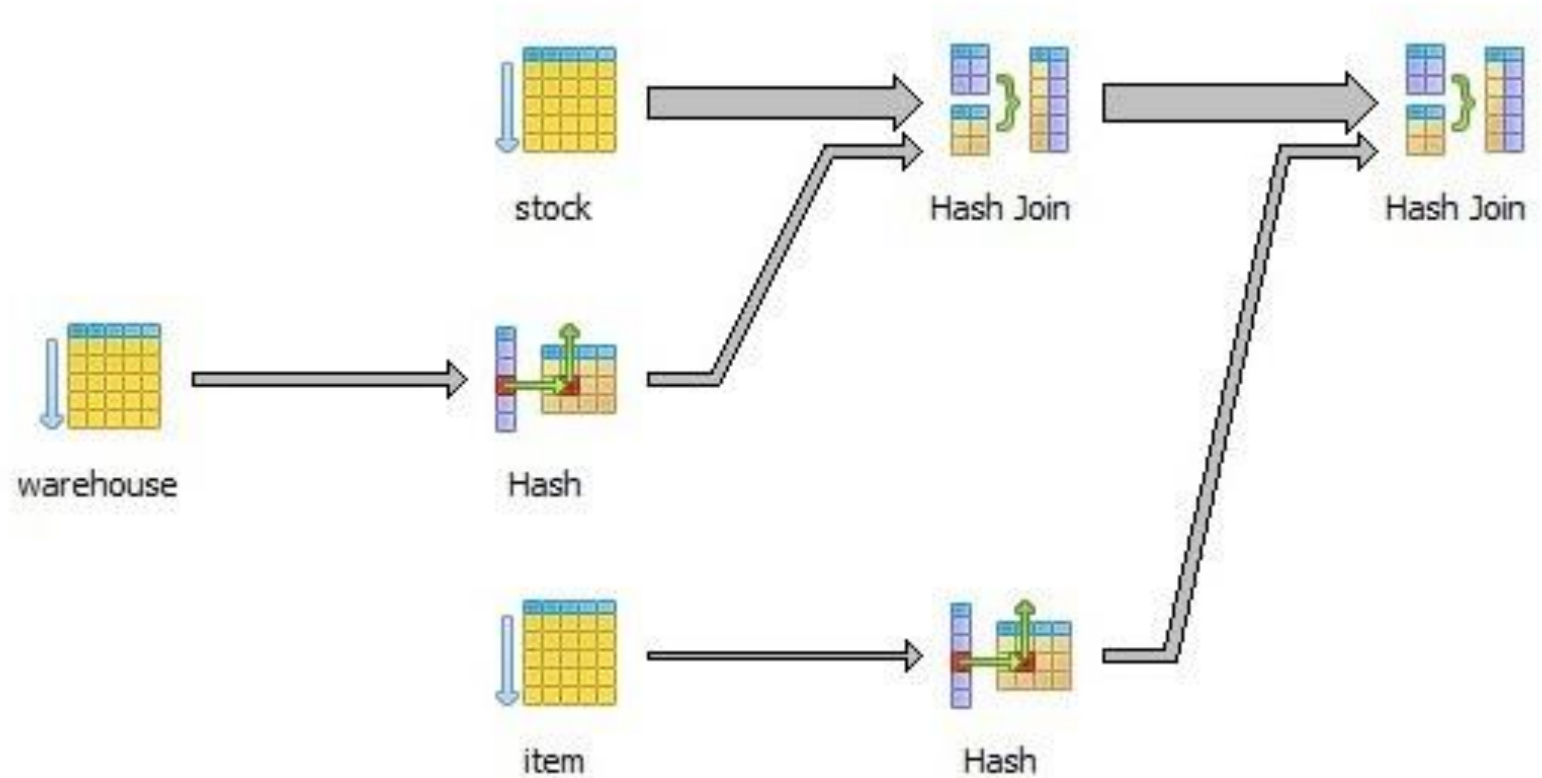
Join Order

The optimizer chooses one amongst several possible join orders. It can (indirectly) be forced to do so.

```
1 SELECT w.w_name, i.i_name, s.s_qty
2 FROM warehouses w, stocks s, items i
3 WHERE w.w_id=s.w_id AND s.i_id=i.i_id;
```

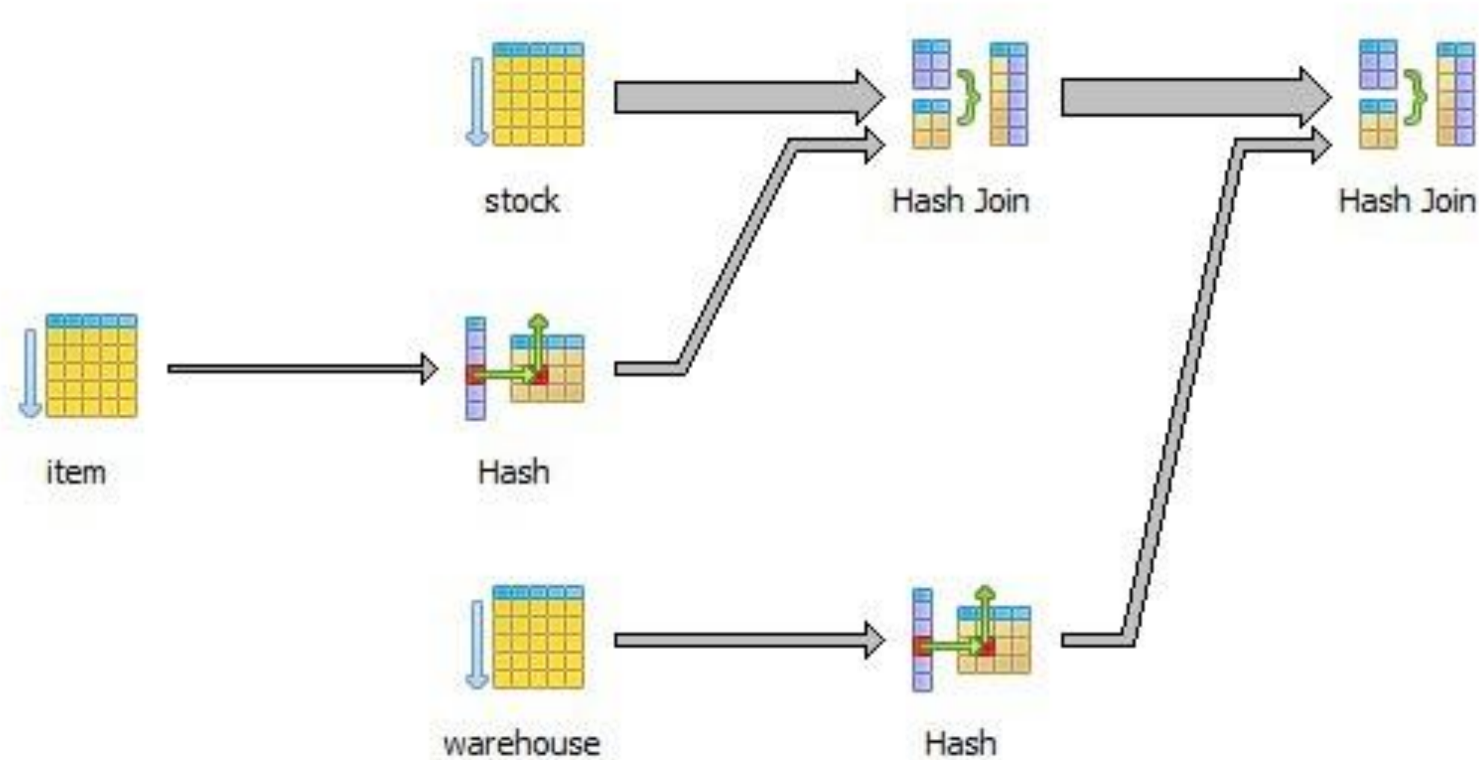
Join Order

```sql
SELECT w.w_name, i.i_name, s.s_qty
FROM warehouse w NATURAL JOIN stock s NATURAL JOIN item i;
```



Join Order

```sql
1 SELECT w.w_name, i.i_name, s.s_qty
2 FROM item i NATURAL JOIN stock s NATURAL JOIN warehouse w;
```



Agenda

- Index scans
- Multicolumn index
- Joins
- Denormalisation
- Views
- Materialized views

Denormalization

- Normalised Schema

A normalised schema requires to join tables on the equality of their primary and foreign keys. Joins can expensive.

Denormalization

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

Query Plan
"Hash Join ...
...
Planning time: 0.966 ms
Execution time: 73.789 ms+

Denormalization

```
1 EXPLAIN ANALYZE SELECT w.w_name, i.i_name, s.s_qty
2 FROM warehouses w NATURAL JOIN stocks s NATURAL JOIN items i
3 WHERE w.w_name='Agimba';
```

Query Plan
"Hash Join ...
...
Planning time: 1.318 ms
Execution time: 1.615 ms +

Denormalization

- Denormalised Schema

We can denormalise 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 (one must use triggers). Some but not all queries are faster.

Denormalization

```
1 EXPLAIN ANALYZE CREATE TABLE tall AS SELECT *
2 FROM warehouses w NATURAL JOIN stocks s NATURAL JOIN items i;
```

Query Plan
"Hash Join ...
...
Planning time: 0.798 ms
Execution time: 123.934 ms

Denormalization

```sql
EXPLAIN ANALYZE SELECT t.w_name, t.i_name, t.s_qty
FROM tall t;
```

Query Plan
Seq Scan on tall t ...
...
Planning time: 0.041 ms
Execution time: 28.884 ms+

Denormalization

```
1 EXPLAIN ANALYZE SELECT t.w_name, t.i_name, t.s_qty
2 FROM tall t
3 WHERE t.w_name='Agimba';
```

Query Plan
Seq Scan on tall t ...
...
Planning time: 0.094 ms
Execution time: 16.289 ms

Agenda

- Index scans
- Multicolumn index
- Joins
- Denormalisation
- Views
- Materialized views

Views

We can create views, but they are for convenience only. They do not change the performance from that of the underlying normalised schema (the view definition is used by the optimizer as would a subquery).

```
1 CREATE VIEW vall AS SELECT *
2 FROM warehouses w NATURAL JOIN stocks s NATURAL JOIN items i;
```

Views

```sql
EXPLAIN ANALYZE SELECT v.w_name, v.i_name, v.s_qty
FROM vall v;
```

Query Plan
"Hash Join ...
...
Planning time: 0.573 ms
Execution time: 52.701 ms

Views

```sql
EXPLAIN ANALYZE SELECT v.w_name, v.i_name, v.s_qty
FROM vall v
WHERE v.w_name='Agimba';
```

Query Plan
"Hash Join ...
...
Planning time: 0.856 ms
Execution time: 1.193 ms

Agenda

- Index scans
- Multicolumn index
- Joins
- Denormalisation
- Views
- Materialized views

Materialised Views

We can create materialised views. They are a middle ground between anormalised and a denormalised schema. The insertions, deletions and updates are more costly because of propagation (currently manual with triggers and REFRESH). Postgres does not use the materialized view definition to optimize the query. propagation.

```
1 CREATE MATERIALIZED VIEW mvall AS SELECT *
2 FROM warehouses w NATURAL JOIN stocks s NATURAL JOIN items i;
```

Materialised Views

```sql
1 EXPLAIN ANALYZE SELECT v.w_name, v.i_name, v.s_qty
2 FROM mvall v;
```

Query Plan
Seq Scan on mvall v ...
...
Planning time: 0.231 ms
Execution time: 22.740 ms

Materialised Views

```
1 EXPLAIN ANALYZE SELECT v.w_name, v.i_name, v.s_qty
2 FROM mvall v
3 WHERE v.w_name='Agimba';
```

Query Plan
Seq Scan on mvall v ...
...
Planning time: 0.112 ms
Execution time: 14.532 ms

Materialised Views

- Refreshing Materialised Views

Currently materialised views must be refreshed manually after insertion, deletion or update in the underlying tables.There is no propagation. The materialised view is just a copy.

```
1  SELECT * FROM stocks s WHERE s.w_id=2 AND s.i_id=1;
2
3  EXPLAIN ANALYZE INSERT INTO stocks VALUES (2,1,1);
4
5  SELECT * FROM stocks s WHERE s.w_id=2 AND s.i_id=1;
6
7  SELECT * FROM mvall v WHERE v.w_id=2 AND v.i_id=1;
8
9  REFRESH MATERIALIZED VIEW mvall;
10
11 SELECT * FROM mvall v WHERE v.w_id=2 AND v.i_id=1;
```

Materialised Views

- Note

We can also analyse the cost of insertion, deletions and updates. We see that constraints and indexes are not for free.

Query Plan
Insert on stock (cost=0.00..0.01 rows=1 width=10) (actual time=0.060..0.060 rows=0 loops=1)
-> Result (cost=0.00..0.01 rows=1 width=10) (actual time=0.001..0.001 rows=1 loops=1)
Planning time: 0.035 ms
Trigger for constraint stock_w_id_fkey: time=0.223 calls=1
Trigger for constraint stock_i_id_fkey: time=0.136 calls=1
Execution time: 0.444 ms"

Materialised Views

- Indexing Materialised Views

Materialised Views can be Indexed.

```
1 EXPLAIN ANALYZE
2 SELECT * FROM mvall v WHERE v.w_id=2 AND v.i_id=1;
```

Query Plan
Seq Scan on mvall v ...
...
Planning time: 0.087 ms
Execution time: 11.090 ms

Materialised Views

```sql
CREATE  INDEX mvall_pkey ON mvall(w_id, i_id);
```

```sql
EXPLAIN ANALYZE
SELECT * FROM mvall v WHERE v.w_id=2 AND v.i_id=1;
```

Query Plan
Index Scan using mvall_pkey on mvall v ...
...
Planning time: 0.554 ms
Execution time: 0.081 ms

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

Arguably Good or Bad Things to Do

```
1 PREPARE q AS SELECT s.i_id
2 FROM stocks s
3 WHERE s.s_qty > 500;
4
5 EXPLAIN ANALYZE EXECUTE q;
```

Query Plan
Seq Scan on stock s (cost=0.00..1047.40 rows=44912 width=4) (actual time=0.482..18.560 rows=44912 loops=1)
Filter: (s_qty > 500)
Execution time: 21.659 ms

```
1 DEALLOCATE q;
```

Arguably Good or Bad Things to Do

- Planner Method Configuration

It is not recommended configure the optimizer by turning off some methods. See https://www.postgresql.org/docs/13/runtime-config-query.html.

```
1 SET enable_seqscan = false;
```

```
1 SET enable_bitmapscan = false;
```

```
1 SET enable_hashjoin = false;
```

etc.

Arguably Good or Bad Things to Do

- Hints

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

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

Arguably Good or Bad Things to Do

```sql
SELECT i.i_name FROM warehouse
USE INDEX (i_i_price) WHERE i.i_price < 100;
```

```sql
SELECT i.i_name FROM warehouse
IGNORE INDEX (i_i_price) WHERE i.i_price < 100;
```

```sql
SELECT i.i_name FROM warehouse
FORCE INDEX (i_i_price) WHERE i.i_price < 100;
```

```sql
SELECT s.i_id, s.s_qty
FROM warehouse w STRAIGHT_JOIN stock s ON w.w_id=s.w_id
WHERE w.w_name='Agimba';
```

Workload optimization

- Materialized view selection
- ML-based hint selection
- Index recommendation
- Many more

Take Home Messages

- Why are Queries Slow
  - Wrong design
  - Poor configuration (increase work_mem);
  - Tuples are scattered, tables and indexes are bloated (VACUUM, CLUSTER, VACUUM FULL, reindexing);
  - Missing indexes (CREATE INDEX);
  - PostgreSQL does not choose the best plan (ANALYZE);

Take Home Messages

- In conclusion
  - Understand the optimizer;
  - Tune the data (normalise, denormalise, index, create views, materialised) for everyone;
  - Help the system maintain good statistics;
  - Hard-tune the queries as a last resort and at every users' current and future risk.