

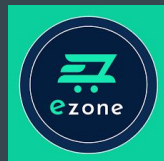
Postgresql Query Optimization Techniques

...

Tripoli Programmers' Club
2nd meeting

About

- My name is Malik Ashebani
- Currently: Backend Engineer @ezone.ly
- Previously worked at: @Muljin, @ICTS
- BSc in CS from @Elmergib Uni 2021
- Interests: Reverse Engineering, PL/PA, Vulnerability Research & Database Systems
- Occasional CTF Player (Hackathon Libya 2023 1st place with @CyberGhosts, Flare-On 5, 6, 7)
- OSS Contrib': B2R2 framework (e.g. LEB128 decoder, Wasm Binary Modules parser)



Where to find me

- LinkedIn: [linkedin.com/in/malik-m-ashebani](https://www.linkedin.com/in/malik-m-ashebani)
- Github: github.com/risc-vee
- Blog: xtlab.wordpress.com

Things to keep in mind

- I might be wrong, do your research if you're in doubt.
- Most of concepts here are roughly simplified due to limited time.
- Most concepts here can be similarly applied to other relational databases (e.g. SQL Server, MySQL, Oracle).
- While relational databases has a lot in common, they differ in capabilities they have and their implementation details, consult your favourite db's documentation.

Brief history of relational model

- Invented by Edgar F. Codd in 1970 @IBM Research Lab.
- Based on Relational Algebra.
- Came to solve pain points in other models at that time:
 - Logical layer (entities and attributes) is tightly coupled to Physical layer (storage).
 - Rewriting queries everytime the (schema) or data changes.

Information Retrieval

P. BAXENDALE, Editor

A Relational Model of Data for Large Shared Data Banks

E. F. CODD

IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section

Relational theory of Edgar Codd

- Fundamental concepts:
 - “tuple”: a set of attributes, equivalent to a row.
 - “relation”: a set of tuples that represent an entity, equivalent to a table.
 - “predicate”: a condition or multiple conditions.
- Relational Algebra provides operations to retrieve and manipulate tuples in a relation based on “set algebra”.
- Each operator takes one or more relations as input and produces a new relation as output.

Relational operators

σ Selection (WHERE in SQL)

π Projection (SELECT in SQL)

\times Product (CROSS JOIN or SELECT * FROM R, S in SQL)

\cup Union (UNION in SQL)

\cap Intersection (INTERSECT in SQL)

$-$ Difference (EXCEPT in SQL)

\bowtie Join (JOIN in SQL)

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

(R \times S)

R.a_id	R.b_id	S.a_id	S.b_id
a1	101	a3	103
a1	101	a4	104
a1	101	a5	105
a2	102	a3	103
a2	102	a4	104
a2	102	a5	105
a3	103	a3	103
a3	103	a4	104
a3	103	a5	105

Relational operators

```
SELECT o.id, o.order_total, c.name
```

```
FROM Customers c
```

```
JOIN Orders o ON o.customer_id = c.id
```

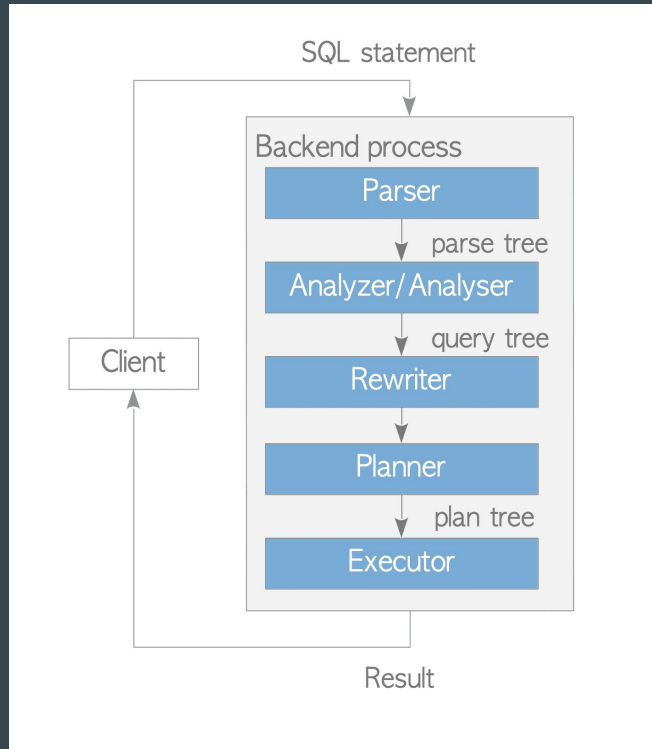
can be expressed like this:

```
 $\pi_{id, order\_total, name} (\sigma_{o.customer\_id = c.id} (Customers \times Orders))$ 
```


Relational operators properties and equivalence rules

- Commutativity:
 - The ability to change the order of operations without affecting the result.
 - Example 1: $R \bowtie S = S \bowtie R$
 - Example 2: $R \cup S = S \cup R$
- Associativity:
 - The ability to regroup operations without affecting the result.
 - Example 1: $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
 - Example 2: $(R \cup S) \cup T = R \cup (S \cup T)$
- Distributivity:
 - Describes how one operation can be distributed over another.
 - Example 1: $\sigma_{\text{condition}}(R \bowtie S) = (\sigma_{\text{condition}}(R)) \bowtie S$
 - Example 2: $\pi_A(R \cup S) = \pi_A(R) \cup \pi_A(S)$

Query lifecycle overview



<https://www.interdb.jp/pg/pgsql03>

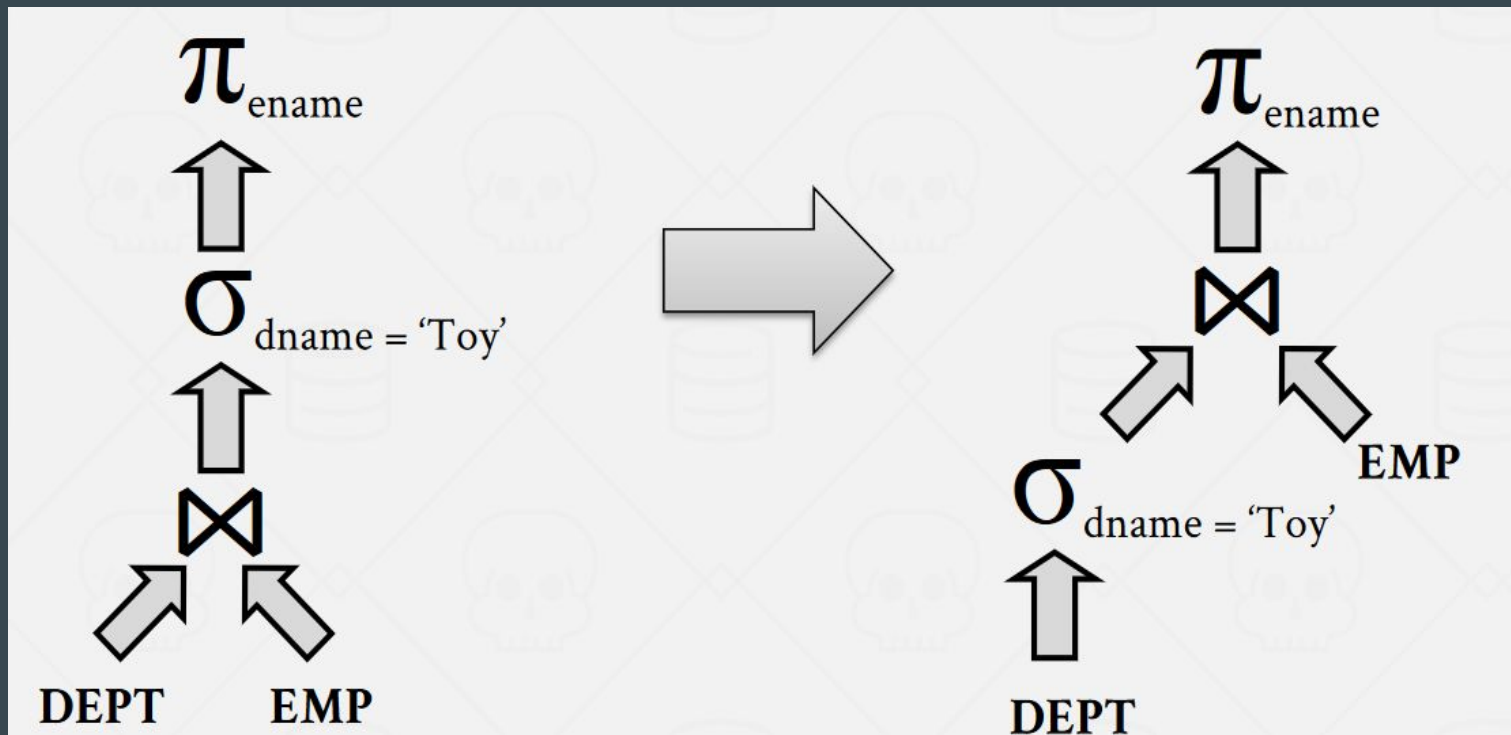
Query optimization strategies

- Rule-based (Heuristics):
 - A pre-existing set of static transformation rules are applied to matching portions of a query to eliminate inefficiencies, these rules always produce more efficient queries.
 - Examples:
 - `SELECT * FROM R WHERE 1 = 0 OR a1 > 5` → `SELECT * FROM R WHERE a1 > 5`
 - `SELECT * FROM S WHERE a1 > 1 + 2 * 3` → `SELECT * FROM S WHERE a1 > 7`
 - `SELECT * FROM R WHERE R.id = 1 OR R.id IN (SELECT id FROM S WHERE 1>1)`
→ `SELECT * FROM R WHERE R.id = 1`
 - Real-world example:
 - `SELECT * FROM Orders WHERE (@status=0 OR status = @status)`
 - → `SELECT * FROM Orders WHERE status=@status` (when @status <> 0)
 - → `SELECT * FROM Orders` (when @status = 0)
- Cost-based search (will be discussed in subsequent slides)

Logical plans

- Logical plan is constructed from the parsed query using relational algebra operators, this plan represents the high-level steps (in the form of a tree) in which a query will be executed without yet considering how the data will be physically accessed or processed.
- Example optimizations:
 - Perform filters as early as possible (Predicate Pushdown).
 - Get the necessary columns only as early as possible (Projection Pushdown).
 - Replace Cartesian Products with Joins.

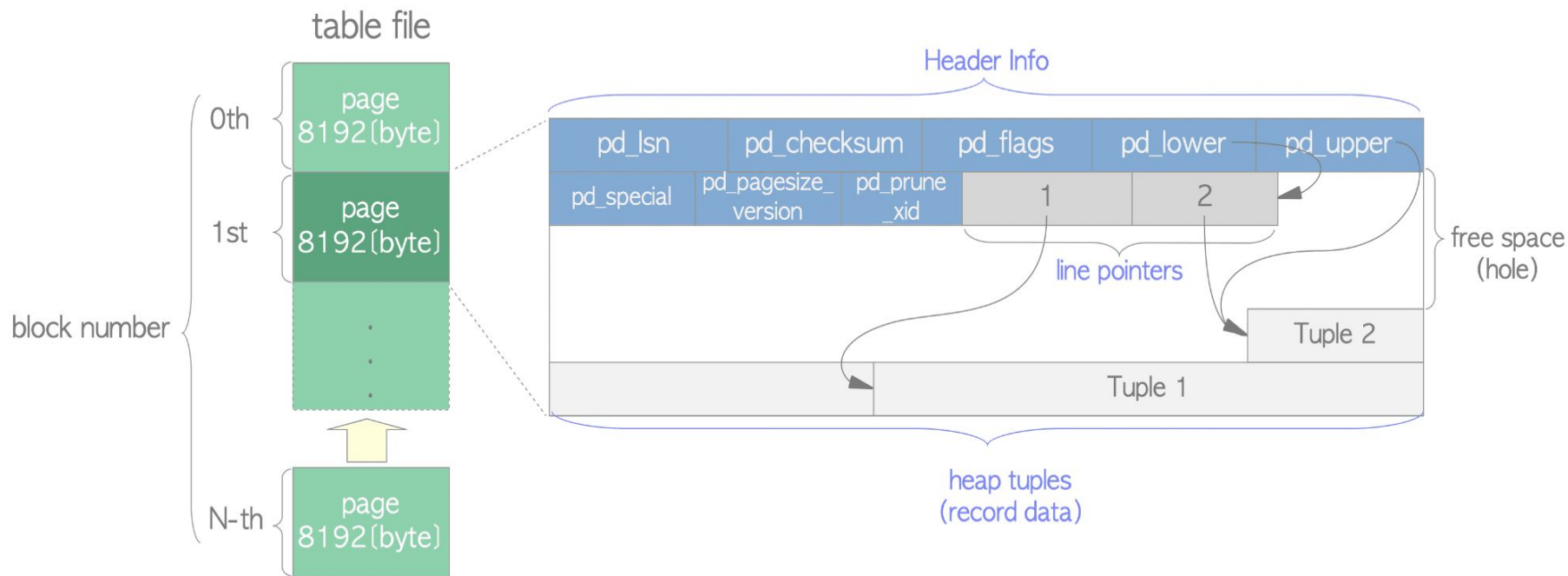
Logical plans (Predicate Pushdown example)



Physical plans and Cost-based search

- In theory, a physical plan is generated by mapping the logical plan into a physical plan.
- Multiple equivalent physical plans are enumerated, the cheapest plan is picked.

How data is actually stored on disk?

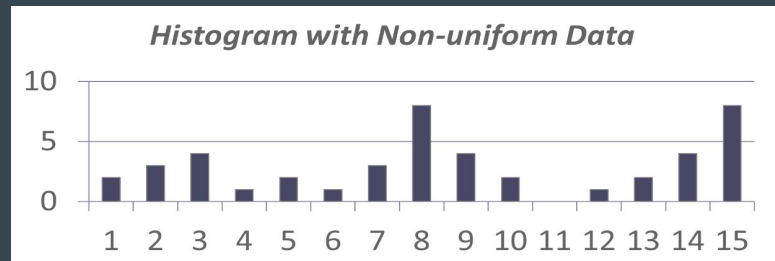


Data Access algorithms

- Full/Sequential scan.
- Index-based table access:
 - Index scan.
 - Bitmap heap scan
- Index-only scan.

How the best algorithm is chosen?

- The database maintains statistics about stored data
 - Table-level statistics:
 - # of tuples (cardinality).
 - # of pages.
 - average tuple width (size).
 - Column-level statistics:
 - # of distinct values.
 - distribution of data through histograms
 - most frequent values and their frequencies.
 - # of NULL values.
- These statistics are randomly-sampled.



How the best algorithm is chosen? contd'

For efficiency reasons, `reltuples` and `relpages` are not updated on-the-fly, and so they usually contain somewhat out-of-date values. They are updated by `VACUUM`, `ANALYZE`, and a few DDL commands such as `CREATE INDEX`. A stand-alone `ANALYZE`, that is one not part of `VACUUM`, generates an approximate `reltuples` value since it does not read every row of the table. The planner will scale the values it finds in `pg_class` to match the current physical table size, thus obtaining a closer approximation.

Most queries retrieve only a fraction of the rows in a table, due to `WHERE` clauses that restrict the rows to be examined. The planner thus needs to make an estimate of the *selectivity* of `WHERE` clauses, that is, the fraction of rows that match each condition in the `WHERE` clause. The information used for this task is stored in the `pg_statistic` system catalog. Entries in `pg_statistic` are updated by the `ANALYZE` and `VACUUM ANALYZE` commands, and are always approximate even when freshly updated.

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;
```

```
QUERY PLAN
```

```
-----  
Seq Scan on tenk1 (cost=0.00..483.00 rows=7001 width=244)  
  Filter: (unique1 < 7000)
```

Notice that the `EXPLAIN` output shows the `WHERE` clause being applied as a “filter” condition attached to the Seq Scan plan node. This means that the plan node checks the condition for each row it scans, and outputs only the ones that pass the condition. The estimate of output rows has been reduced because of the `WHERE` clause. However, the scan will still have to visit all 10000 rows, so the cost hasn't decreased; in fact it has gone up a bit (by $10000 * \text{cpu_operator_cost}$, to be exact) to reflect the extra CPU time spent checking the `WHERE` condition.

The actual number of rows this query would select is 7000, but the `rows` estimate is only approximate. If you try to duplicate this experiment, you will probably get a slightly different estimate; moreover, it can change after each `ANALYZE` command, because the statistics produced by `ANALYZE` are taken from a randomized sample of the table.

How the best algorithm is chosen? contd'

- Generally, the best algorithm is chosen based on Selectivity (along with other factors).
- Selectivity is the percentage of data accessed for a given predicate.
- Selectivity drives the number of output rows (aka cardinality estimation) which in turn, influences the cost of an algorithm.
- Postgres uses “magic constants” cost model, which are a constant value parameters that represent the relative cost of each physical operator.

Sequential scan

foreach page in R:

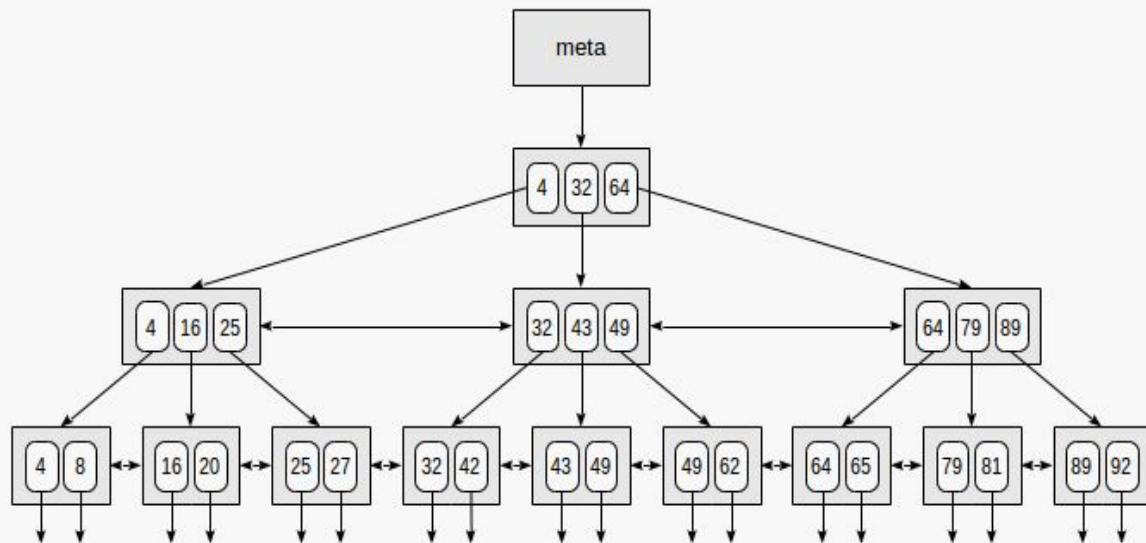
 read page into memory

 for each row in page:

 if condition(row):

 emit row

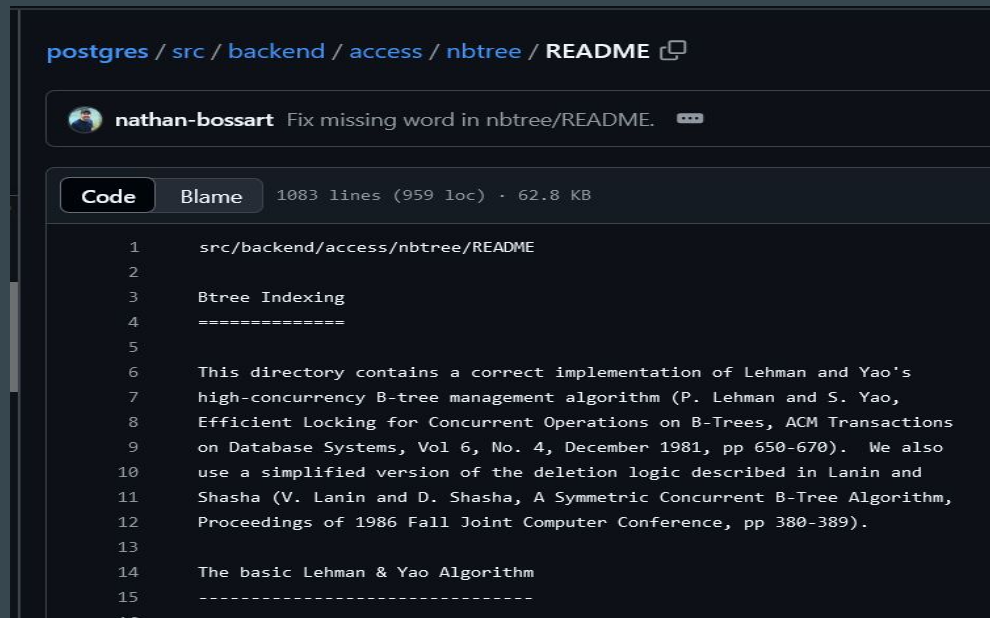
B-Tree indexes





<https://postgrespro.com/blog/pgsql/4161516>

B-Tree Indexes

- Most popular index of all times.
- Very efficient searches, inserts and deletes in $O(\log n)$ time.
- Has many variants (B+Tree, B ϵ Tree, BlinkTree)



```
postgres / src / backend / access / nbtree / README 
nathan-bossart Fix missing word in nbtree/README. 
Code Blame 1083 lines (959 loc) · 62.8 KB
1 src/backend/access/nbtree/README
2
3 Btree Indexing
4 =====
5
6 This directory contains a correct implementation of Lehman and Yao's
7 high-concurrency B-tree management algorithm (P. Lehman and S. Yao,
8 Efficient Locking for Concurrent Operations on B-Trees, ACM Transactions
9 on Database Systems, Vol 6, No. 4, December 1981, pp 650-670). We also
10 use a simplified version of the deletion logic described in Lanin and
11 Shasha (V. Lanin and D. Shasha, A Symmetric Concurrent B-Tree Algorithm,
12 Proceedings of 1986 Fall Joint Computer Conference, pp 380-389).
13
14 The basic Lehman & Yao Algorithm
15 -----
```

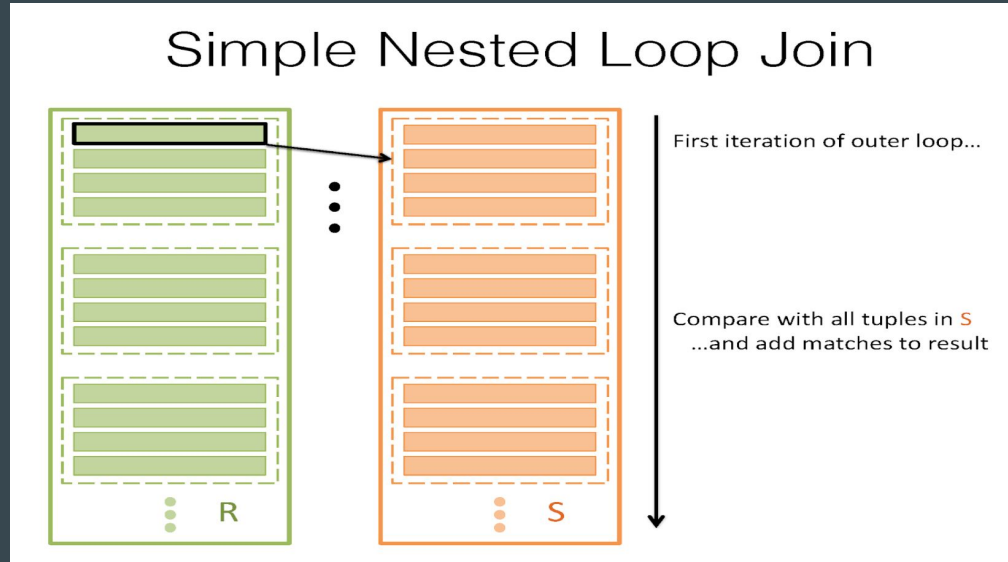
Bitmap heap scan

- Generally, used when the selectivity is high for index scan to be efficient but low for a seq scan to be efficient.

Block #	1	2	3	4	5
index 1	1	0	1	0	1
index 2	1	1	0	0	1
BitmapAnd					
Result	1	0	0	0	1

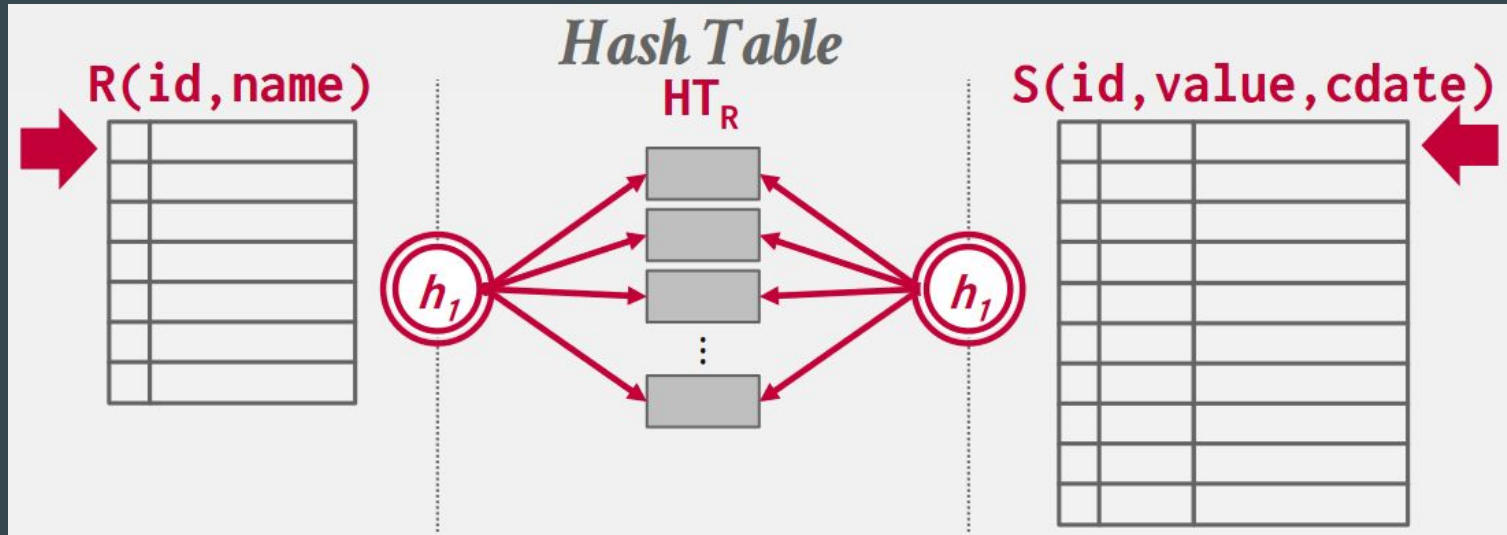
Nested Loop Joins

$$\text{Cost} = \text{size}(R) + (\text{size}(R) \times \text{size}(S))$$



Hash Join

$$\text{Cost} = \text{size}(R) + \text{size}(S)$$



Sort-Merge Join

Cost = size(R) + size(S) assuming tables are sorted

R(id, name)	
id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id, value, cdate)		
id	value	cdate
100	2222	10/4/2023
100	9999	10/4/2023
200	8888	10/4/2023
400	6666	10/4/2023
500	7777	10/4/2023

Anatomy of a Postgresql Execution Plan

SELECT *

FROM flight

WHERE scheduled_departure BETWEEN '2023-08-15' AND '2023-08-31';

QUERY PLAN

text



Bitmap Heap Scan on flight (cost=865.75..10324.44 rows=59446 width=71)

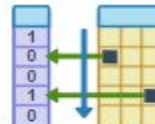
Recheck Cond: ((scheduled_departure >= '2023-08-15 00:00:00+00'::timestamp with time zone) AND (scheduled_departure <= '2023-08-31 00:00:00+00'::timestamp with time zone))

-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..850.88 rows=59446 width=0)

Index Cond: ((scheduled_departure >= '2023-08-15 00:00:00+00'::timestamp with time zone) AND (scheduled_departure <= '2023-08-31 00:00:00+00'::timestamp with time zone))



flight_scheduled_d-
eparture



flight

Optimizing “short” queries

- What are “short” queries?
- Long query example:

```
SELECT
    d.airport_code AS departure_airport,
    a.airport_code AS arrival_airport
FROM airport a, airport d
```

Optimizing “short” queries

- Short query example:

```
SELECT
    f.flight_no,
    f.scheduled_departure,
    boarding_time,
    p.last_name,
    p.first_name,
    bp.update_ts as pass_issued, ff.level
FROM flight f
JOIN booking_leg bl ON bl.flight_id = f.flight_id
JOIN passenger p ON p.booking_id=bl.booking_id
JOIN account a on a.account_id =p.account_id
JOIN boarding_pass bp on bp.passenger_id=p.passenger_id
LEFT OUTER JOIN frequent_flyer ff on ff.frequent_flyer_id=a.frequent_flyer_id
WHERE f.departure_airport = 'JFK'
    AND f.arrival_airport = 'ORD'
    AND f.scheduled_departure BETWEEN '2023-08-05' AND '2023-08-07';
```

Optimizing “short” queries

- The general rule of thumb for optimizing “short” queries is to make filters as restrictive as possible.
- This can be achieved by applying the right indexes on the most selective search terms.
- Generally speaking, selecting columns to index is achieved by comparing the number of unique values to the total number of rows in a table, the lower the selectivity, the more likely an index will be used by the query planner.

Example of when an index is NOT utilized by the planner

Actual flights = 352,829

```
SELECT *  
FROM flight f  
WHERE f.scheduled_departure BETWEEN '2023-05-15' AND '2023-08-31'
```

QUERY PLAN

text



Seq Scan on flight f (cost=0.00..18814.67 rows=351292 width=71)

Filter: ((scheduled_departure >= '2023-05-15 00:00:00+00':timestamp with time zone) AND (scheduled_departure <= '2023-08-31 00:00:00+00':timestamp with time z...

Example of when an index is utilized by the planner

Actual flights = 113,038

```
SELECT *  
FROM flight f  
WHERE f.scheduled_departure BETWEEN '2023-08-01' AND '2023-08-31'
```

QUERY PLAN

text



Bitmap Heap Scan on flight f (cost=1608.60..11838.29 rows=110846 width=71)

Recheck Cond: ((scheduled_departure >= '2023-08-01 00:00:00+00':timestamp with time zone) AND (scheduled_departure <= '2023-08-31 00:00:00+00':timestamp with time z...

-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..1580.88 rows=110846 width=0)

Index Cond: ((scheduled_departure >= '2023-08-01 00:00:00+00':timestamp with time zone) AND (scheduled_departure <= '2023-08-31 00:00:00+00':timestamp with time z...

Column transformations and functional indexes

Column transformations block the usage of an index.

```
CREATE INDEX account_last_name
ON account (last_name);

SELECT *
FROM account
WHERE lower(last_name)='daniels';
```

QUERY PLAN

text



Seq Scan on account (cost=0.00..21233.43 rows=4325 width=5...)

Filter: (lower(last_name) = 'daniels':text)

Column transformations and functional indexes

```
CREATE INDEX account_last_name_lower  
ON account (lower(last_name));
```

```
SELECT *  
FROM account  
WHERE lower(last_name)='DaNieLs'
```

QUERY PLAN

text



Bitmap Heap Scan on account (cost=49.94..7201.32 rows=4325 width=53)

Recheck Cond: (lower(last_name) = 'DaNieLs'::text)

-> Bitmap Index Scan on account_last_name_lower (cost=0.00..48.86 rows=4325 width=...

Index Cond: (lower(last_name) = 'DaNieLs'::text)

Column transformations and type casting

```
SELECT *  
FROM flight  
WHERE scheduled_departure ::date  
BETWEEN '2023-08-17' AND '2023-08-18'
```

QUERY PLAN

text



Seq Scan on flight (cost=0.00..22230.56 rows=3416 width=71)

Filter: (((scheduled_departure)::date >= '2023-08-17'::date) AND ((scheduled_departure)::date <= '2023-08-18'::da...

Column transformations and type casting

```
SELECT *  
FROM flight  
WHERE scheduled_departure  
BETWEEN '2023-08-17' AND '2023-08-18'
```

QUERY PLAN

text



Bitmap Heap Scan on flight (cost=63.99..7308.95 rows=4250 width=71)

Recheck Cond: ((scheduled_departure >= '2023-08-17 00:00:00+00'::timestamp with time zone) AND (scheduled_departure <= '2023-08-18 00:00:00+00'::timestamp with time z...

-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..62.92 rows=4250 width=0)

Index Cond: ((scheduled_departure >= '2023-08-17 00:00:00+00'::timestamp with time zone) AND (scheduled_departure <= '2023-08-18 00:00:00+00'::timestamp with time z...

Column transformations and coalesce()

```
SELECT *  
FROM flight  
WHERE coalesce(actual_departure, scheduled_departure)  
BETWEEN '2023-08-17' AND '2023-08-18'
```

QUERY PLAN

text

Seq Scan on flight (cost=0.00..18814.67 rows=3416 width=71)

Filter: ((COALESCE(actual_departure, scheduled_departure) >= '2023-08-17 00:00:00+00':timestamp with time zone) AND (COALESCE(actual_departure, scheduled_departure) <= '2023-08-18 00:00:00+00':timestamp with time zone))

Column transformations and coalesce()

```
SELECT * FROM flight
WHERE (actual_departure
      BETWEEN '2023-08-17' AND '2023-08-18')
OR (actual_departure IS NULL
    AND scheduled_departure BETWEEN '2023-08-17' AND '2023-08-18')
```

QUERY PLAN

text

Bitmap Heap Scan on flight (cost=68.57..7334.79 rows=2429 width=71)

Recheck Cond: (((actual_departure >= '2023-08-17 00:00:00+00':timestamp with time zone) AND (actual_departure <= '2023-08-18 00:00:00+00':timestamp with time zone)) OR ((scheduled_departure >= '2023-08-17 00:00:00+00':timestamp with time zone) AND (scheduled_departure <= '2023-08-18 00:00:00+00':timestamp with time zone)))

Filter: (((actual_departure >= '2023-08-17 00:00:00+00':timestamp with time zone) AND (actual_departure <= '2023-08-18 00:00:00+00':timestamp with time zone)) OR ((actual_departure IS NULL) AND (scheduled_departure >= '2023-08-17 00:00:00+00':timestamp with time zone) AND (scheduled_departure <= '2023-08-18 00:00:00+00':timestamp with time zone)))

-> BitmapOr (cost=68.57..68.57 rows=4250 width=0)

-> Bitmap Index Scan on flight_actual_departure (cost=0.00..4.43 rows=1 width=0)

Index Cond: ((actual_departure >= '2023-08-17 00:00:00+00':timestamp with time zone) AND (actual_departure <= '2023-08-18 00:00:00+00':timestamp with time zone))

-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..62.92 rows=4250 width=0)

Index Cond: ((scheduled_departure >= '2023-08-17 00:00:00+00':timestamp with time zone) AND (scheduled_departure <= '2023-08-18 00:00:00+00':timestamp with time zone))

Compound Indexes (before)

```
SELECT
    scheduled_departure ,
    scheduled_arrival
FROM flight
WHERE departure_airport='ORD'
    AND arrival_airport='JFK'
    AND scheduled_departure BETWEEN '2023-07-03' AND '2023-07-04';
```

QUERY PLAN

text

Bitmap Heap Scan on flight (cost=314.17..318.19 rows=1 width=16)

Recheck Cond: ((scheduled_departure >= '2023-07-03 00:00:00+00':timestamp with time zone) AND (scheduled_departure <= '2023-07-04 00:00:00+00':timestamp with time zone) AND (arrival_airport = 'JFK':bpchar))

-> BitmapAnd (cost=314.17..314.17 rows=1 width=0)

-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..49.84 rows=3341 width=0)

Index Cond: ((scheduled_departure >= '2023-07-03 00:00:00+00':timestamp with time zone) AND (scheduled_departure <= '2023-07-04 00:00:00+00':timestamp with time zone))

-> Bitmap Index Scan on flight_arrival_airport (cost=0.00..114.30 rows=10384 width=0)

Index Cond: (arrival_airport = 'JFK':bpchar)

-> Bitmap Index Scan on flight_departure_airport (cost=0.00..149.53 rows=13481 width=0)

Index Cond: (departure_airport = 'ORD':bpchar)

Compound Indexes

```
CREATE INDEX flight_depart_arr_sched_dep ON flight(  
departure_airport,  
arrival_airport,  
scheduled_departure);
```

QUERY PLAN

text

Index Scan using flight_depart_arr_sched_dep on flight (cost=0.42..8.45 rows=1 width=16)

Index Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar) AND (scheduled_departure >= '2023-07-03 00:00:00+00'::timestamp with time zone) AND (scheduled_departure <=

Covering indexes

```
CREATE INDEX flight_depart_arr_sched_dep_inc_sched_arr ON flight
(departure_airport,
arrival_airport,
scheduled_departure)
INCLUDE (scheduled_arrival);

SELECT
    departure_airport,
    scheduled_departure,
    scheduled_arrival
FROM flight
WHERE arrival_airport='JFK' AND departure_airport='ORD'
      AND scheduled_departure BETWEEN '2023-07-03' AND '2023-07-04';
```

QUERY PLAN

text

Index Only Scan using flight_depart_arr_sched_dep_inc_sched_arr on flight (cost=0.42..4.45 rows=1 width=20)

Index Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar) AND (scheduled_departure >= '2023-07-03 00:00:00+00'::timestamp with time zone) AND (scheduled_departure <=

Partial indexes (before)

```
SELECT * FROM flight WHERE  
scheduled_departure between '2023-08-13' AND '2023-08-14'  
AND status='Canceled';
```

QUERY PLAN

text



Bitmap Heap Scan on flight (cost=51.67..6683.21 rows=1 width=71)

Recheck Cond: ((scheduled_departure >= '2023-08-13 00:00:00+00'::timestamp with time zone) AND (scheduled_departure <= '2023-08-14 00:00:00+00'::timestamp with time z...

Filter: (status = 'Canceled'::text)

-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..51.66 rows=3524 width=0)

Index Cond: ((scheduled_departure >= '2023-08-13 00:00:00+00'::timestamp with time zone) AND (scheduled_departure <= '2023-08-14 00:00:00+00'::timestamp with time z...

Partial indexes

```
CREATE INDEX flight_canceled ON flight(flight_id)
WHERE status='Canceled';
```

QUERY PLAN

text

Bitmap Heap Scan on flight (cost=60.52..64.53 rows=1 width=71)

Recheck Cond: ((status = 'Canceled'::text) AND (scheduled_departure >= '2023-08-13 00:00:00+00'::timestamp with time zone) AND (scheduled_departure <= '2023-08-14 00:00:00+00'::timestamp with time zone))

-> BitmapAnd (cost=60.52..60.52 rows=1 width=0)

-> Bitmap Index Scan on flight_canceled (cost=0.00..8.60 rows=91 width=0)

-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..51.66 rows=3524 width=0)

Index Cond: ((scheduled_departure >= '2023-08-13 00:00:00+00'::timestamp with time zone) AND (scheduled_departure <= '2023-08-14 00:00:00+00'::timestamp with time zone))

When we should not create an index?

- Generally, you should let the query planner do its job.
- An index is not going to be used when:
 - A small table is completely read into memory.
 - The query uses a large portion of a table.

```
SELECT * FROM pg_catalog.pg_stat_all_indexes
WHERE schemaname='postgres_air'
ORDER BY idx_scan ASC
```

Thank you!