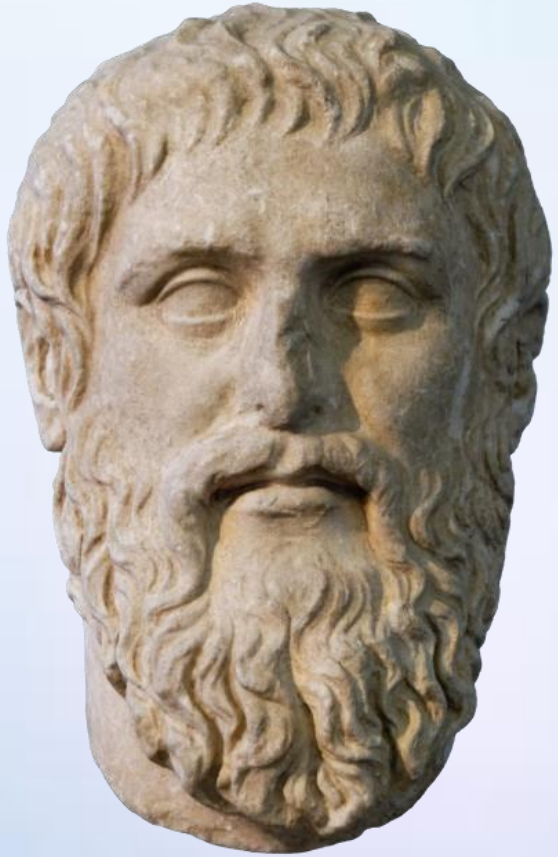
Indexing & Query Execution Planning 2

Deep dive and internals

CODEUS_

“The right question is usually more important than the right answer.”

— Plato



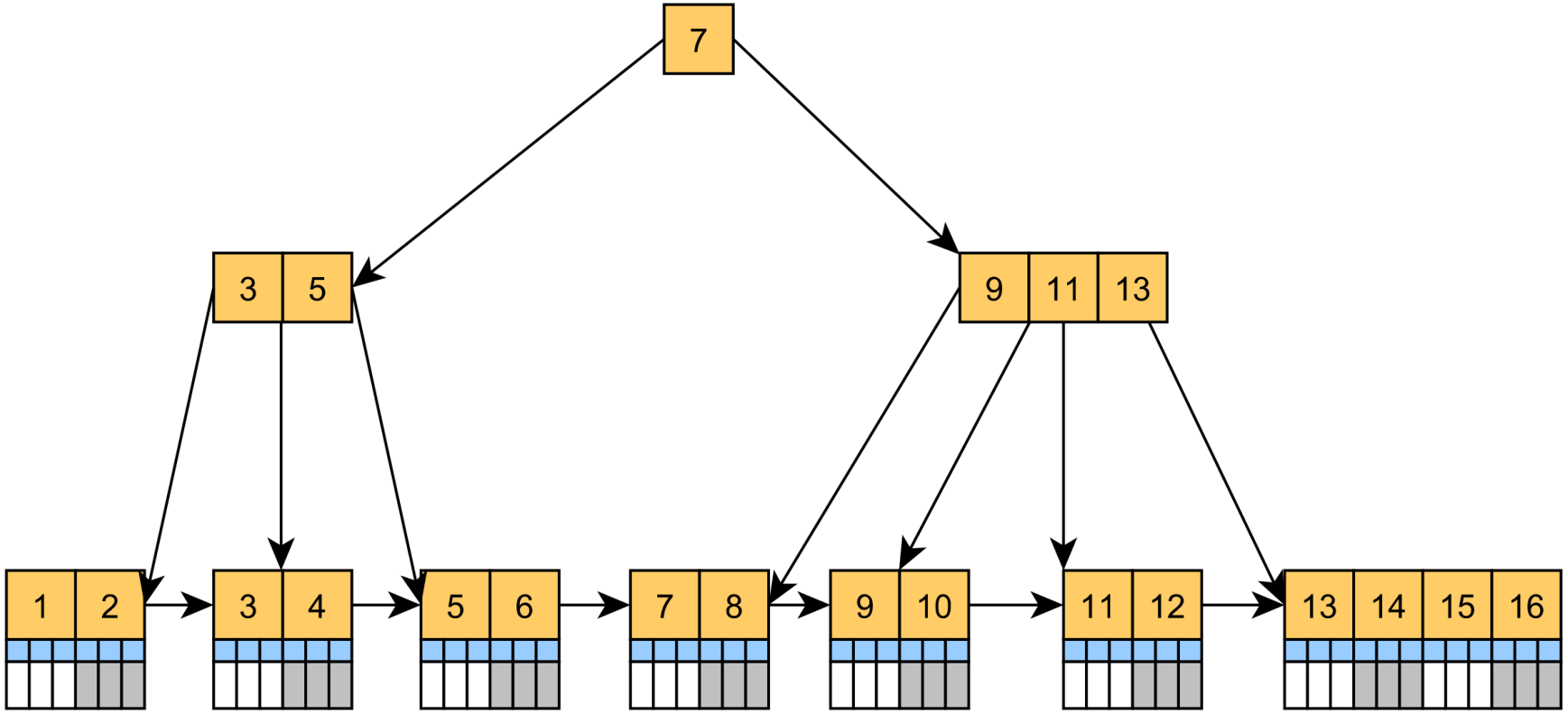
Agenda

- ***Clustered vs non-clustered, internals***
  - *MySQL vs Postgres*
  - *Primary vs secondary index relation*
  - *MVCC*
  - *Postgres Page Layout*
  - *Index Bloat and optimizations*
- ***Execution planning***
  - *Overview*
  - *Scan Method*
  - *Join Method*
  - *Join Order*
  - *Statistics*
- ***Q&A***
- ***Kahoot!***

Clustered vs Non-clustered & Internals

Clustered index

A Clustered Index is basically a tree-organized table.

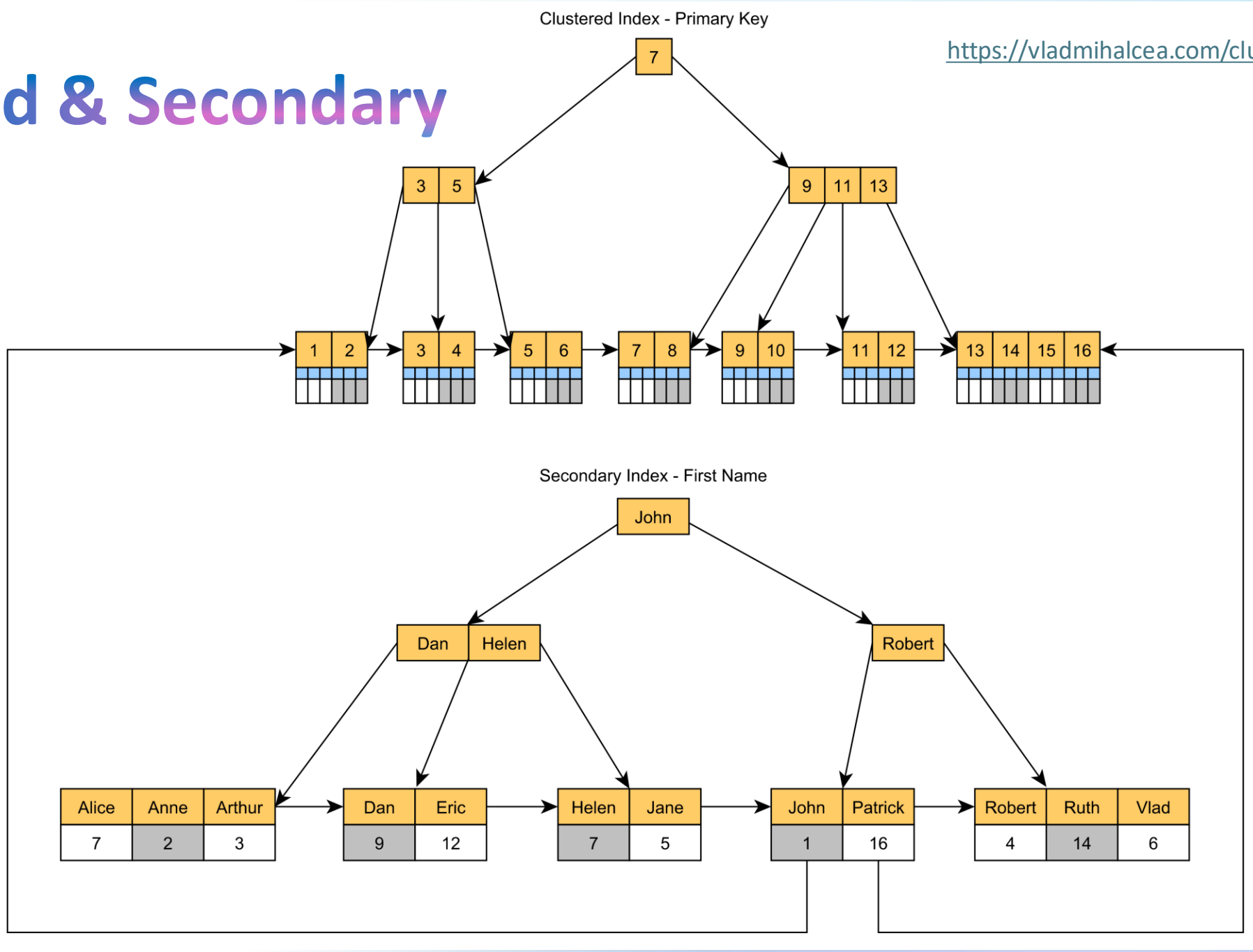


CODEUS

https://vladmihalcea.com/clustered-index/

Clustered & Secondary

https://vladmihalcea.com/clustered-index/



CODEUS_

Clustered index & UUID as primary key

Clustered Indexes sizes

table_name	size_in_mb
products	4311.00
products_uuid_column	4939.00
products_uuid_pk	7888.86
component	0.02
sys_config	0.02

Secondary Indexes sizes

table_name	size_in_mb
products	1923.80
products_uuid_column	2771.78
products_uuid_pk	5557.90

CHAR(32) = 32 bytes
CHAR(36) = 36 bytes
BIGINT = 8 bytes
INT = 4 bytes

It is not only size -> search, insert...

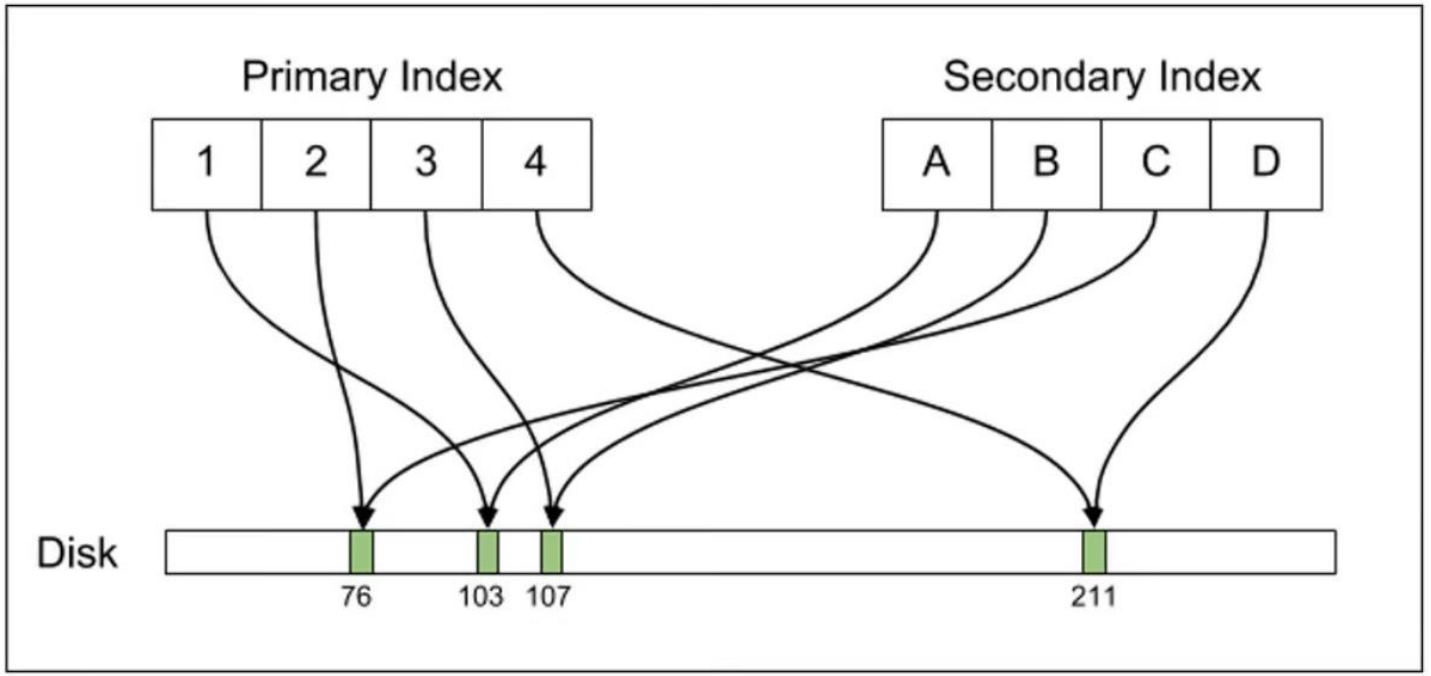
Віктор Турський про програмування
https://youtu.be/Ot7b03Fj_mo?si=3VeYkLx-Fvehiv5A

CODEUS_

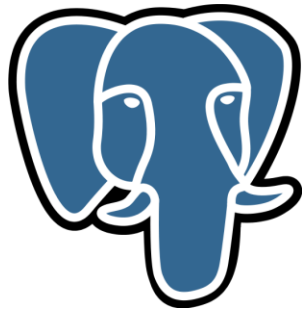
Why Uber Engineering Switched from Postgres to MySQL

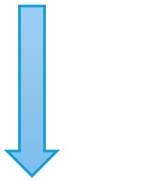
https://www.uber.com/en-UA/blog/postgres-to-mysql-migration/

- **Inefficient architecture for writes**
- **Inefficient data replication**
- Issues with table corruption
- Poor replica MVCC support
- Difficulty upgrading to newer releases



With Postgres, the primary index and secondary indexes all point directly to the on-disk tuple offsets. When a tuple location changes, all indexes must be updated.



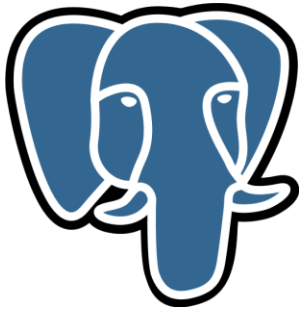


https://www.cs.cmu.edu/~pavlo/blog/2023/04/the-part-of-postgresql-we-hate-the-most.html

Why Uber Engineering Switched from Postgres to MySQL

https://www.uber.com/en-UA/blog/postgres-to-mysql-migration/

Write Amplification



The first problem with Postgres's design is known in other contexts as **write amplification**. Typically, write amplification refers to a problem with writing data to SSD disks: a small logical update (say, writing a few bytes) becomes a much larger, costlier update when translated to the physical layer. The same issue arises in Postgres.

1. Write the new row tuple to the tablespace
2. Update the primary key index to add a record for the new tuple
3. Update the (`first`, `last`) index to add a record for the new tuple
4. Update the `birth_year` index to add a record for the new tuple

In fact, these four updates only reflect the writes made to the main tablespace; each of these writes needs to be reflected in the WAL as well, so the total number of writes on disk is even larger.

This write amplification issue naturally translates into the **replication layer** as well because replication occurs at the level of on-disk changes.

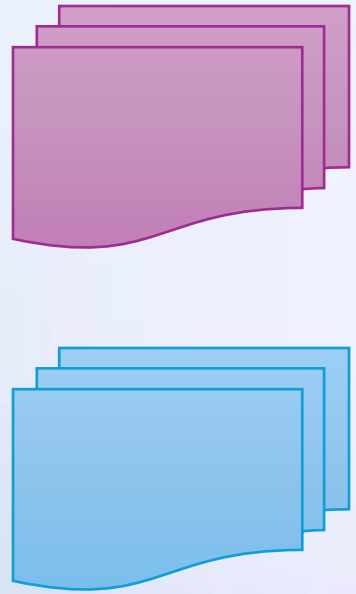
CODEUS

https://www.cs.cmu.edu/~pavlo/blog/2023/04/the-part-of-postgresql-we-hate-the-most.html

What is MVCC?

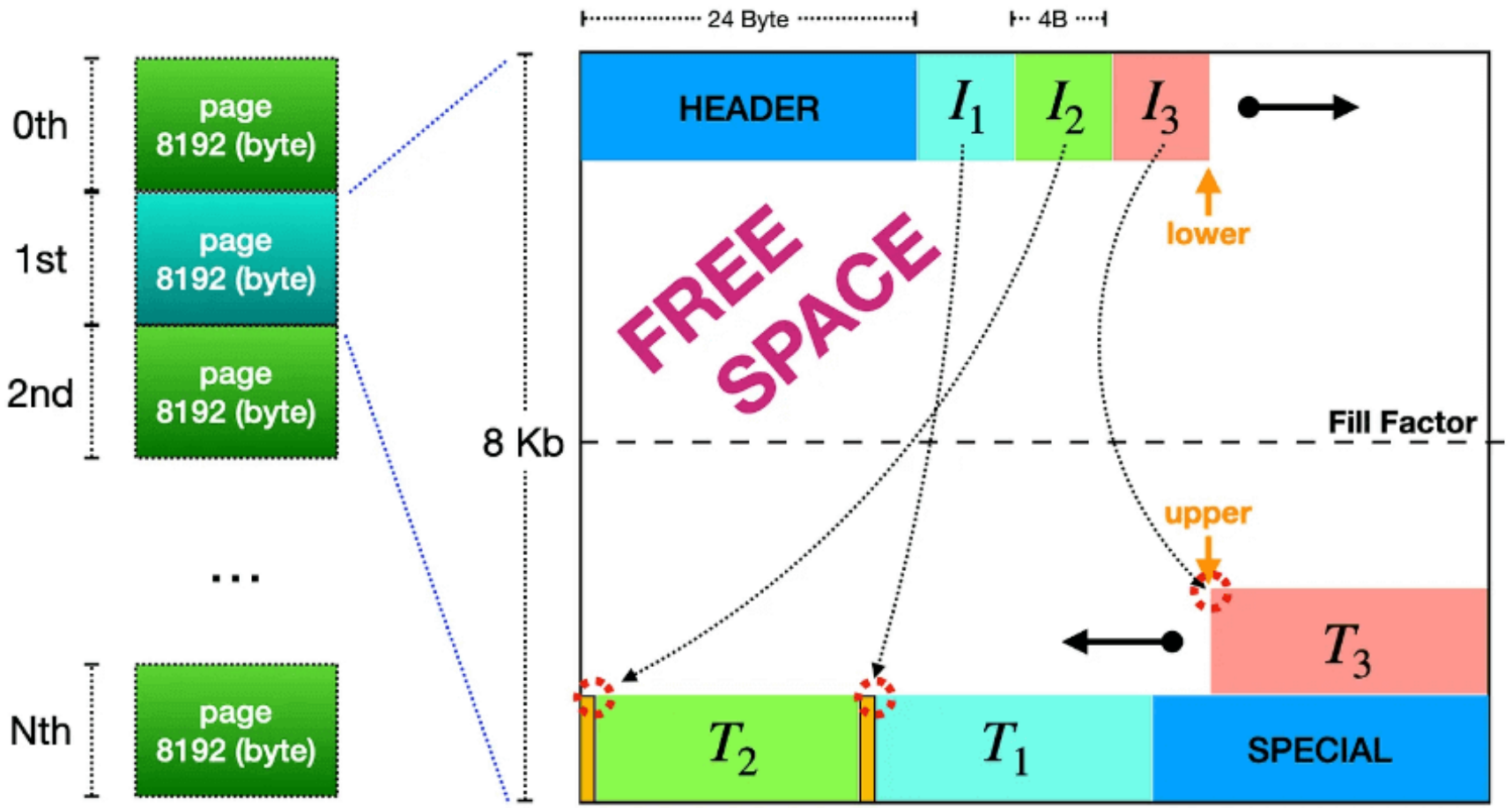
Multiversion Concurrency Control (MVCC) allows Postgres to offer high concurrency even during significant database read/write activity.

"readers never block writers, and writers never block readers".



PostgreSQL Page Layout Overview

When a table is created, a corresponding data file is generated. Within this file, data is organized into fixed-length pages, typically set at 8KB by default. Each page is sequentially numbered starting from 0, known as block numbers. PostgreSQL appends a new empty page at the end of a file when it reaches capacity, That is how PostgreSQL increases the size of a data file.

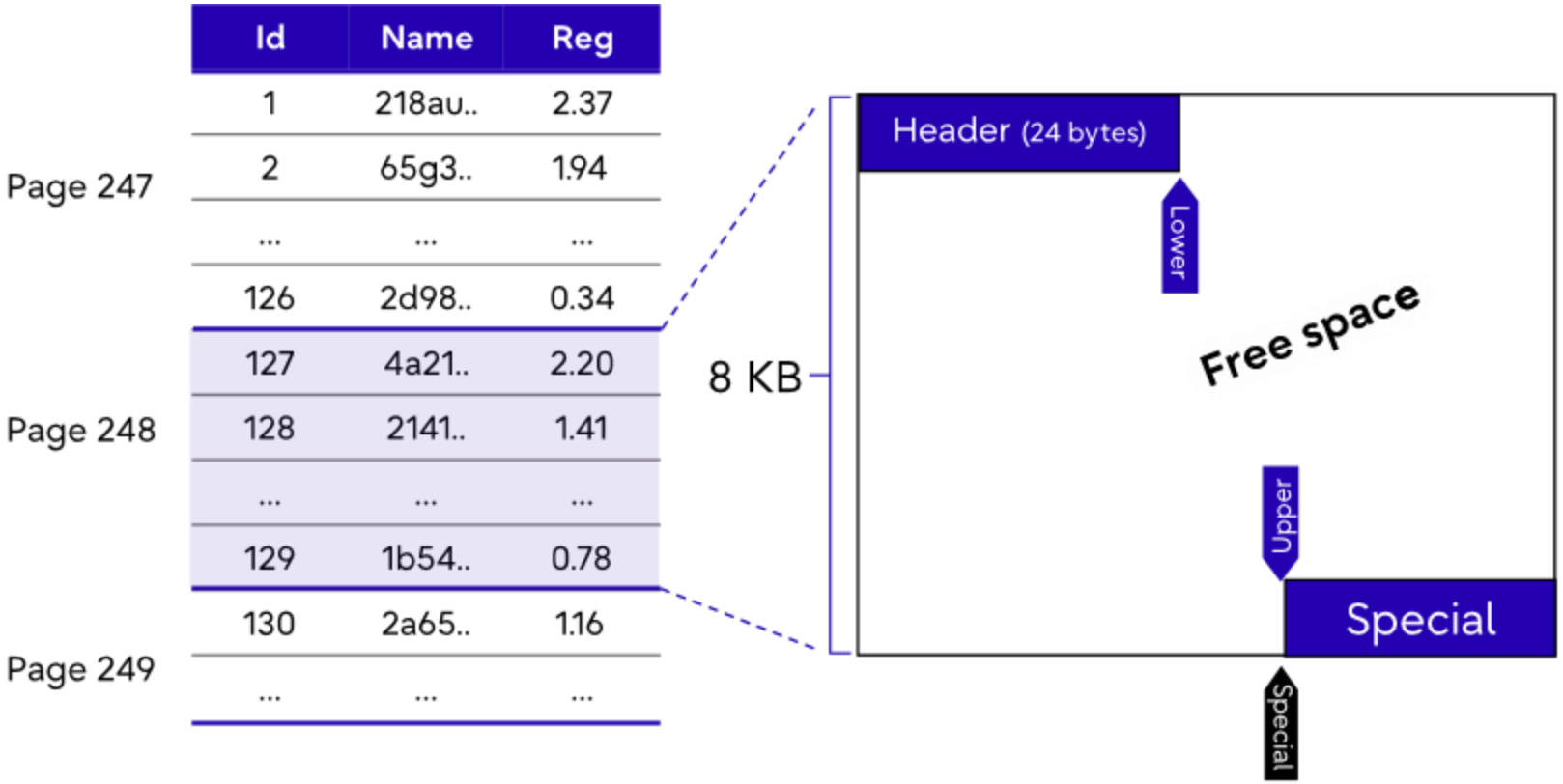


T1.. T3 - tuples

I1.. I3- items

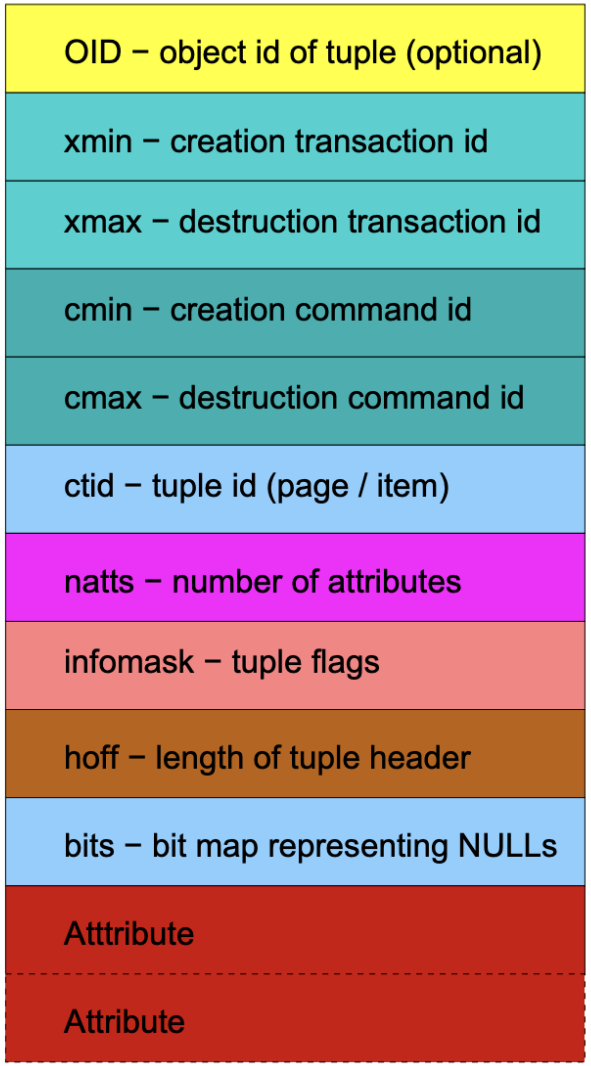
https://stormatics.tech/blogs/postgresql-internals-part-1-understanding-database-cluster-database-and-tables
https://stormatics.tech/blogs/postgresql-internals-part-2-understanding-page-structure

Page structure

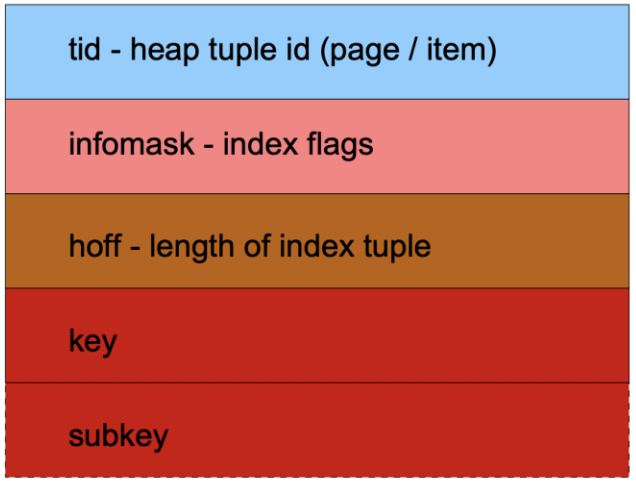


https://www.postgresql.fastware.com/hubfs/Images/Diagrams/img-dgm-anim-postgresql-structure-of-database-row.svg

Heap Tuple Structure



Index Tuple Structure




```c
typedef struct SnapshotData
{
    TransactionId xmin;          /* XID < xmin are visible to me */
    TransactionId xmax;          /* XID >= xmax are invisible to me */
    uint32        xcnt;          /* # of xact below */
    TransactionId *xip;          /* array of xacts in progress */
    ItemPointerData tid;         /* required for Dirty snapshot -:( */
} SnapshotData;
```

https://github.com/postgres/postgres/blob/master/src/include/utils/snapshot.h

Hidden columns

Select ctid, * from transactions;

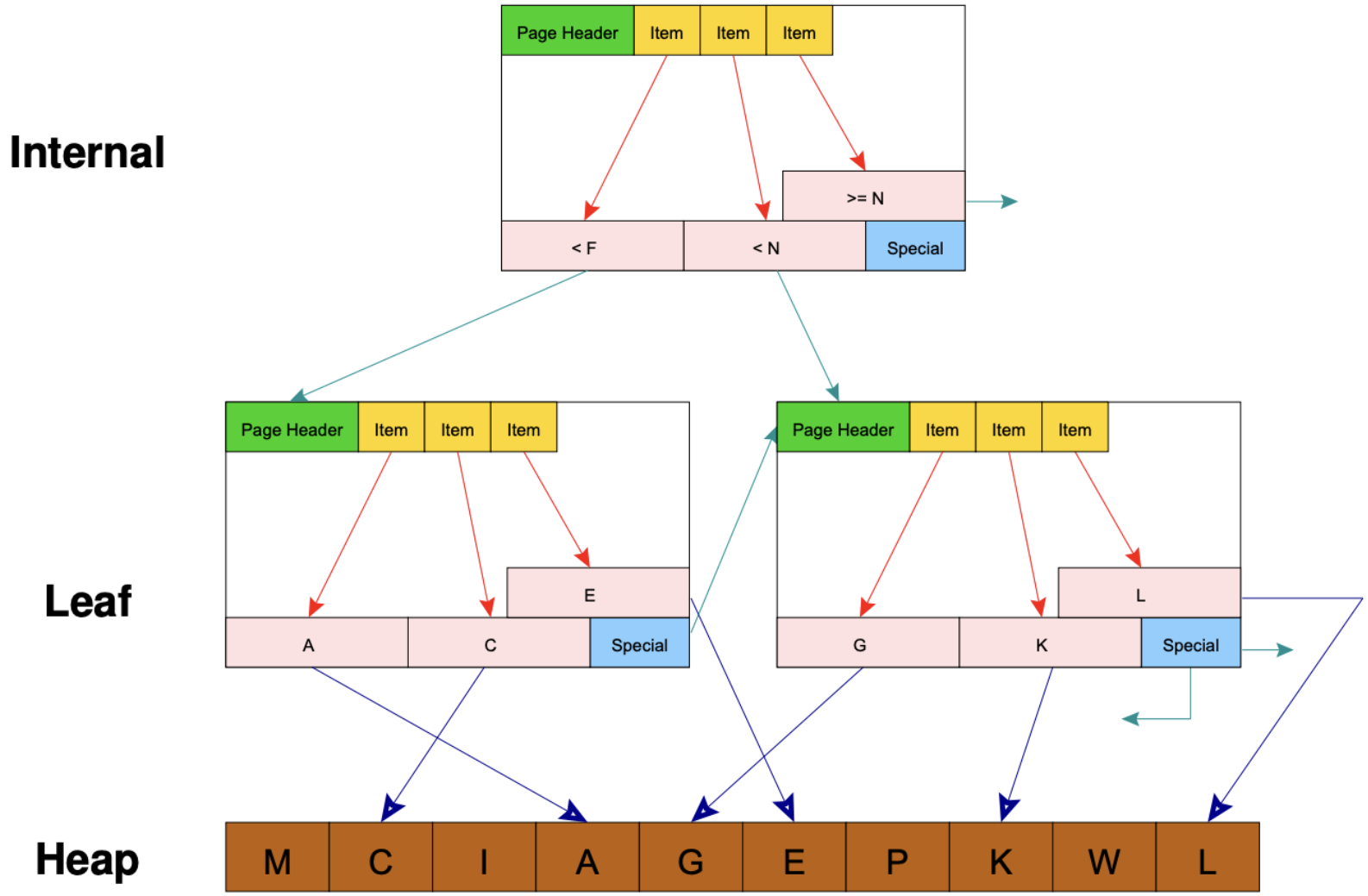
```
5
6  SELECT ctid, * from transactions LIMIT 500;
7
8
```

Data Output | Messages | Notifications

	ctid tid	id [PK] integer	account_id integer	transaction_type character varying (20)	amount numeric (15,2)	transaction_date timestamp without time zone	target_account_id integer
126	(0,126)	126	85145	deposit	37.09	2024-08-05 17:25:35.754268	[null]
127	(0,127)	127	7211	deposit	82.38	2025-04-04 22:13:20.614597	[null]
128	(0,128)	128	19664	deposit	18.82	2025-03-18 09:21:49.238219	[null]
129	(0,129)	129	121502	withdrawal	70.23	2025-03-02 19:06:00.51189	[null]
130	(0,130)	130	120518	deposit	138.08	2025-04-06 16:45:22.435855	[null]
131	(0,131)	131	73988	deposit	31.32	2024-09-22 05:38:10.681579	[null]
132	(1,1)	132	13172	deposit	69.60	2024-05-27 12:38:25.171767	[null]
133	(1,2)	133	112274	deposit	140.03	2024-12-28 16:44:25.295837	[null]
134	(1,3)	134	118558	deposit	100.18	2025-04-07 15:53:23.719526	[null]
135	(1,4)	135	164970	deposit	165.44	2025-03-21 03:41:55.274781	[null]
136	(1,5)	136	61973	deposit	130.98	2025-02-25 04:40:28.268546	[null]
137	(1,6)	137	93901	deposit	210.35	2025-04-07 12:11:08.16502	[null]

Index Page structure

The B-tree index is also multi branched, where each page of the index contains hundreds of pointers to either another index node or to row data, keeping the number of levels very small. On average, a **PostgreSQL** B-tree index can index up to 108 million rows of a table in only 3 levels.



https://momjian.us/main/presentations/

Index Bloat. B-Tree index Optimizations

- **HOT (Heap-Only Tuple) updates**
- Index deduplication (Introduced in **PostgreSQL** 13)
- B-Tree page deletion
- Parallel index scans
- Index-only scans
- Covered indexes
- Bottom-up Index Deletion

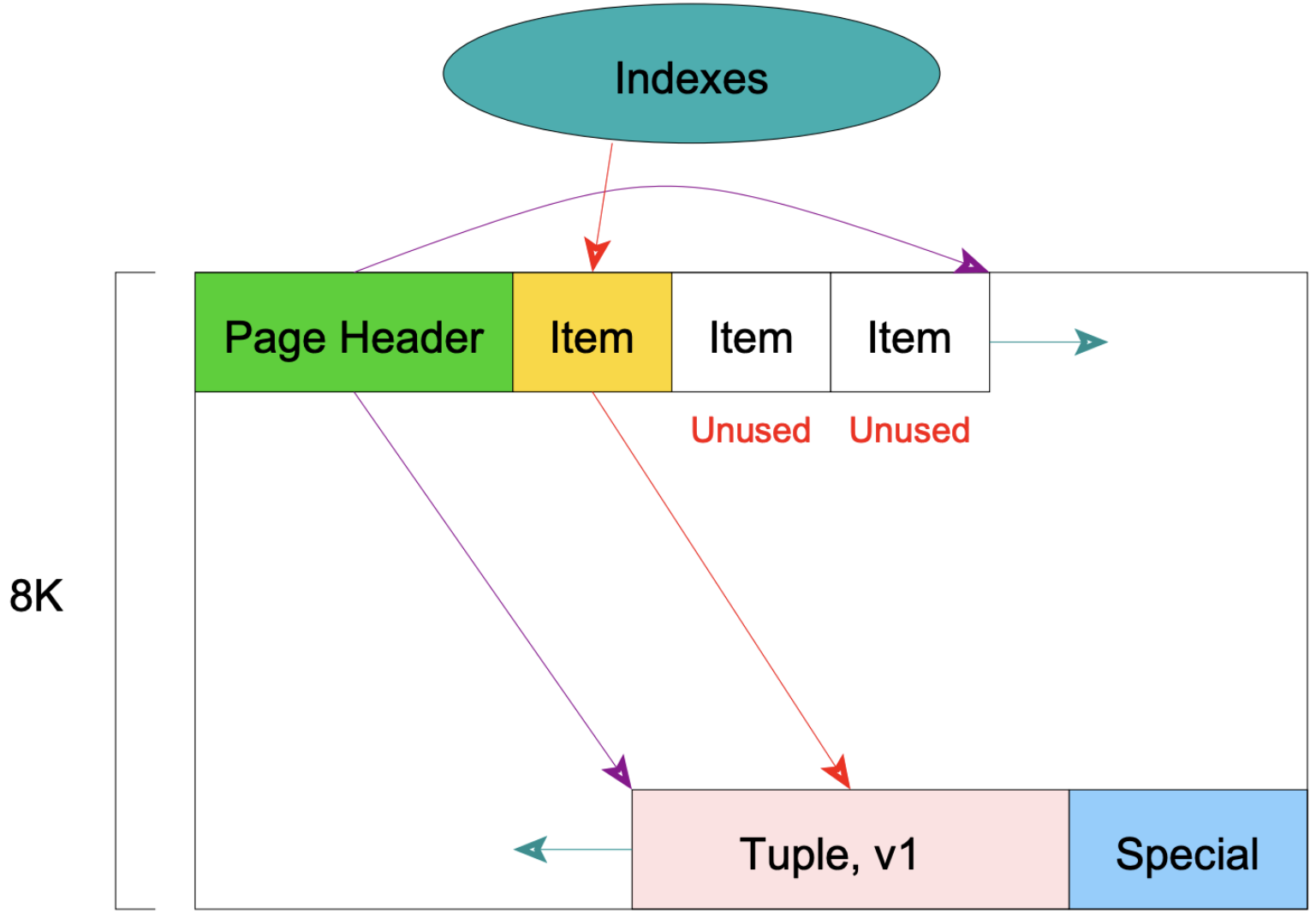
The B-tree index is also multi branched, where each page of the index contains hundreds of pointers to either another index node or to row data, keeping the number of levels very small. On average, a **PostgreSQL** B-tree index can index up to **108 million** rows of a table in only **3 levels**.

https://www.postgresql.org/docs/16/btree-implementation.html

https://www.postgresql.org/docs/16/storage-hot.html

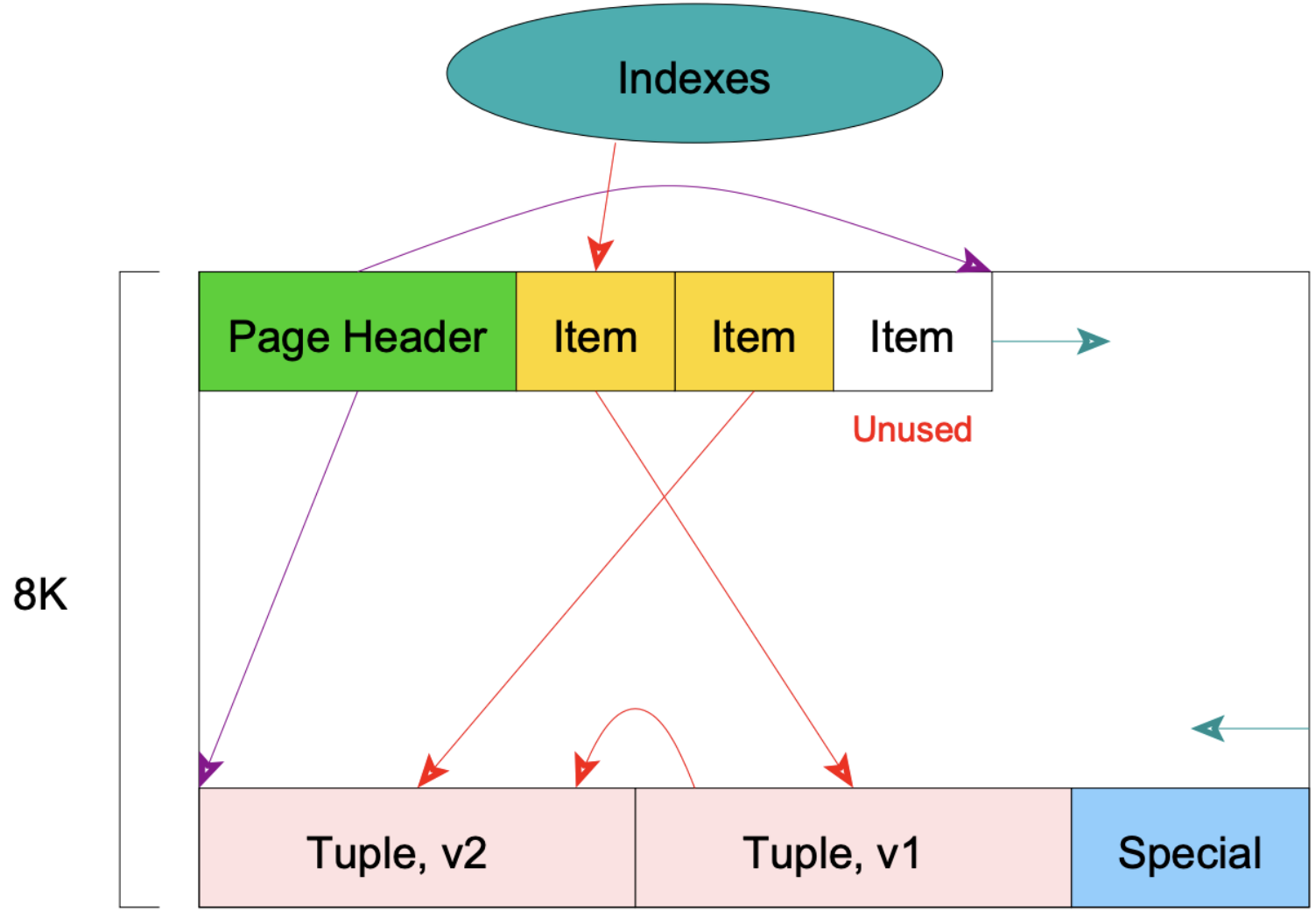
https://www.postgresql.fastware.com/trunk-line/2025-02-postgresql-btree-index-optimizations?hs_preview=hBZfLhBk-185954821917

Initial Single-Row State



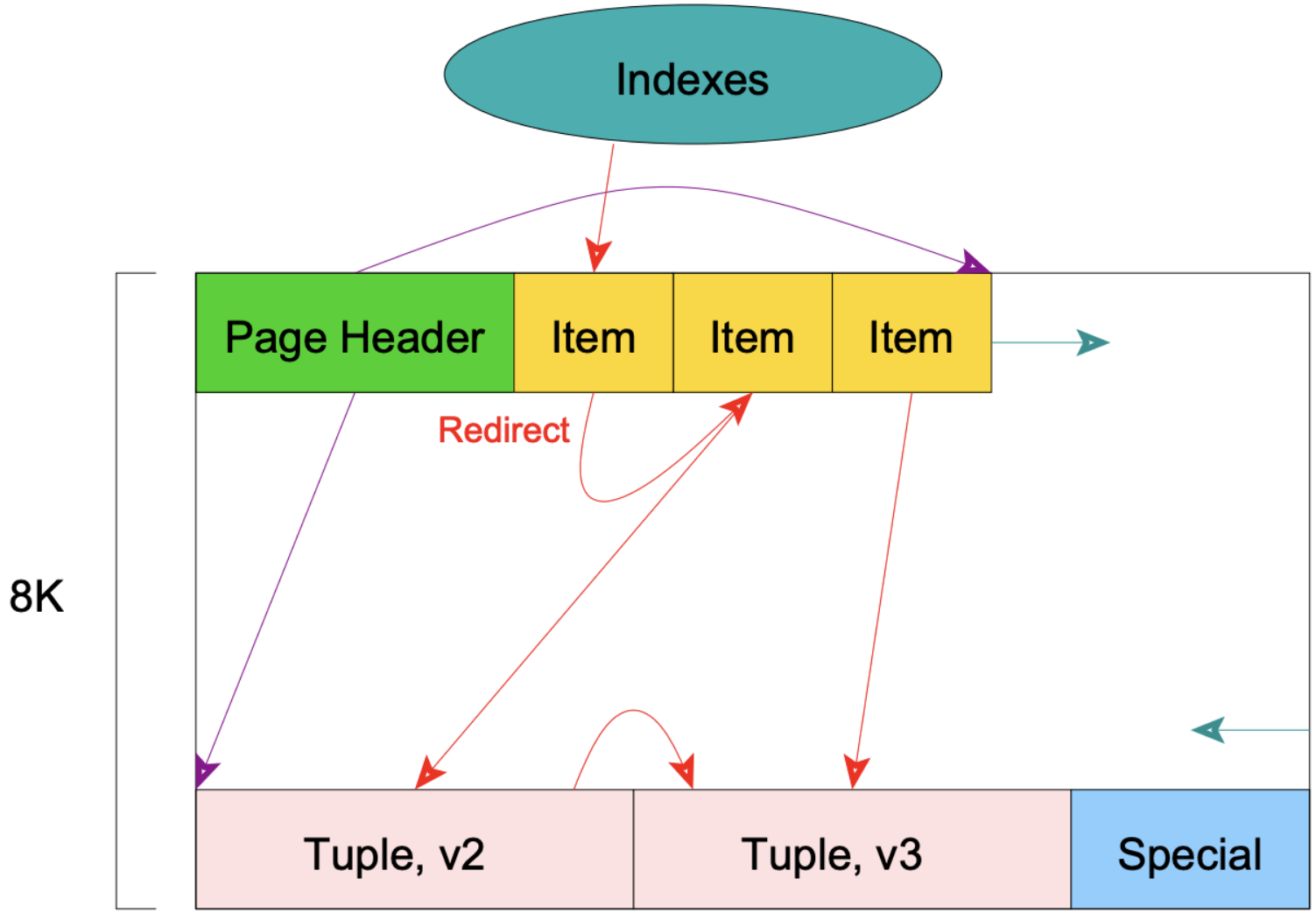
https://momjian.us/main/presentations/

Updates Add a New Row (HOT optimization)



No index entry added because indexes only point to the head of the HOT chain.

Redirect Allows Indexes To Remain Valid



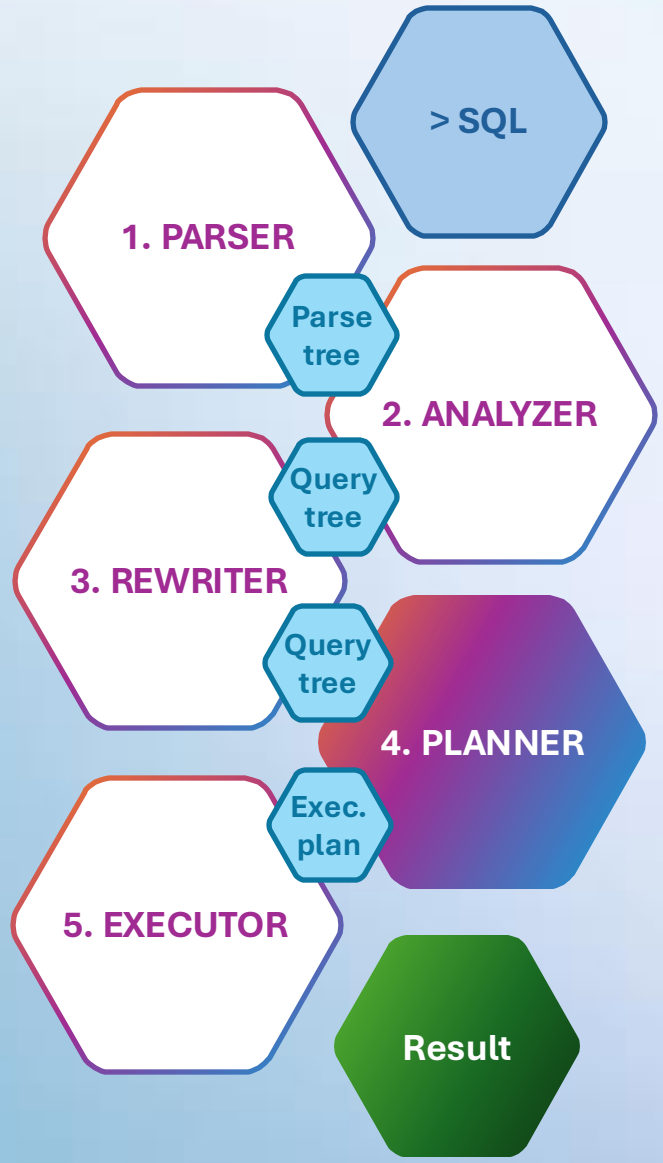
https://momjian.us/main/presentations/

Index types

- B-Tree
- Hash
- BRIN - block range index
- GIN - Generalized Inverted Index
- GiST - Generalized Search Tree
- SP-GiST - Space-Partitioned Generalized Search Tree

Execution planning

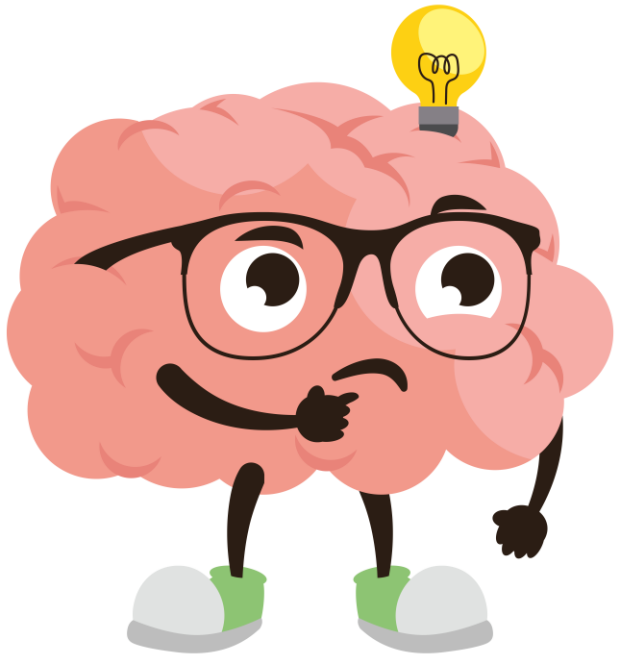
The 5 Query Processing Stages



1. PARSER - Responsible for checking literal syntax errors. Generate **parse tree**. Tokens -> Tree (grammar)
2. ANALYSER - Responsible for deeper syntax analysis. Access the designated database. Check if designated table exists. Check the correctness of data formats. Convert table names to internal OIDs. Generate **query tree**.
3. REWRTITER - The main purpose of Rewriter is to rewrite and optimize the query tree output by the analyzer.
4. PLANNER - Responsible for generating an **execution plan.**
5. EXECUTOR - Main purpose of executor it to run the execution plan generated by the planner. Return **results**

https://www.highgo.ca/2024/01/26/a-comprehensive-overview-of-postgresql-query-processing-stages/

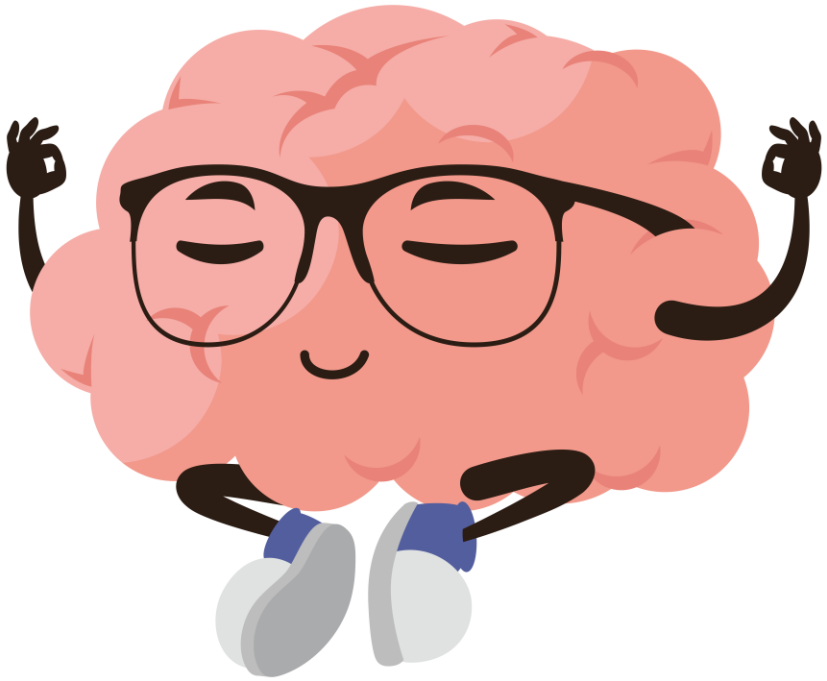
PLANNER



- identify sub query, partitioned table, foreign table, joins…etc
- estimate the sizes of all tables involved with help of table access method
- identify all possible paths to complete the query
  - sequential scan, index, scan, tid, parallel worker…etc
- out of all the paths, find the best one, normally the cheapest
- make a plan out of it.

https://www.highgo.ca/2024/03/22/understand-postgresqls-planner-simple-scan-paths-vs-plans/

PLANNER

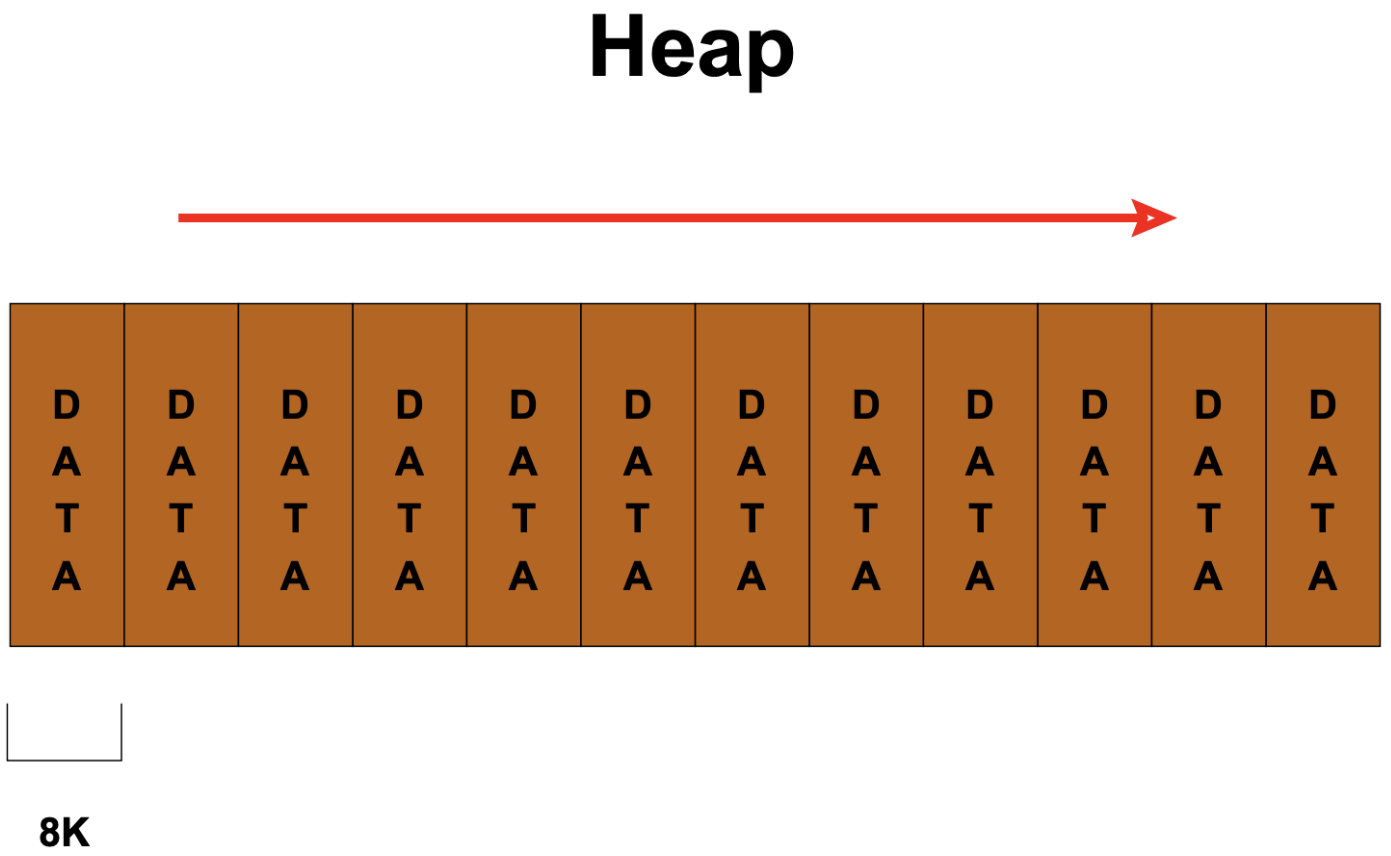


- Scan Method
- Join Method
- Join Order

Which scan method?

- Sequential Scan
- Bitmap Index Scan
- Index Scan
- Index Only Scan

Sequential Scan



https://momjian.us/main/presentations/

Bitmap Index Scan

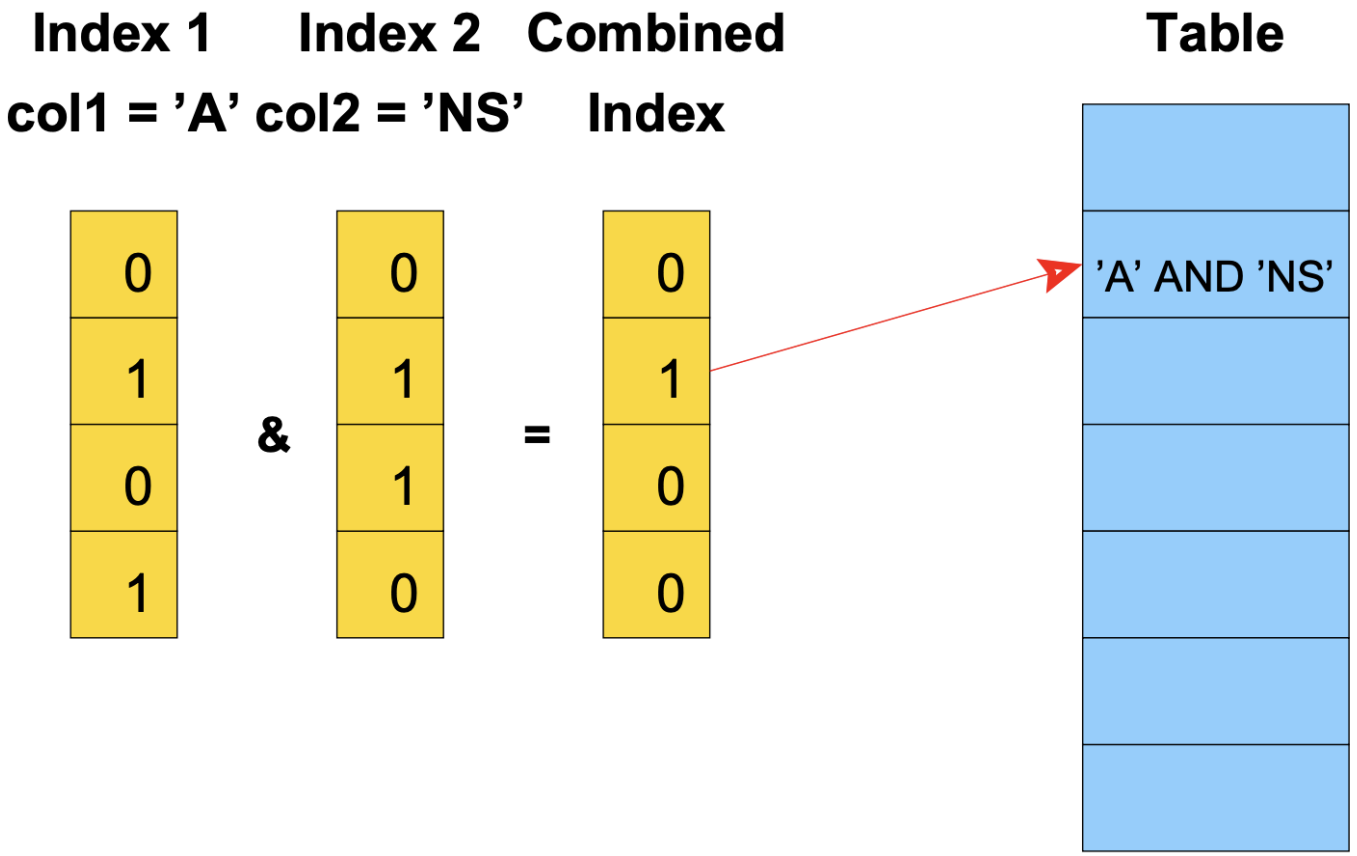
- The planner chooses this index scan method when the query asks for a large enough amount of data that can leverage the benefits of the bulk read, like the sequential scan, but not that large that actually requires processing ALL the table. We can think of the **Bitmap Index Scan** as something between the Sequential and Index Scan.
- The **Bitmap Index Scan** always works in pair with a **Bitmap Heap Scan**; the first scan the index to find all the suitable row locations and builds a bitmap, then the second use that bitmap to scan the heap pages one by one and gather the rows

```
Bitmap Heap Scan on transactions  (cost=2447.47..11645.18 rows=130457 width=12) (actual time=29.154..74.158 rows=134064 loops=1)
  Recheck Cond: (amount > '150'::numeric)
  Heap Blocks: exact=7567
  ->  Bitmap Index Scan on transaction_amount_idx  (cost=0.00..2414.85 rows=130457 width=0) (actual time=27.532..27.533 rows=134064 loop...
        Index Cond: (amount > '150'::numeric)
Planning Time: 0.219 ms
Execution Time: 81.909 ms
```

CODEUS

https://www.percona.com/blog/one-index-three-different-postgresql-scan-types-bitmap-index-and-index-only/

Bitmap Index Scan



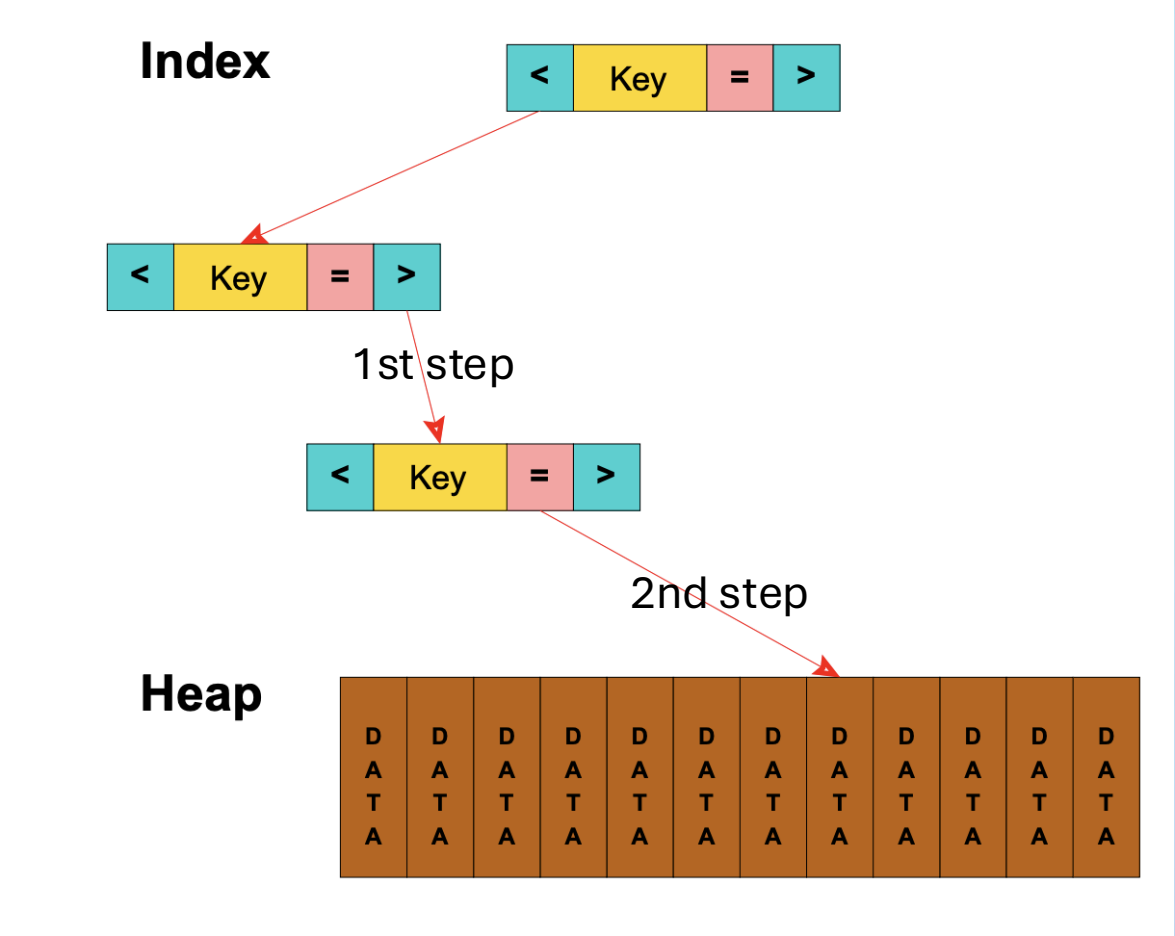
https://momjian.us/main/presentations/

Index Scan

The **Index Scan** consists of two steps:

- the first is to get the row location from the index
- the second is to gather the actual data from the heap or table pages.

So, every **Index Scan** access is two read operations. But still, this is one of the most efficient ways of retrieving data from a table.



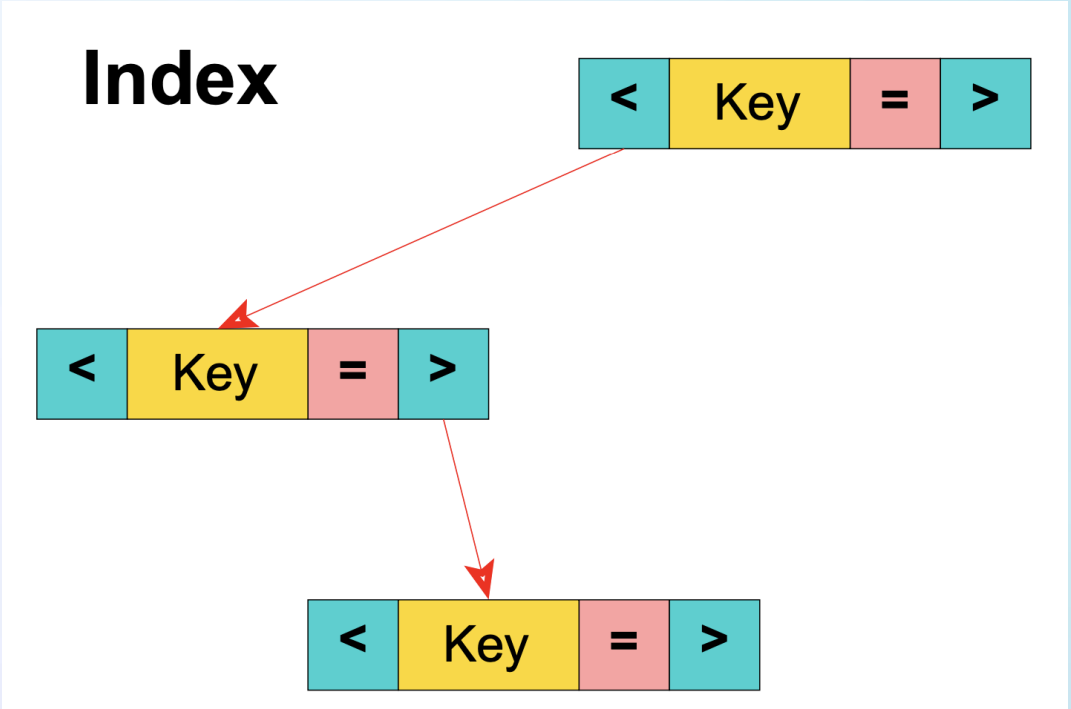
https://www.percona.com/blog/one-index-three-different-postgresql-scan-types-bitmap-index-and-index-only/

CODEUS_

Index-only Scan

This is a really good approach that PostgreSQL uses to improve the *standard* **Index Scan** method.

See the EXPLAIN output now says **Index Only Scan**, and also, it confirms there was no access to the heap (table pages) with the line **Heap Fetches: 0**



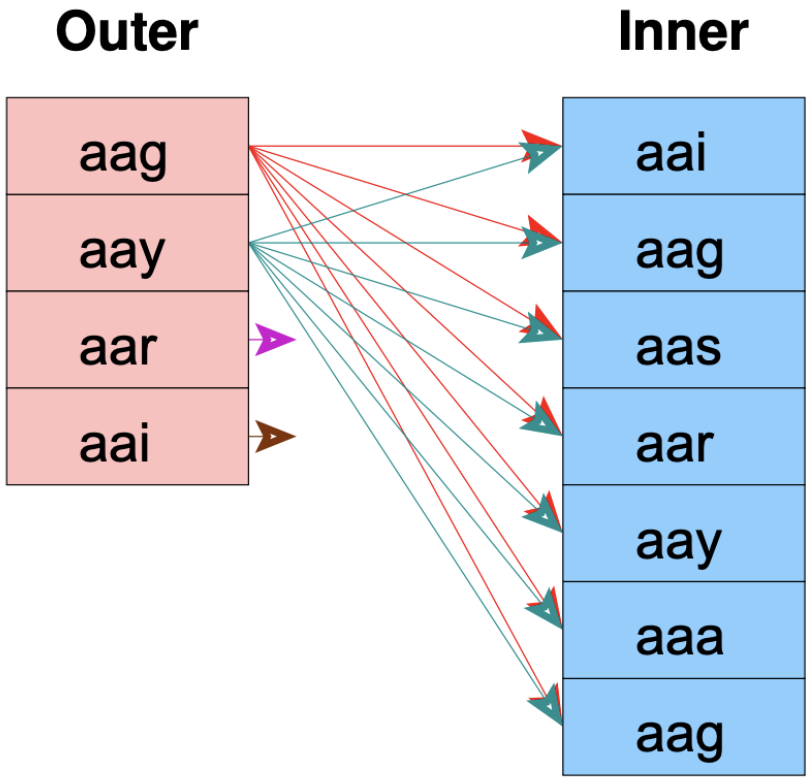

```
Index Only Scan using transaction_amount_covering_idx on transactions  (cost=0.42..4871.42 rows=130457 width=12)
  Index Cond: (amount > '150'::numeric)
  Heap Fetches: 0
Planning Time: 0.085 ms
Execution Time: 38.894 ms
```

https://www.percona.com/blog/leverage-index-only-scans-in-postgresql/

Which Join method?

- Nested loop
  - With Inner Sequential Scan
  - With Inner Index Scan
- Merge join
- Hash join

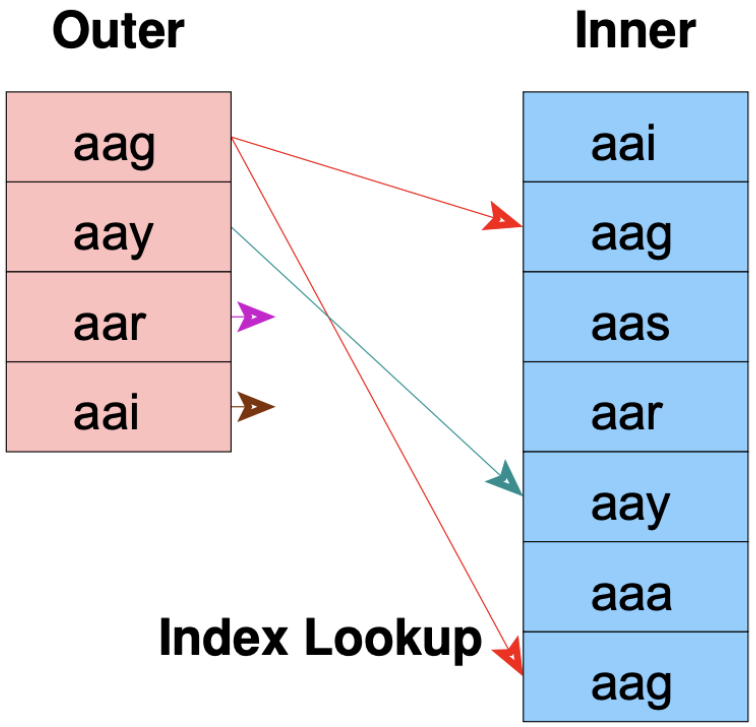
Nested Loop With Inner Sequential Scan



No Setup Required

Used For Small Tables

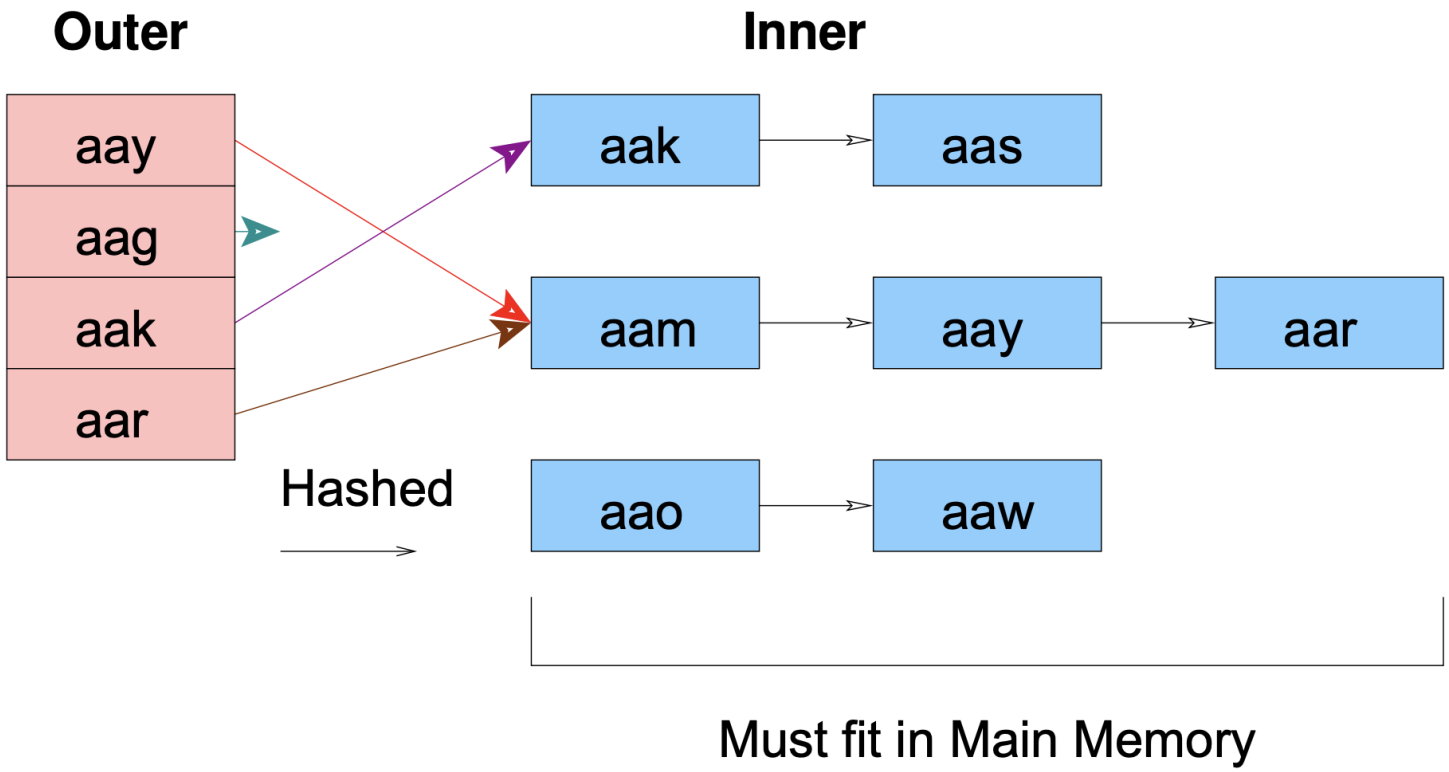
Nested Loop with Inner Index Scan



No Setup Required

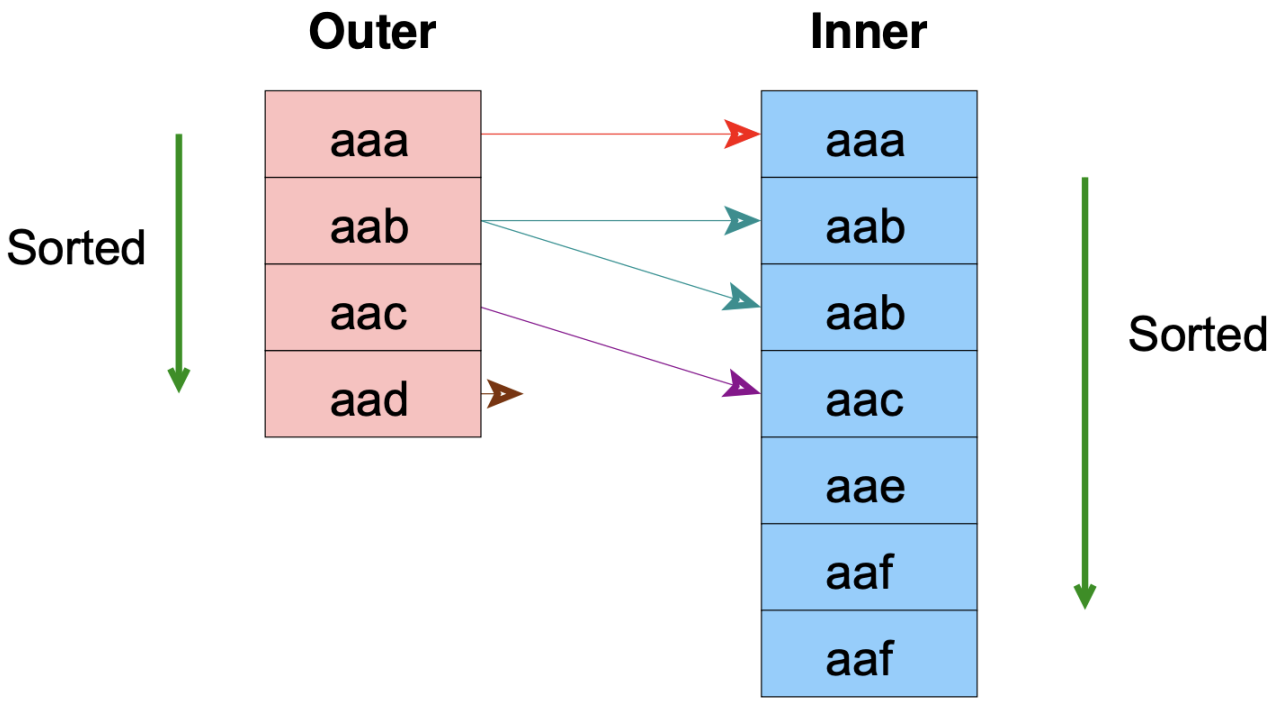
Index Must Already Exist

Hash Join



https://momjian.us/main/presentations/

Merge Join



Ideal for Large Tables

An Index Can Be Used to Eliminate the Sort

What if too many options?

- Among all relational operators the most difficult one to process and optimize is the join. The number of alternative plans to answer a query grows exponentially with the number of joins included in it. Further optimization effort is caused by the support of a variety of *join methods* (e.g., nested loop, hash join, merge join in PostgreSQL) to process individual joins and a diversity of *indexes* (e.g., B-tree, hash, GiST and GIN in PostgreSQL) as access paths for relations.

Planning Time > Execution Time

Genetic algorithms solves the join ordering problem

https://www.postgresql.org/docs/current/geqo.html

We can force index scan or specific join type

- SET enable_seqscan = false;
- SET enable_bitmapscan = false;

- SET enable_hashjoin = false

https://www.postgresql.org/docs/current/runtime-config-query.html#RUNTIME-CONFIG-QUERY-CONSTANTS

https://www.postgresql.org/docs/current/performance-tips.html

When to create indexes?

- pg_stat_user_tables.seq_scan is high
- Check frequently-executed queries with EXPLAIN (find via pg_stat_statements or pgbadger)
- Squential scans are not always bad
- If pg_stat_user_indexes.idx_scan is low, the index might be unnecessary
- Unnecessary indexes use storage space and slow down INSERTs and some UPDATEs

https://www.postgresql.org/docs/current/monitoring-stats.html

https://momjian.us/main/presentations/

Statistics

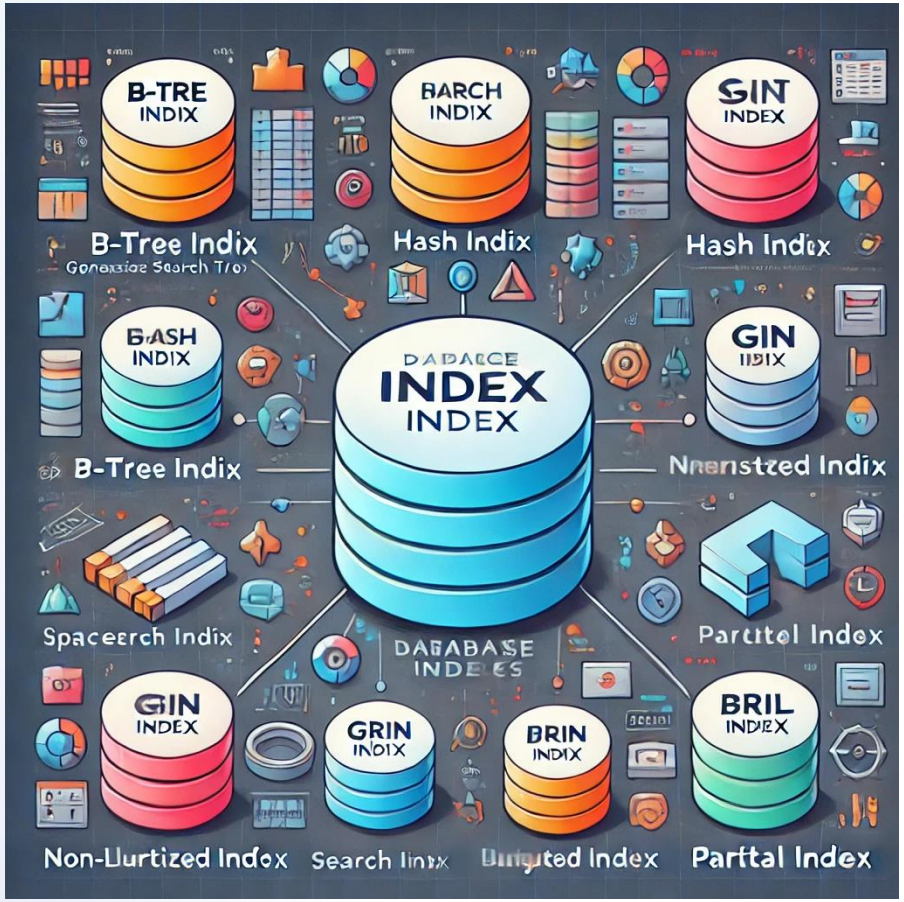
```
SELECT *
FROM pg_stats
WHERE tablename = 'transactions';
```

Data Output | Messages | Notifications

	schemaname name	tablename name	attname name	inherited boolean	null_frac real	avg_width integer	n_distinct real	most_common_vals anyarray	most_common_freqs real[]	histogram_ anyarray
1	public	transactions	transaction_date	false	0	8	-0.997816	[null]	[null]	{"2024-04-(
2	public	transactions	target_account_id	false	0.9007	4	62039	[null]	[null]	{29,1908,3
3	public	transactions	account_id	false	0	4	-0.139785	[null]	[null]	{9,1885,35
4	public	transactions	transaction_type	false	0	8	3	{deposit,transfer,withdrawal}	{0.80193335,0.0993,0.09876667}	[null]
5	public	transactions	id	false	0	4	-1	[null]	[null]	{119,1048{
6	public	transactions	amount	false	0	6	19860	{20.49,24.92,43.50,0.70,2.60,22....	{0.0003,0.0003,0.0003,0.0002666(	{0.00,1.27,:

Evaluating Index Types

- Index build time
- Index storage size
- INSERT/UPDATE overhead
- Access speed
- Operator lookup flexibility



https://momjian.us/main/presentations/

Indexes & locks & Isolation levels & on-disk structures

- What is locked when index is creating?
- Table locks:
  - SHARE (**ShareLock**) - *CREATE INDEX* (without *CONCURRENTLY*).
  - SHARE UPDATE EXCLUSIVE (**ShareUpdateExclusiveLock**) - *CREATE INDEX CONCURRENTLY*, *REINDEX CONCURRENTLY*
- Row locks?

https://www.postgresql.org/docs/current/explicit-locking.html

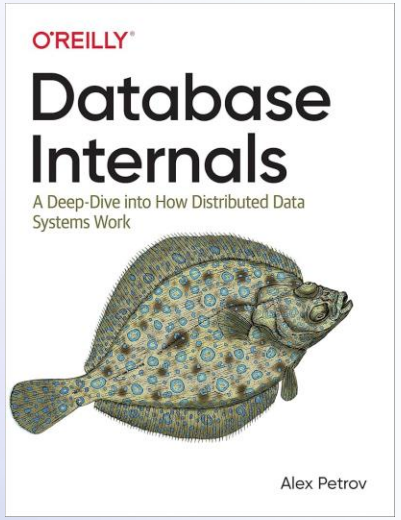
- Why does it matter?
  - Deadlock between primary and secondary index X record locks (MySQL)
  - Deadlock between Shared(created by MySQL to maintain foreign key integrity during INSERT) and Exclusive record lock created implicitly by UPDATE.

Resources

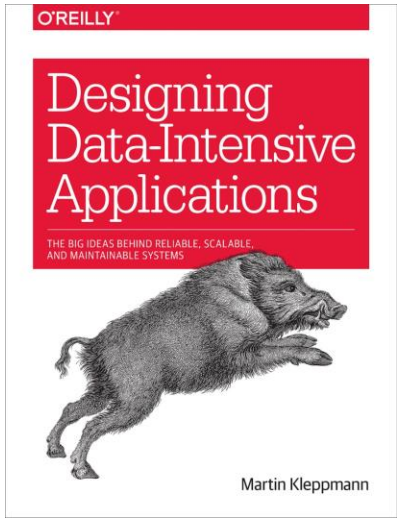
How MySQL and PostgreSQL handle concurrency under the hood

https://notes.andywu.tw/en/2022/how-mysql-and-postgresql-handle-concurrency-under-the-hood/

Database Internals
By Alex Petrov



Designing Data-Intensive Applications
By Martin Kleppman
Chapter 3. Storage and retrieval
Chapter 7. Transactions



CODEUS


THINK LIKE A PLANNER
Q & A
CODEUS

Thank you

- Author: Bohdana Sherstyniuk
- My LinkedIn: https://www.linkedin.com/in/bohdana-sherstyniuk-2826b317b
- Date: April 2024
- Join Codeus community in Discord
- Join Codeus community in LinkedIn