Indexes in PostgreSQL

Overview of indexing in PostgreSQL database

Agenda

- Intro
  - Overview
  - Types of indexes
  - MySQL differences
  - B*Tree in detail
  - Indexing for query tuning
  - Functional & partial indexes
  - Other index types
- Q & A

servicenow

Indexes - overview

Index: an object by which we can retrieve specific rows (data) faster.

- Index is a pointer to data in a table
- Can be created using one or multiple columns
- Stored on disk as a separate object
- Consumes significant disk space for big tables
- Can be unique or non-unique
- Adds overhead for DML operations and query planning
- All indexes in PostgreSQL are secondary indexes
- “An index makes a query fast” still applies in PostgreSQL

Indexes - overview

Index

friends_name_asc	
name	**id**
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

points to

Table

Friends		
id	**name**	**city**
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

servicenow

Indexes - overview

PostgreSQL has **a lot** of different index types available out of the box!

- B-Tree
- GiST
- SP-GiST
- Hash
- BRIN
- GIN

servicenow

Indexes - overview

But most of them are easy to understand:

- **B-Tree** — Most common index type
- **GiST** — Geometry
- **SP-GiST** — GiST extension / ++
- **Hash** — Only equality predicates
- **BRIN** — Fast to build & small
- **GIN** — Text search, arrays, JSON

servicenow

PostgreSQL vs MySQL: differences

servicenow

PostgreSQL vs MySQL - differences

There is **one** important thing we should be aware of, coming from MySQL / MariaDB.

Default table organization in InnoDB:

- In MySQL (InnoDB), each table is organized via Clustered Index
  - Oracle term: IoT (`Index Organized Table`)
  - What it means: data is stored in a B-tree structure, organized by PK
  - Data is sorted by the Primary Key of each row
  - If no PK or UNIQUE index exists, InnoDB will auto-generate a hidden clustered index (`GEN_CLUST_INDEX`)
  - Each secondary index includes the PK + the secondary index columns
  - Significant index size implications for wide PK

servicenow

PostgreSQL vs MySQL - differences

Default table organization in PostgreSQL:

- In PostgreSQL, each table is a heap (same as Oracle)
  - What it means: data is stored unsorted (as a heap object)
  - All indexes are secondary indexes
    - implication: each index is stored separately from the table main data
    - PK of the table is NOT stored with the index
    - Less worries concerning the size / width of the Primary Key
  - Each row retrieval requires fetching data from both the index and the heap
  - Heap-access portion may involve a lot of random I/O
  - Oracle equivalent: `TABLE ACCESS BY INDEX ROWID`

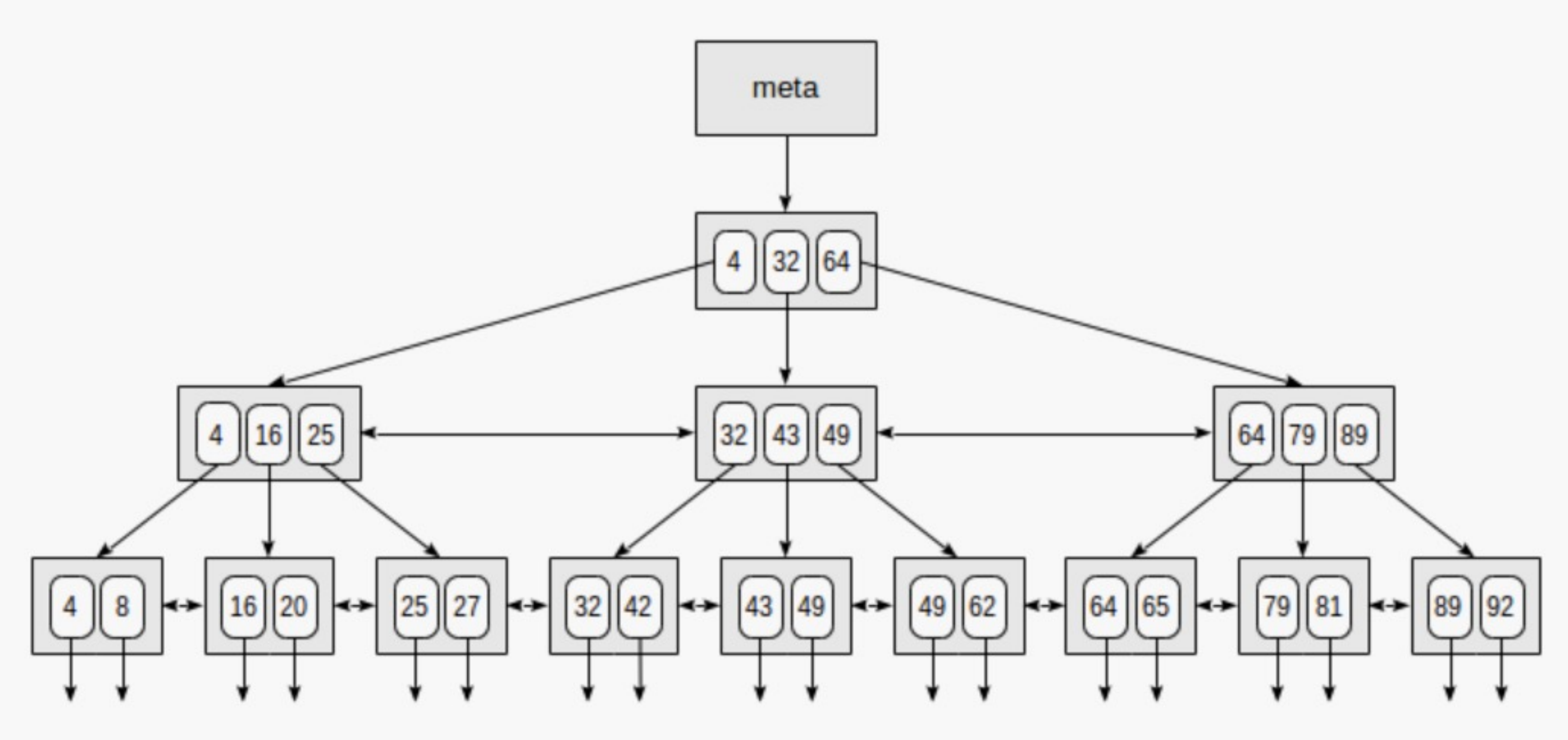
B-Tree indexes

B-Tree: index overview

Main features of B-Tree indexes in PostgreSQL:

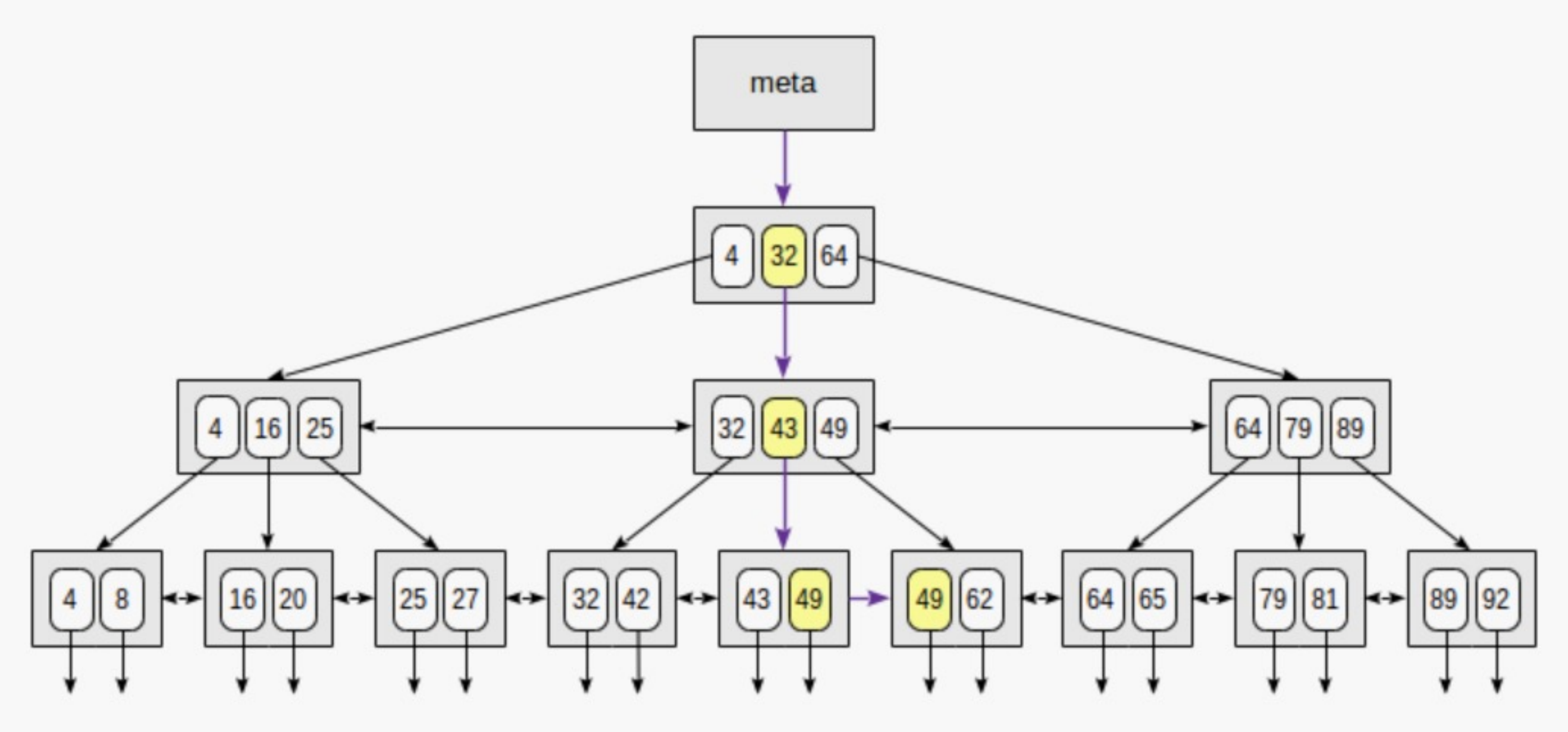
- B-Tree: self-balancing tree data structure
- Balanced = each leaf page separated from root by the same number of internal pages (consistent search time for any value)
- Good for data that can be sorted (e.g. numbers or characters)
- Think: greater >, less <, equal = (but also >= and <=)
- … but also works for: `LIKE`, `ORDER BY`, `GROUP BY`, `JOIN`
- The only index supporting index-only scans
- Index entry deduplication (PostgreSQL 13)

B-Tree: tree structure



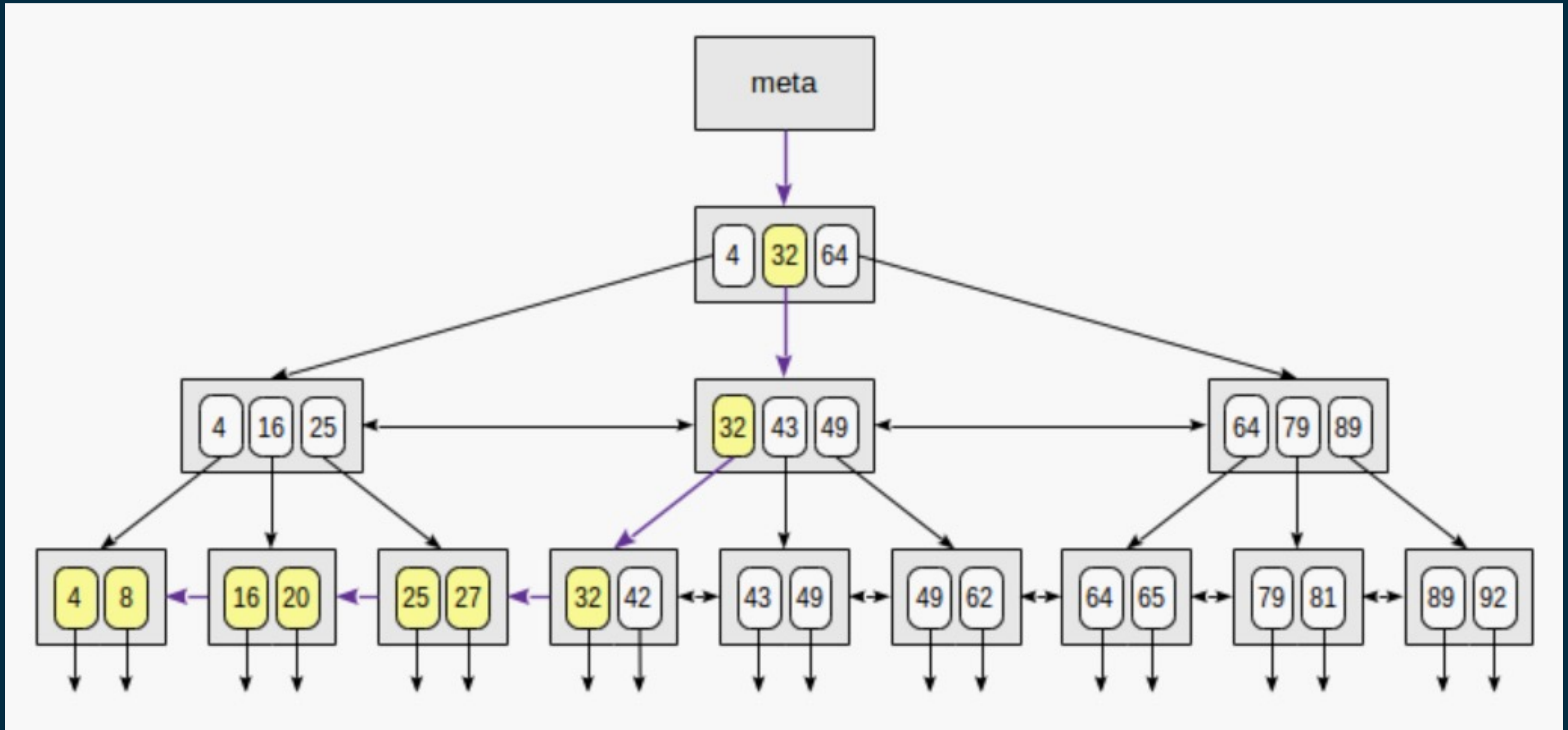
servicenow

B-Tree: equality search



servicenow

B-Tree: range search



servicenow

Indexing for query tuning

servicenow

Indexing for query tuning

Next slides will cover the core rules for efficient indexing in PostgreSQL.

General rules to follow:

- Single-column index for single WHERE predicate
- Composite index for multiple WHERE predicates against a single table
- Range predicates (>, >=, <, <=) can be only used as a last index column
- LIKE only works if specified as: … LIKE (‘bob%’) – will not work for ‘%bob%’
- Indexing (a, b) helps with GROUP BY (a, b)
- Indexing (a, b) helps with ORDER BY (a, b)
- Works for GROUP BY + ORDER BY only if columns match for both
- Helps with JOIN operations (depends on JOIN algorithm used)

servicenow

Indexing for equality

How composite index helps with equality predicates:

- Composite index DDL & example queries:

```
1 create index idx1 on t1 (a, b, c);
2
3 select ... from t1 where a = 3;
4 select ... from t1 where a = 3 and b = 7;
5 select ... from t1 where a = 3 and b = 7 and c = 6;
6 select ... from t1 where c = 2 and b = 7 and a = 1;
7 select ... from t1 where b = 3 and a = 7 and c > 3;
8 select ... from t1 where a = 3 and b = 7 and c = 3 and d = 8;
```

Lines 3-7: index idx1 fully used in all the examples
Line 6: order does not matter if all predicates are equality
Line 8: no filter exists for column d

servicenow

Indexing for ranges

Rule of thumb #1: composite index can be used to cover range predicates ONLY if it's the right-most column of the index.

```
3  # range predicates - index used
4  select ... from t1 where a > 2;
5  select ... from t1 where a = 1 and b > 2;
6  select ... from t1 where a = 1 and b = 2 and c > 7;
7
8  # range predicates - index partially used
9  select ... from t1 where a > 2 and b > 2;
10 select ... from t1 where a = 7 and c < 8;
11 select ... from t1 where a = 1 and b > 2 and c > 7;
12
13 # range predicates - index not used
14 select ... from t1 where b > 2;
15 select ... from t1 where b = 2 and c < 7;
16 select ... from t1 where c = 6 and b > 2;
```

Line 9: index used for a, not used for b
Line 11: index used for a and b, not used for c
Lines 14-16: no predicate against a, index can't be used

Indexing for LIKE

Rule of thumb #2: treat LIKE ‘abc%’ similar to how you would treat a range scan.

```
1  # LIKE predicate is similar to range
2
3  psql> explain select * from bears where name like 'bob%';
4
5                                   QUERY PLAN
6  --------------------------------------------------------------------------------
7   Index Scan using nfw on bears  (cost=0.43..4.45 rows=1 width=25)
8     Index Cond: (((name)::text >= 'bob'::text) AND ((name)::text < 'boc'::text))
9     Filter: ((name)::text ~~ 'bob%'::text)
10 (3 rows)
```

Line 3: bears table has 5 columns: id, name, fur, birth, weight
Line 7: index NFW covers (name, fur, weight)
Line 8: LIKE is converted into (name >= ‘bob’ and name < ‘boc’)

Indexing for LIKE

Rule of thumb #3: all the indexing benefits are lost if we use '%bob%' instead of 'bob%':

```
1 # LIKE doesn't work with index if double % is used
2
3 psql> explain select * from bears where name like '%bob%';
4
5                         QUERY PLAN
6 ----------------------------------------------------------
7  Seq Scan on bears  (cost=0.00..21488.20 rows=1 width=25)
8    Filter: ((name)::text ~~ '%bob%'::text)
9 (2 rows)
```

Line 3: predicate 'bob%' replaced with '%bob%'
Line 7: sequential scan on bears table instead of an index
Line 7: cost skyrockets to 21488

servicenow

Indexing for GROUP BY & ORDER BY

Rule of thumb #4: composite index on (a, b, c) will help with GROUP BY & ORDER BY operations on indexed columns:

```
1  SELECT ... FROM t1 ... GROUP BY a;
2  SELECT ... FROM t1 ... GROUP BY a, b;
3  SELECT ... FROM t1 ... GROUP BY a, b, c;
4
5  SELECT ... FROM t1 ... ORDER BY a;
6  SELECT ... FROM t1 ... ORDER BY a, b;
7  SELECT ... FROM t1 ... ORDER BY a, b, c;
8
9  # but also!
10 SELECT ... FROM t1 ... GROUP BY a ORDER BY a;
11 SELECT ... FROM t1 ... GROUP BY a, b ORDER BY a, b;
12 SELECT ... FROM t1 ... GROUP BY a, b, c ORDER BY a, b, c;
```

Lines 1-3: GROUP BY matching column list, order matters
Lines 5-7: ORDER BY matching column list, order matters
Line 10-12: GROUP BY and ORDER BY combined, order matters
Note: ORDER BY needs to be left-side subset of GROUP BY

Functional & partial indexes

servicenow

Functional indexes

Functional index: index based on a result of a function, applied to one or more columns in the table.

- Simple example:

```
1 SELECT * FROM t1 WHERE lower(col1) = 'value';
2 CREATE INDEX idx1 ON t1 (lower(col1));
```

- More complex example:

```
4 SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
5 CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

Line 1: lower function would make regular index invalid for this query
Line 4: string concatenation – basic index won’t work
Typical examples: lower(), upper(), trim(), length(), substr()

Partial indexes

Partial index: index build on a subset of a table.

- Defined by a conditional expression (partial index predicate)
- Contains entries only for rows that satisfy the predicate
- Good use case: avoid indexing common / popular values
  - Job queue – no need to index completed jobs
  - Application processing system – index only 'in progress' applications
- example:

```sql
CREATE INDEX idx_partial
ON task(sys_created_on)
WHERE active = 1;
```

index_name	size_mb
sys_created_on	297
partial_idx	28

servicenow

Include indexes

Include index: make a distinction between columns kept in the entire index or only leaf nodes.

- Key columns are contained in the entire index
- Include columns are only contained in the leaf nodes
- Use case:
  - #1: include column is needed to provide an `Index Only Scan` for the query

  AND

  - #2: include column is not needed for filtering, sorting or joining

  - example:

```
create index idx_include
    ON task (sys_created_on)
    INCLUDE (sys_id);
```

Index `idx_include` will work for:

```
SELECT sys_id FROM task
WHERE sys_created_on > '2024-10-03';
```

Unique indexes can use `INCLUDE` columns to add columns without impacting the `UNIQUE` constraint.

Other index types

servicenow

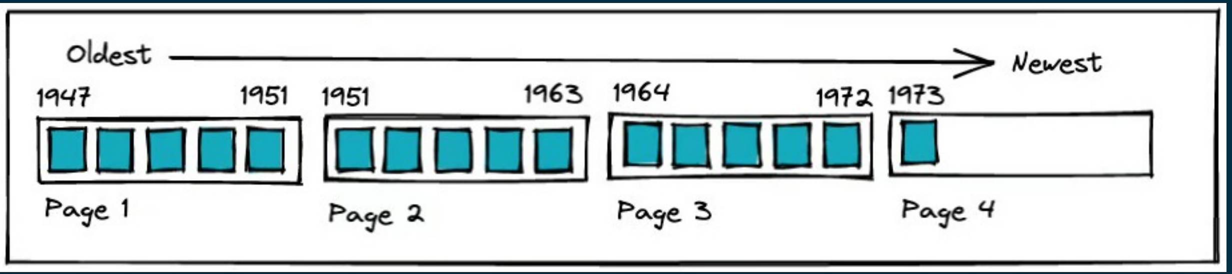
BRIN indexes

BRIN is a Block Range Index – designed for very large tables with high data correlation.

- BRIN works best if physical table layout and column ordering is strongly correlated
- Very low cost of INSERT operations
- Extremely small index sizes

Typical use cases:

- Logging tables
- IoT / sensors
- Time series data



- One entry for each range of pages (very small size)
- Number of pages is configurable, 128 is the default
- Can be 1000x smaller than B-Tree

servicenow

BRIN indexes – size & performance

BRIN vs B-Tree size:

Relation	Size
table_size	42 MB
btree_random_size	21 MB
brin_random_size	24 kB
btree_sequential_size	21 MB
brin_sequential_size	24 kB

Performance:

Rows	BTree Rand	BTree Seq	BRIN Rand	BRIN Seq
100	0.6 ms	0.5 ms	211 ms	11 ms
1000	5 ms	2 ms	207 ms	10 ms
10000	22 ms	13 ms	221 ms	15 ms
100000	98 ms	85 ms	250 ms	67 ms

servicenow

Hash indexes

Hash index: index used only for equality predicates (`WHERE x = value`):

- 32-bit hash code derived from the value of the indexed column
- Good use case – long / wide columns: URLs, UUIDs etc.
- Safe to use from PostgreSQL 10+ (not written to WAL in 9.6!)

Pros:

- Fast search performance
- Reduced disk I/O
- Potentially smaller size than B-Tree

Cons:

- Limited for range queries
- No ordering
- Hash Collisions
- Different hash functions

servicenow

GIN indexes

GIN index – Generalized Inverted Index: preferred approach for full-text indexing in PostgreSQL.

- GIN use cases:
  - Array columns
  - Text Search documents (`tsvector`)
  - Binary JSON documents (`jsonb`)

- Full text search is based on a match operator @@
- The operator returns true if a tsvector (`document`) matches a tsquery (`query`)
- Order does not matter

servicenow

Why my index is not working?

There are 3 common reasons why an index is not used:

- ➢ Wrong index ordering
  - `WHERE b > 3` and `c = 0` for a (`a, b, c`) composite index
- ➢ Function / expression
  - `WHERE upper(name) = 'Bob';`
  - `WHERE length(string) > 20;`
- ➢ Data type / collation mismatch
  - `WHERE id = '7'`

Remember: sometimes the index is not used because it's not worth it!

Always check cardinality / selectiveness.

servicenow

Summary

Summary / closing thoughts:

- Indexing in PostgreSQL is still a critical part of database performance
- B-Tree indexes will be 90%+ of use cases
- No need for 3rd party tools for building indexes – no exclusive locks
  - `CREATE INDEX … CONCURRENTLY;`
- Indexes can be created in PARALLEL
  - Automatic decision based on `max_parallel_maintenance_workers`
- Consider 3% - 10% as a threshold to make the index worth it
  - For any potential index on column A and table T, compare:
    - `SELECT count(distinct A) from T;`
    - `SELECT count(*) from T where A = <value>;`
    - `SELECT count(*) from T;`

Q & A

Thank you.

servicenow®