Indexing Beyond Columns in SQL: Expressions, Text Fragments, JSON Attributes, and Top-N Queries

Franck Pachot, Developer Advocate

yugabyteDB

@FranckPachot

What are you indexing?

yugabyteDB

When was CREATE INDEX introduced in the SQL Standard?

Year	Name	Alias	Comments
1986	SQL-86	SQL-87	First formalized by ANSI
1989	SQL-89	FIPS 127-1	Minor revision that added integrity constraints adopted as FIPS 127-1
1992	SQL-92	SQL2, FIPS 127-2	
1999	SQL:1999	SQL3	...nd ...pes), support
2003	SQL:2003		...ith
2006	SQL:2006		...oring XML ...l SQL data in XML form. In addition, it lets applications integrate queries into their SQL code with XQuery, the XML Query Language published by the World Wide Web Consortium (W3C), to concurrently access ordinary SQL-data and XML documents.[32]
2008	SQL:2008		Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers, TRUNCATE statement,[33] FETCH clause
2011	SQL:2011		Adds temporal data (PERIOD FOR)[34] (more information at Temporal database#History). Enhancements for window functions and FETCH clause.[35]
2016	SQL:2016		Adds row pattern matching, polymorphic table functions, operations on JSON data stored in character string fields
2019	SQL:2019-2020		Adds Part 15, multidimensional arrays (MDarray type and operators)
2023	SQL:2023		Adds data type JSON (SQL/Foundation); Adds Part 16, Property Graph Queries (SQL/PGQ)

This old idea of indexing columns

IMAGE

compression.

indicates that an index is to be maintained for the values in the column. Join operations can be performed only if both columns referenced in the joining predicate are defined as IMAGE.

Oracle Users Guide - Version 2.3
Copyright (c) April 1981
By Relational Software Incorporated
All rights reserved. Printed in U.S.A.

2-33

The CREATE TABLE Statement

```
CREATE TABLE table

  column(|CHAR(len) [VAR]| [NONULL] [UNIQUE] [UC] [IMAGE]), . . .
        |NUMBER          |
```

The CREATE TABLE statement defines a new table that is to be physically stored in the database. A table may contain from 1 to 255 columns. The CREATE TABLE specifies the name of the table, the names of the columns, and the column data types. The presence of null or duplicate values within a column may be restricted. High-performance access paths may be specified on any columns.

ORACLE automatically maintains an index (IMAGE) for the first column defined in the table. To optimize sequential processing the rows of the table are stored in physical sequence based on this index. This column is automatically treated as a required (NONULL) item.

You don't index columns, but predicate values to find rows from one table

An optimal index access uses one index per table

- better use one index with many columns than two index on one column

WHERE predicates can filter on a prefix, or case insensitive, a hash value

- index the searched value rather than the stored value

Think of indexes as redundant storage organized like your query result

Example: list all orders shipped within 1 day, in France, Top-10 by amount

```
create index (   trunc(ship_date-order_date)
               , initcap(country) , amount desc )
```

Indexes are redundant storage organized for your queries on a table

Indexes are maintained when table is updated

- Sorted on a key (the order of columns, binary values)
- May be partitioned (local index on partitioned table)
- May index a subset of rows (partial indexes)
- May include more columns (not sorted, just to filter or fetch them)

Indexes are used transparently in queries

- to find one point (row) or range (rows) or multiple ranges (loose index scan)
- to read the rows in a specific order (to avoid further sorting)
- to read a smaller structure (not all the table columns)

Partial indexes

yugabyteDB

You don't need to index all table rows

Example: table `orders ( id uuid, processed boolean )`

The many processed orders a kept for analytic queries

- no need to index 98% or rows with processed=true

The few unprocessed orders are selected to be processed

- useful to index 2% of rows with processed=false

```
create index orders_to_process
    on orders ( id ) where not processed
```

The cost of maintaining indexes is divided by 2, and it is cache efficient

When partial index is not supported

Oracle: no partial index

but null entries are not indexed

```
create index orders_to_process
  on orders ( (case when not processed then id end) )
```

You must use the same expression in SELECT

- or use a view / virtual columns

Or use partitioning and partial indexes

Covering indexes

Index Access to a Table is a Nested Loop Join

Forget about O(logN) time complexity of B-Tree indexes
What takes long is fetching the rows scattered in the table from the index

```
create table orders ( country, order_date, product );
create index order_country on orders (country);
select * from orders where country='France'
                  and order_date > now()-interval'1 day';
```

This will fetch all orders from France, scattered on disk (entered through years), to finally discard most of them

Avoid to read many rows from the table and discard them later

Indexes should cover the most selective predicates to avoid unnecessary hops to the table

```
create index order_by_country_and_date
                on orders (country, order_date);
select * from orders where country='France'
              and order_date > now()-interval'1 day';
```

This will filter before going to the table,
fetching only the rows needed for the result

What is a covering index?

There's no universal covering index

An index may be covering for a query

- covering the filtering (WHERE)
- covering the sorting (ORDER BY)
- covering the projection (SELECT)

It can cover more columns by

- adding them at the end of the index key
- adding them in INCLUDE if they don't need to be sorted on it

What about Index Only Scan?

🅾️ Oracle: you see fully covering index by the absence of
`TABLE ACCESS BY ROWID`

🐘 PostgreSQL: you see fully covering index by the presence of
`Index Only Scan`
and `Heap Fetches: 0` (needs fresh vacuum to update visibility map)

YugabyteDB: like PostgreSQL but no need for vacuum
(`Heap Fetches` is always `0`)

Order preserving indexes

yugabyteDB

Indexes are physically ordered

List the Top-10 orders by amount of the past year

You don't want to:

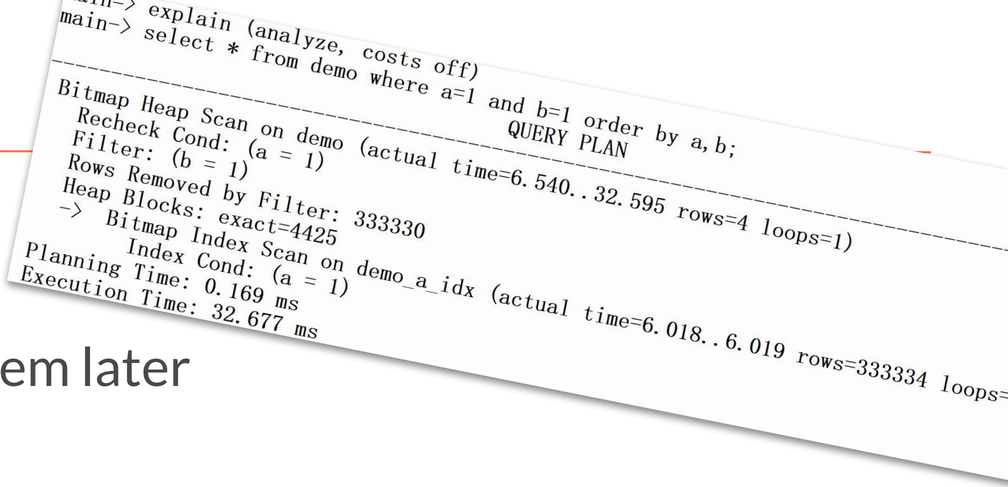
read 1 million order, sort them, display the first 10 ones

You want to:

read the orders in amount descending order, and stop at ten

```
create index … on ( order_year desc, amount desc )
```

When you look at an execution plan



Anti-pattern:

read many rows and discard them later

What you want: read only the rows that you will need in the result

Indexes should be used to:

- access directly to highly selective predicate result
- get rows in the order expected for the result
- avoid going to the table for many rows

Expressions-based indexes

You don't index the columns, but the expressions you filter on

If you apply a function to the column, it changes the order and index cannot be used.
Example: `uppercase(name)` cannot use an index on `name`

But you can index the result of the function or expression
Example: index the duration by indexing (`end_date-start_date`)

⚠️the function must be deterministic and immutable

Deterministic functions

The indexed expression must be always the same for the same row values

You cannot index `age()` as `now()-birth_date`

You cannot index `to_char(date)` as it depends on the locale context

You can create your deterministic/immutable function

but if you lied to the database, you will get corrupted results

Text-search indexes

LIKE 'prefix%'

Example: The first name starts with 'xyz'. Or name ~ 'Dupon[td]'

An index is sorted. You can find the prefix, and then filter without going to the table. `create index on … ( name )`

For large columns, some databases can use `on(substr(name,1,5) )`

LIKE '%suffix'

Example: The name ends with 'xyz'.

```
create index on … ( reverse(name) )
```

Query it with `select`

```
        where reverse(name) like reverse('%xyz')
```

PostgreSQL `plvstr.rvrs()` (from orafce) is faster than `reverse()`

LIKE '%middle%'

Example: There is 'xyz' in the middle.

🐘 PostgreSQL or YugabyteDB:

```
create extension if not exists pg_trgm;
create index on … using gin on ( name gin_trgm_ops )
```

It indexes trigrams (all 3-consecutive characters combinations)

It searches for those ' x', ' xy', 'xyz', 'yz ', ' z'

GIN indexes can have multiple index entries for one rows (fragments)

Vectors

With embeddings and vector search (pg_vector in PostgreSQL and YugabyteDB) you can use LLM (Large Language Model) to find possible matching text (vector distance)

Be careful: non-deterministic

JSON/JSONB indexing

Indexing a single path within JSON

For a single value per row, regular indexes can be used because the path is an expression

Example: `create index on … ( (data->document->>name) )`

For arrays you need one index per item:

```
create index on ( (data->versions->0->>name) )
create index on ( (data->versions->1->>name) )
create index on ( (data->versions->2->>name) )
```

Indexing paths through arrays within JSONB in PostgreSQL

An array can have many values, GIN index can index many values

Example:

```
create index on …
using gin ( data->versions jsonb_path_ops )
```

This indexes all values in a subdocument, to be queried with

- `@>` contains — examples: `@>'[1,3]'` or `@>'{"tag":"Devoxx"}'`
- `@?` json path item exists? — example: `@? '$.tags[*] ? (@ == "qui")'`
- `@@` json path return first item — example: `@? '$.tags[*] ? '$.tags[*] == "qui"'`

Indexing all keys and values within JSONB in PostgreSQL

The non-default `jsonb_path_ops`
indexes for `@>` , `@?` , `@@` , when knowing the key where we look for

The default `jsonb_ops`
Example: `create index on … using gin ( data jsonb_ops )`

This indexes all keys in a document, to be queried with `@>`, `@?`, `@@` plus:

- `?` exists one (key or array element) examples: `? 'tag'` or `? 'PostgreSQL'`
- `?|` exists any example: `tags ?| array['PostgreSQL','YugabyteDB']`
- `?&` exists all example: `tags ?& array['Distributed','PostgreSQL']`

Top-N and Pagination

Top-N queries and pagination without sorting all rows

Top-N: read the first rows only, from the beginning (or end) of the index
Pagination: read rows from the last value that was read

💡 look at the execution plan for the absence of `Sort`

The index starts with columns to filter
on equality, then range if same column as order by

```
create index … on ( country asc, name desc )

select … where country='FR' and name>'LastRetreived'
         order by name limit 10
```

Min/Max, distinct on, and Loose Index Scan

yugabyteDB

Min/Max optimisation

Easy to get the lowest or greatest value for an index:
  the first or last entry

```
create index…on ( score desc, name asc )
select min(score), max(score)
```

In the execution plan: `Index Scan` or `Index Scan Backward`
- B-Tree have 2-way links between leaves: same performance
- LSM-Tree may need more key comparisons for backward scan

First Row of Each Set of Grouped Rows Using GROUP BY

- Latest measure for each metric in a timeseries database
- Last contract with each supplier
- First purchase from each client
- Employee with the lowest salary in each department
- Lowest value for each sample taken at one time

```
create index…on ( metric, time desc ) include (value)
```

it is like a Min/max for each group

Without Loose Index Scan

```
create index … on ( metric, time desc ) include (value)
```

Those work in PostgreSQL but are not efficient (reads all index entries):

```
select distinct on (metric), value order by value asc;
```

```
select (
 row_number() over (partition by metric order by value asc)
) as r … where r=1 ;
```

Simulate Loose Index Scan (PostgreSQL)

```
create index … on ( metric, time desc ) include (value)

with recursive R as (
   select * from metrics
   order by metric, time limit 1
  union all
  select * from R , lateral (
   select * from metrics where metric > R.metric
   order by metric, time limit 1
  )
) select * from R
```

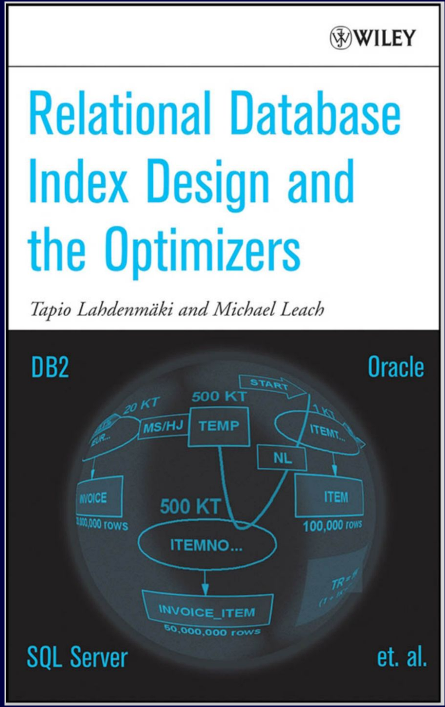
With Loose Index Scan for DISTINCT (Timescale, YugabyteDB)

```
create index … on ( metric, time desc ) include (value)
```

Efficient when DISTINCT can skip through one Index Scan:

```
with D as (
   select distinct metric from metrics
)
  select * from D , lateral (
   select * from metrics where metric = D.metric
   order by metric, time limit 1
```

Fat indexes or Too many indexes



yugabyteDB

Think of an index as a materialized query on one table

CREATE INDEX parts are very similar to a single-table SELECT statement

```
create index on t
    ( a, b, c asc, d ) include (e) where (f)
```

is like:

```
select d, e  from t
             where a=? and b=? and c<? and f
             order by c asc
```

Fat Indexes but not too many

You could create the ideal indexes for each query (one per table access)

But:

- they take space (on disk, in memory)
- they must be maintained on insert, delete, update

The ideal index serves multiple queries

- not defined per query but per access patterns (use cases)
- depends on your workload (fast ingest/update vs. query efficiency)

https://www.yugabyte.com/blog/author/fpachot/

PostgreSQL Tips & Tricks

How To

Improving Your SQL Indexing: How to Effectively Order Columns

JUNE 17, 2024
BY FRANCK PACHOT

Optimizing SQL indexing involves carefully considering the column order, understanding index scan ranges, and leveraging the index scan capabilities of your database to ensure efficient query performance. Discover how you can do this effectively by following the steps laid out in this new 'How to' guide.

Franck Pachot
Developer Advocate YugabyteDB,
AWS Hero, SQL Dev & DBA (OCM)

E-mail:
fpachot@yugabyte.com

Blogs:
dev.to/FranckPachot

Twitter:
@FranckPachot

LinkedIn:
www.linkedin.com/in/franckpacho

t

YugabyteDB
Community Slack / Github:
www.yugabyte.com/community

yugabyteDB

@FranckPachot