PostgreSQL Indexes demystified

Charly Batista, Postgres Tech Lead @ Percona

I am Charly Batista

I'm the Postgres Tech Lead at Percona. I'm passionate about database and algorithmic efficiency. I've been working with Database and Development for more than 20 years always trying to get the best performance of the databases, data structures and algorithms… Other than that, I'm a Brazilian who lives in China for more than 6 years but currently located in South America.

You can find me at **https://www.linkedin.com/in/charlybatista**
or more contacts info by the end of this presentation

Agenda

- An introduction to how Postgres stores data
- What is index?
- Index and Table Access
    - Seq Scan
    - Index Scan
    - Index Only Scan
    - Bitmap Scan

Agenda

- Join Operations
  - Nested Loops
  - Hash Join
  - Merge Join
  - Parallel joins
- Operators and data types

Agenda

- Index types
  - B-Tree
  - Generalized Inverted Index (GIN)
  - Block Range Index (BRIN)
  - Generalized Inverted Search Tree (GiST)

PERCONA

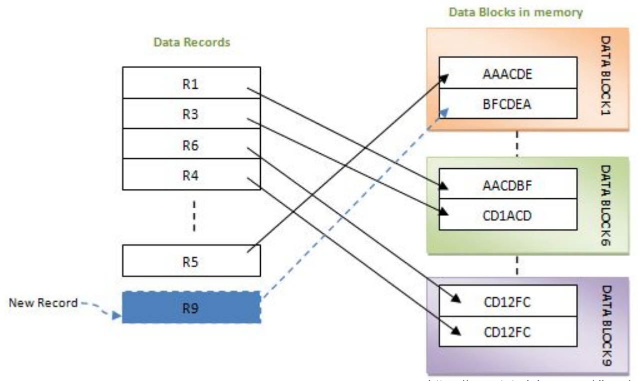
Agenda

- Working with indexes
    - Finding the right index type
    - Multiple columns and Partial indexes
    - Using functions and expressions in an index definition
    - Using INCLUDE to create a covering index for Index-Only Scans
    - Adding and dropping PostgreSQL indexes safely in production
- Conclusion

How Postgres stores data

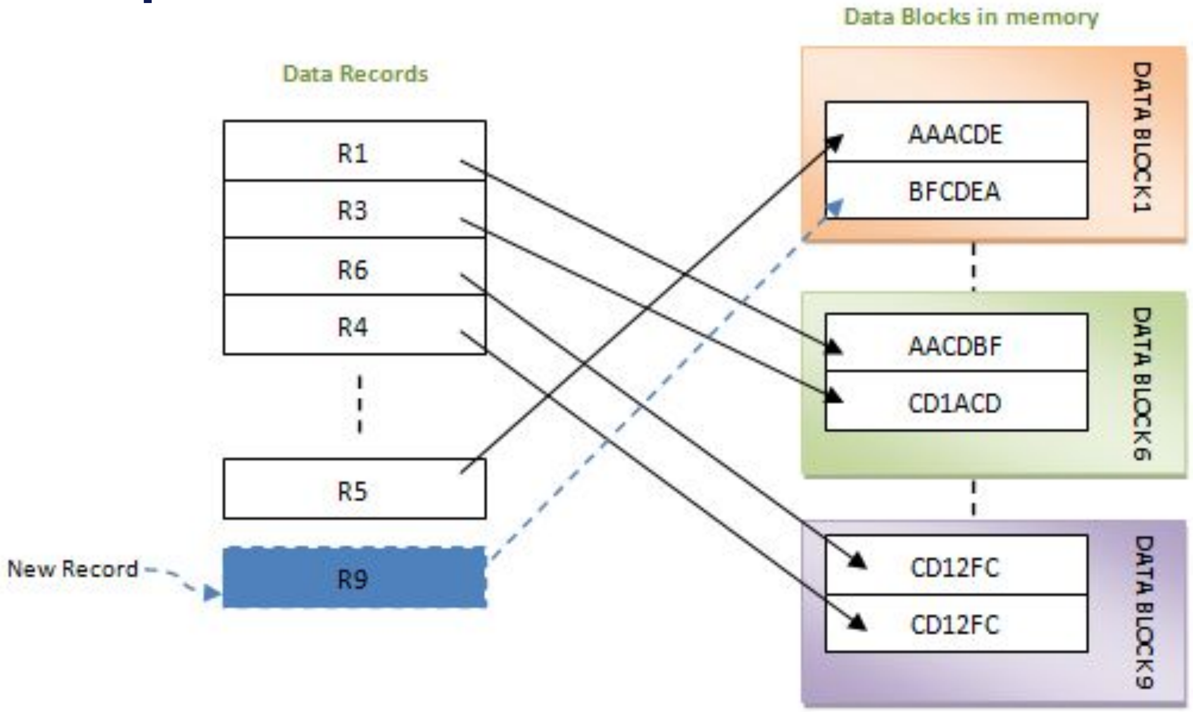
Heap File

- One of the simplest form of file organization
- Unordered set of records stored on pages
- Insert efficient
  - New records are inserted at the end of the file
- No sorting or ordering of the records can be expected
- Once the page is full, next record is stored in a new page
- The new page is logically the next closer page
- The new page can be physically located anywhere in the disk
- Deletion is accomplished by marking records as "deleted"
- Update is done by: “delete” the old record and insert the new one



https://www.tutorialcup.com/dbms/heap-file-organization.htm

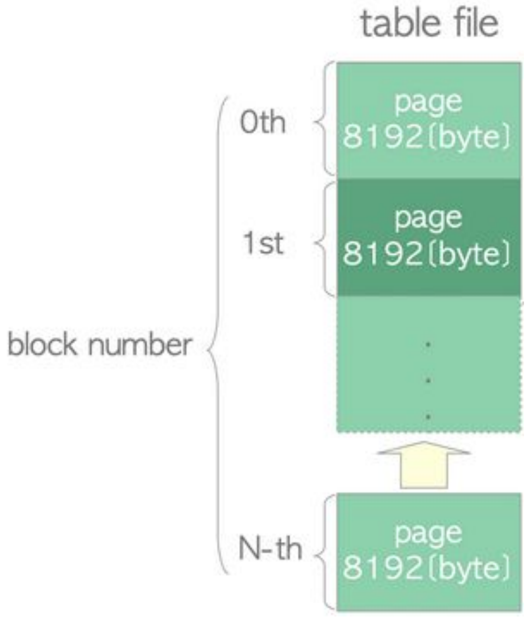
Heap File



https://www.tutorialcup.com/dbms/heap-file-organization.htm

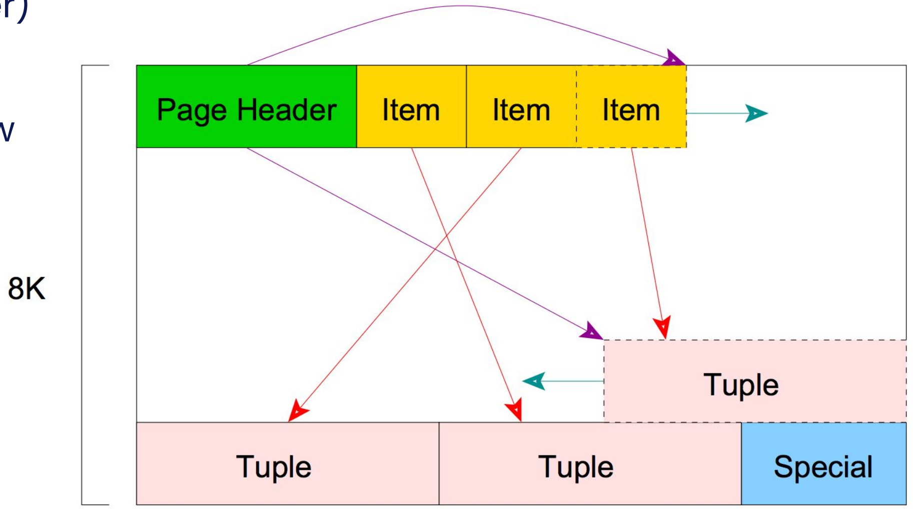
Heap File in Postgres

- The tables are heap files
- Each heap file has a limit of 1GB
- Meaning that:
  - Each table has a primary heap disk file
  - When growing more than 1GB other files are created
- It's divided into pages (or blocks) of fixed length
- The default page size is 8 KB
  - It can only be changed at compilation time
- In a table, all the pages are logically equivalent
- A row can be stored in any page



Page Layout

- A page is divided into:
  - **PageHeaderData**: The first 24 bytes of each page is a page header
  - **ItemIdData**: Array of item identifiers (line pointer) pointing to the actual items
  - **Free space**: The unallocated space used for new ItemIdData and new Items
  - **Items or heap tuple**: The actual items (rows) themselves
  - **Special space**: Holds index access method specific data. Empty in ordinary tables



https://stackoverflow.com/questions/59861645/postgres-and-tables-internal-organization

Table Row Layout

- All table rows are structured in the same way:
  - **Fixed-size header** (23 bytes on most machines), followed by an optional null bitmap
  - An optional **object ID field**
  - The **user data**
    - The user data begins at the offset indicated by **t_hoff** field in the header
  - The value of **t_hoff** must always be a multiple of the **MAXALIGN** distance for the platform
  - The field **t_infomask** in the header defines if the null bitmap is present
  - If the **null bitmap** is present it begins just after the fixed header
  - The null bitmap occupies enough bytes to have

    one bit per data column
  - When the bitmap is not present, all columns

    are assumed not-null

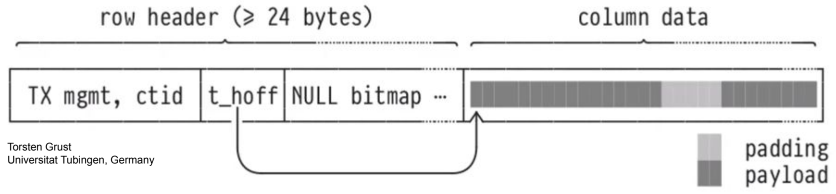
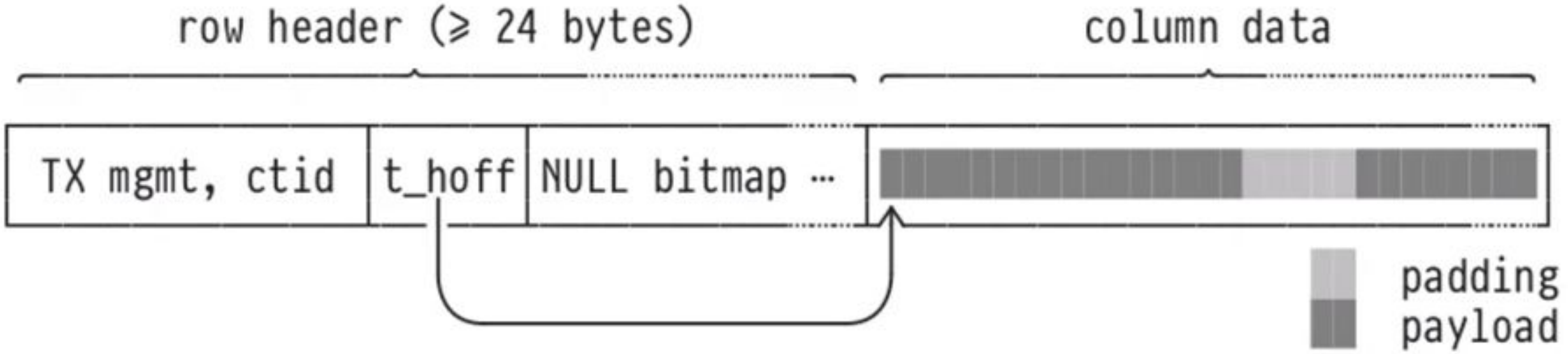


Table Row Layout



Torsten Grust
Universitat Tubingen, Germany

The Oversized-Attribute Storage Technique

- PostgreSQL uses a fixed page size (commonly 8 kB)
- PostgreSQL does not allow tuples to span multiple pages
- Large field values are stored outside of the heap table in separated files
- They are compressed and broken up into multiple physical rows outside
- The technique is affectionately known as TOAST
- Not all data types support TOAST
- Each table that is created has its own associated (unique) TOAST table

The Oversized-Attribute Storage Technique

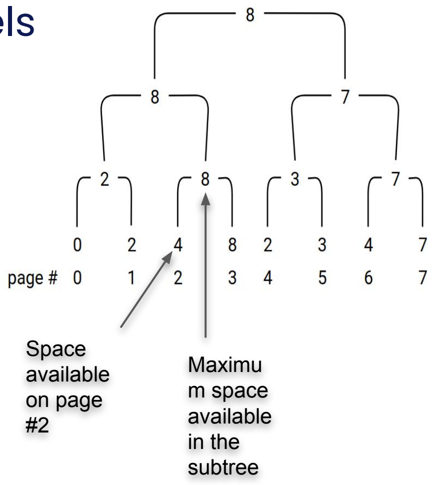
- How does it work?
  - When a row is "too large" (> 2KB by default), the TOAST mechanism attempts to compress any wide field values;
  - If that isn't enough to get the row under 2KB, it breaks up the wide field values into chunks that get stored in the associated TOAST table;
  - Each original field value is replaced by a small pointer that shows where to find this "out of line" data in the TOAST table;

Free Space Map

- Row delete or update may lead to holes in the heap file
- “New” records could fill those holes
- Postgres maintains a map to keep track of the available free space
- Used by INSERTs and UPDATEs
- Each heap and index relation has a Free Space Map (FSM)
- Hash index is an exception, it doesn’t have a FSM

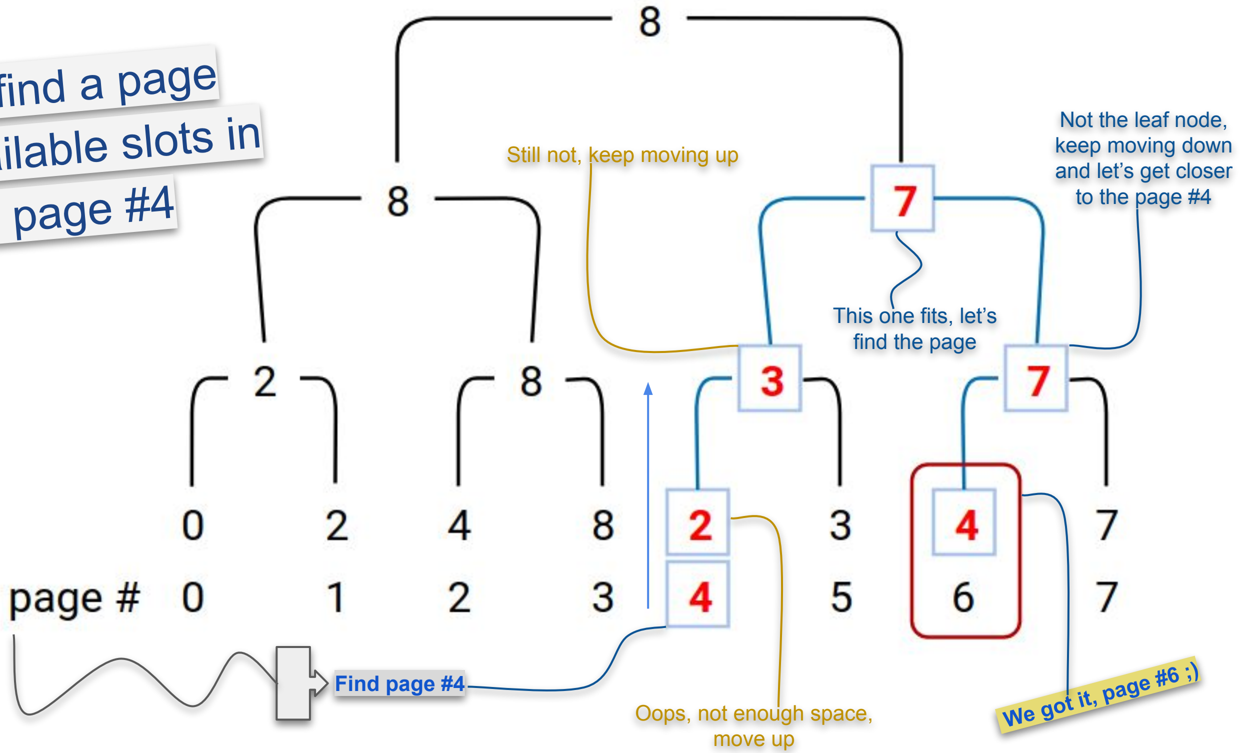
Free Space Map Tree

- The FSM file is organized as a tree of FSM pages
- The bottom pages store the free space available on each heap page
- The upper levels aggregate information from the lower levels
- Summary is:
    - One byte per node
    - Space measured in 32 byte units (1/256 of a 8k page)
    - Leaf node: space available in heap file page
    - Inner nodes: max space found in the subtree
    - Root node: max space found in this segment (file)


8
8
7
2
8
3
7
0 2 4 8 2 3 4 7
page # 0 1 2 3 4 5 6 7
Space available on page #2
Maximu m space available in the subtree

Free Space Map Tree

Say we want to find a page with at least 4 available slots in the vicinity of page #4


8
8
2
8
3
7
7
0
2
4
8
2
3
4
7
page #
0
1
2
3
4
5
6
7
Find page #4
Oops, not enough space, move up
Still not, keep moving up
This one fits, let's find the page
Not the leaf node, keep moving down and let's get closer to the page #4
We got it, page #6 ;)

PERCONA

Table access

What is index?

The documentation says:

“Indexes are a common way to ***enhance database performance****. An index allows the database server to* ***find and retrieve specific rows much faster*** *than it could do without an index. But indexes also* ***add overhead to the database*** *system as a whole, so they* ***should be used sensibly****.”*

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

Index and Table Access

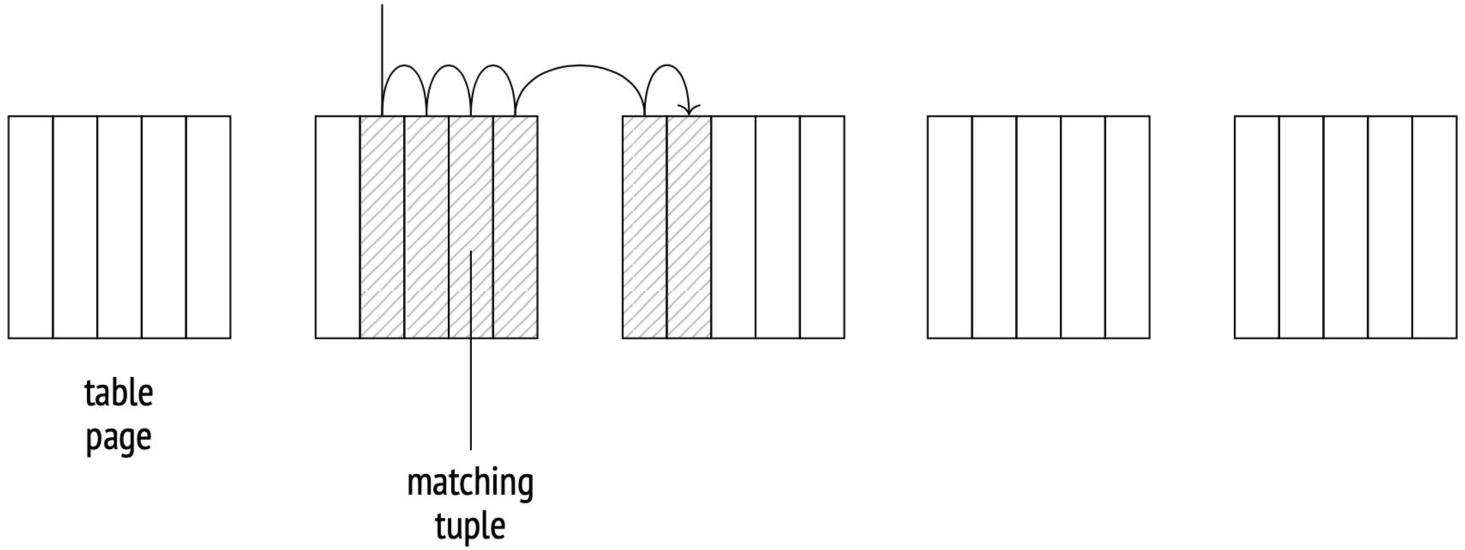
- **Seq Scan**
  - Scans the entire table in sequential order
  - Checks the visibility of each row version, discarding the ones that don't match the query
  - The scanning is done through the buffer cache
  - Uses a small buffer ring to prevent larger tables from abusing the buffer
  - If another process needs to scan the same table, it joins the buffer ring
  - Scan does not necessarily start from the beginning of the file
  - Sequential scan is the most cost-effective method of scanning a whole table
  - Sequential scan is efficient when having low selectivity

PERCONA

Index and Table Access

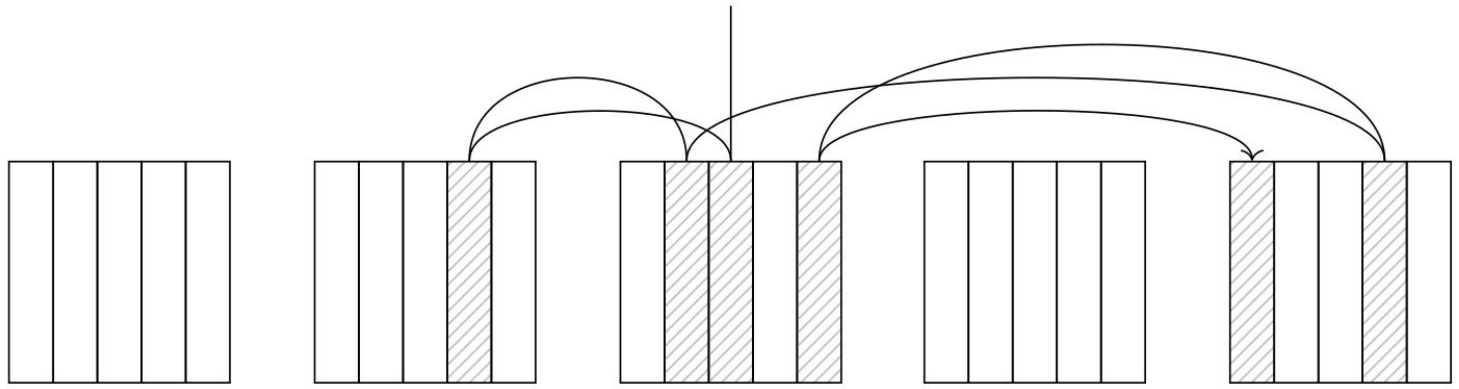
- **Index Scan**
  - Not all index methods support the INDEX SCAN access method
  - Fetches one tuple-pointer at a time from the index, and immediately visits that tuple in the table
  - Index page access patterns are random by nature
  - It can cause the read pattern to change from sequential to random
  - Index access is expensive, the cost is index access cost + table pages fetch cost
  - The cost of accessing a single page is estimated as random_page_cost
  - Index Scan is efficient when only few tuples from the table are scanned

High correlation



Source: PostgresPro
https://postgrespro.com/

Low Correlation



Source: PostgresPro
https://postgrespro.com/

Index and Table Access

- **Index Only Scan**
  - Also known as *covering index*
  - Happens when the index has all the required data to process a query
  - There is no need to access the table to retrieve data
  - The exception is to access visibility information, which it uses the visibility map
  - The cost is determined by the number of table pages flagged in the visibility map
  - Changes still present in the heap, not vacuumed yet increase the total plan cost

Index and Table Access

- **Bitmap Scan**
  - We’ll divide here into “Bitmap Heap Scan” and “Bitmap Index Scan”
  - Bitmap Index Scan fetches all the TIDs from the index in one go, sorts them using an in-memory bitmap data structure
  - Bitmap Heap Scan visits the table tuples in physical tuple-location order
  - The pages are fetched in ascending order, and each page is only fetched once
  - Help when the pattern is more random than sequential and the number of page fetches increases
  - If the bitmap gets too large it’s converted to "lossy" style, holding only the matching pages, not the matching tuples
  - The planner has to recheck its tuples against the query conditions again

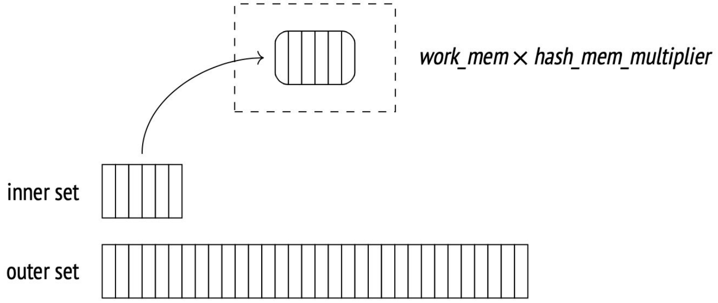
Join Operations

Join Operations

- Join is a key feature of SQL and they process two tables at a time
- In case a query has more joins, they are executed sequentially:
    - First two tables, then
    - The intermediate result with the next table
- In the context here, the term “table” can also mean “intermediate result”
- PostgreSQL offers several join methods:
    - Nested loop join
        - Use two main loops
        - First loop fetches all the rows from one table
        - It them uses the result to query the other table for each row from the first loop

Join Operations

- Hash join
  - builds a hash table based on the join key
  - Uses this new hash table to scan the outer table
  - Building the hash table is an added cost but removes the need for multiple loops
  - The planner usually decides to do a hash join when:
    - The tables are more or less the same size and
    - The hash table can fit in memory



Source: PostgresPro
https://postgrespro.com/

Join Operations

- Hash join
  - The planner can splits the inner set into multiple batches and processes them separately if it estimates that the hash table won’t fit into the allocated memory
  - This is what is called Two-pass hash join
  - During the first stage, the algorithm scans the inner set and builds a hash table separated into batches
  - During the second stage, it scans the outer set and matches against the hash batch
  - Both sets are sorted into batches and stored into temporary files
  - It proceeds with one batch a time
  - Wipes both temporary files and the hash table when finished, process the next batch
  - With N total batches, the number of temporary files will be $2^{(N-1)}$

Source: PostgresPro
https://postgrespro.com/

Join Operations

- Merge join
  - Uses two lists sorted by the so called “merge key” and returns an ordered output
  - The input data sets may be pre-sorted by an index scan or be sorted explicitly
  - It’s a lot faster than the other methods when using an index
  - The planner can be influenced by an “**order by**” clause
  - The output can be used as an input for another merge join as long as the order is kept intact
- Parallel joins
  - Parallel joins make use of multiple cores to speed up execution
  - It can be disabled setting the parameter max_parallel_workers_per_gather = 0
  - Not always makes queries faster

Indexes

Operators and data types

- Operators are used to compare one or two values of a given type
- Operators are essential for creating the right index
- An operator indicates whether a particular index can be used or not
- An operator can be thought of as the "how" one wants to search the table for values
- It can be as simple as “=” operator to match values for equality against an input value
- Or complex as “@@” to perform a text search on a tsvector column
- We can query the internal tables “pg_am”, “pg_opfamily”, and “pg_amop” to inspect the operators

Operators and data types

- Operator classes can be specified during CREATE INDEX:
  - *CREATE INDEX ON users (login text_pattern_ops);*
- PostgreSQL has a rich set of native data types available to users
- Users can add new types to PostgreSQL using the **CREATE TYPE** command
- Operators work on specific data types
- Using the wrong data type may result in an error
- Using the wrong data type may drive the planner to wrong choices

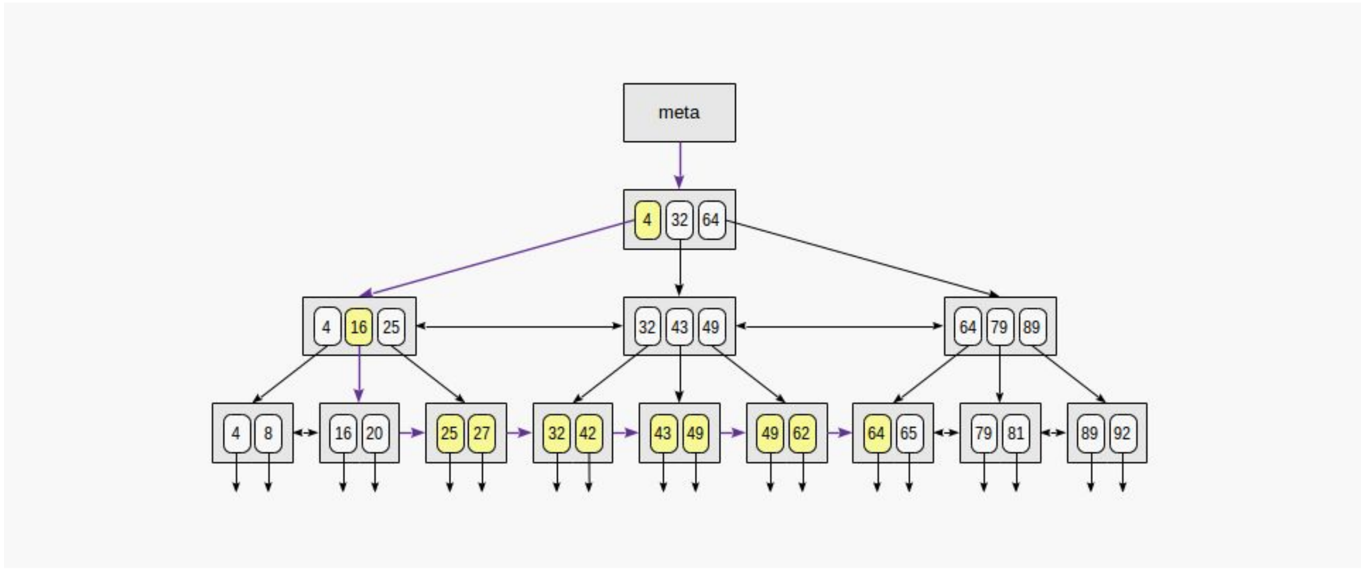
PERCONA

Index types

- Postgres implements few different index types
- B-Tree:
  - Good old B-tree
  - Default index type
  - Tuples are stored on pages, ordered by key
  - The very first page of the index is a metapage referencing the index root
  - B-trees are balanced
  - B-trees are multi-branched (each 8 KB page contains many TIDs)
  - Data in the index is sorted in nondecreasing order
  - Same-level pages are connected to one another by a bidirectional list

Index types

- B-Tree



Source: PostgresPro
https://postgrespro.com/

Index types

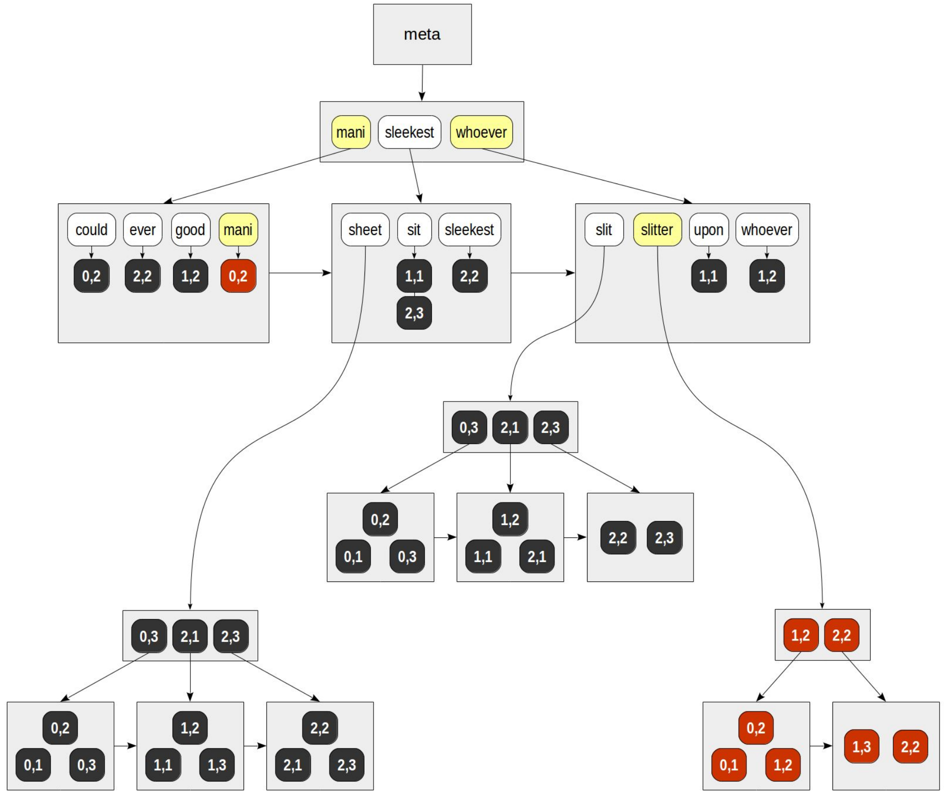
- What can we do wit B-Tree:
  - Find key = X
  - Find keys < X or > X
  - ORDER BY
  - LIKE 'foo%'

Index types

- GIN
  - Generalized Inverted Index
  - Internal structure is basically just a B-tree
    - Optimized for storing a lot of duplicate keys
    - Duplicates are ordered by heap TID
  - Bitmap scans only
  - Good fit for JSONB data, Array, Range Types, Full text search and HStore
  - Individual element gets indexed instead of entire value for indexed column
  - Expensive updates, uses a strategy, “fastupdate”, to improve performance
  - GIN pending list can be a problem

Index types

- GIN

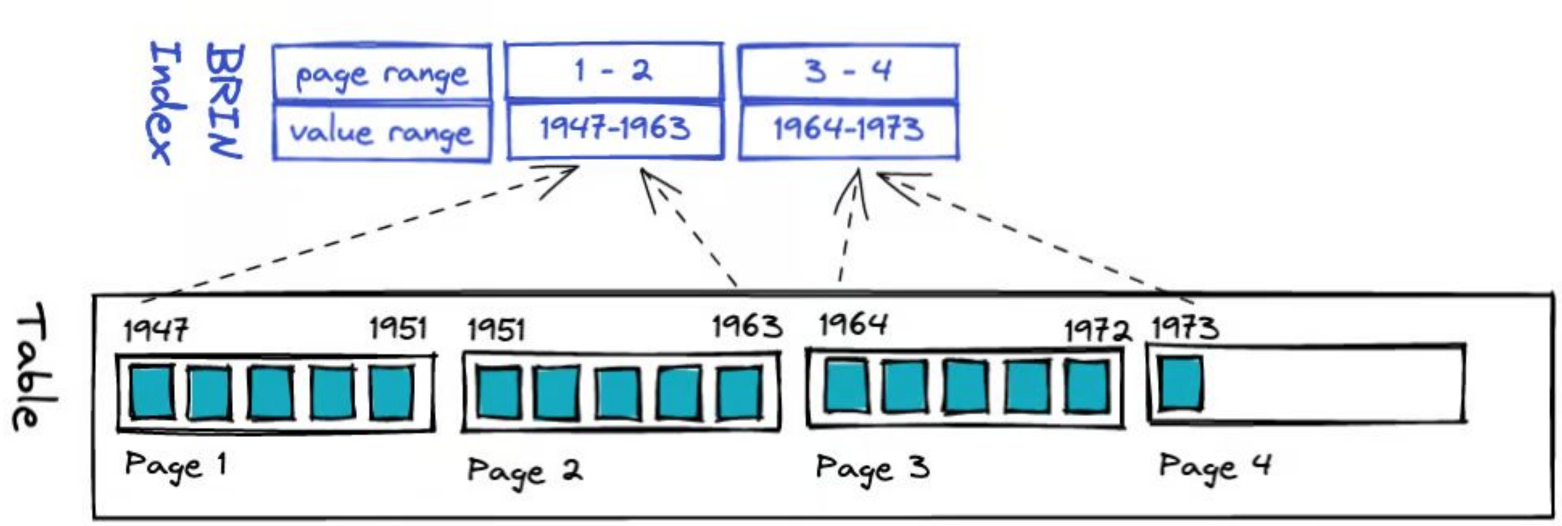


Index types

- BRIN
  - Block Range Index
  - Stores the page’s minimum and maximum values of indexed column
  - Good fit for time-series data or data which has liner sort order
  - Doesn’t perform well with dataset that has frequent updates
  - Performs better than B-Tree for liner data use cases
  - Very small in size compare to B-Tree
  - Lossy in nature and as workaround it performs little revalidation work and discard any tuples which don’t satisfy the condition

Index types

- BRIN



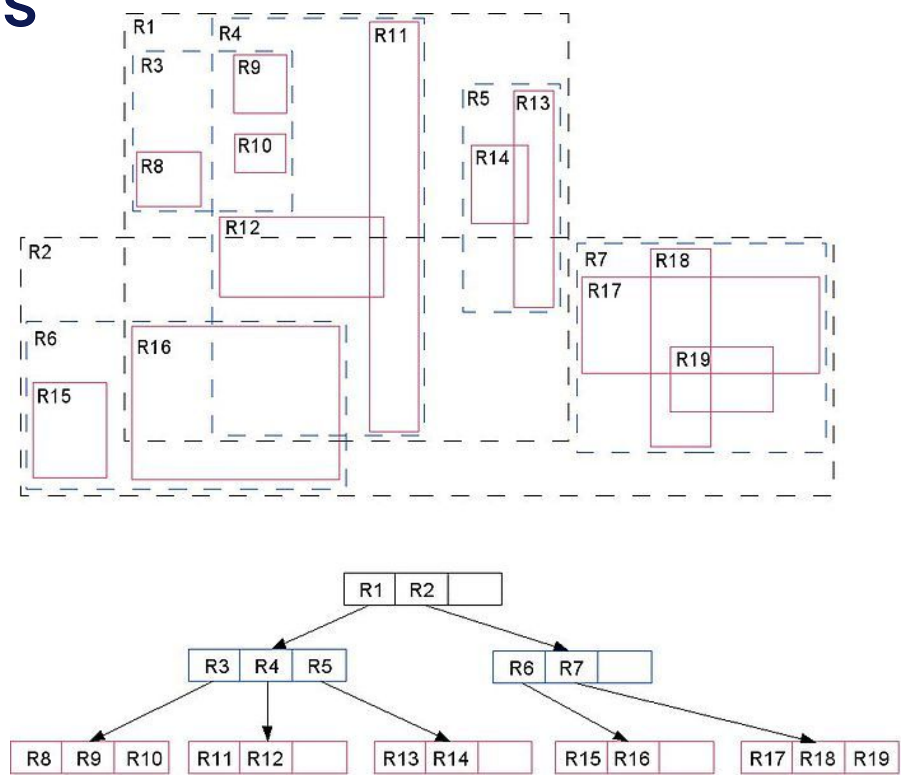
Source: crunchydata
https://www.crunchydata.com/blog/postgres-indexing-when-does-brin-win

Index types

- GiST
  - Generalized Inverted Search Tree
  - GiST is a balanced tree-like B-tree
  - Good fit for Full-text search or Geometric types
  - Highly used by PostGIS
  - For dynamic data, GiST is a better option compare to GIN
  - Wide Range of Query Types including equality queries, range queries, and partial match queries
  - Large Index Size
  - Slower Updates

Index types

- GiST



Mastering PostgreSQL 9.6 by Hans-Jurgen Schonig
https://www.oreilly.com/library/view/mastering-postgresql-96/9781783555352/bd3a3a9f-ebd3-422a-8e15-c8dd081db064.xhtml

Working with indexes

- Finding the right index type is sometimes tricky but:
  - Think of an index just like a specific data structure that supports a specific, limited set of search operators
  - Operator class is key
  - Data type is also key
  - Cardinality matters
  - Data distribution can change everything
  - Index size may become a problem

Working with indexes

- Multiple columns and Partial indexes
  - Most of the index types has the option to add multiple columns to an index definition:

    CREATE INDEX ON [table] ([column_a], [column_b]);
  - The end result depends on the index type
  - Each index type has a different representation for multiple columns in its data structure
  - BRIN or Hash do not support it, for example
  - **B-tree** multi-column indexes work well but **column order matters**
  - It's sometimes better to have 2 separate indexes than a multi-column

Working with indexes

- We can use functions and expressions in an index definition:

  CREATE INDEX ON users (lower(name));

  - It is useful to obtain fast access to tables based on the results of computations
  - Now queries like *SELECT * FROM users WHERE lower(name) = 'charly'* can use the index
  - It can be combined with UNIQUE constraint to prevent the insert of rows which values differ only in case
  - We can also use expressions, for example:
    - *CREATE INDEX people_names ON people ((first_name || ' ' || last_name));*
  - It would work for queries like this:
    - *SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';*

PERCONA

Working with indexes

- One be tempted to create multicolumn index to benefit from Index-Only Scan
- It is expensive and may not work with UNIQUE indexes
- B-tree indexes has the INCLUDE keyword which is a better approach:
  - *CREATE INDEX ON users (login, id) INCLUDE (fullname);*
- This keeps the overhead for the additional columns slightly lower
- Doesn’t interfere with UNIQUE constraint indexes
- Communicates the intent: only added a column to support Index-Only Scans
- Still, adding more data to the index means larger index values, which on its own can be a problem

Working with indexes

- Adding and dropping PostgreSQL indexes safely in production may be tricky
- It can have impact on the I/O and also locks while the new index being built
    - Postgres will take an exclusive lock when one simply runs *CREATE INDEX*
- Postgres has the special **CONCURRENTLY** keyword to help
    - *CREATE INDEX CONCURRENTLY ON users (name) WHERE deleted_at IS NULL;*
- It works with CREATE and DROP
- There are some caveats, for example:
    - CREATE INDEX command can fail leaving behind an “invalid” index
    - This index will be ignored for querying purposes because it might be incomplete
    - However it will still have update overhead
- Always make sure the operation finished successfully

Conclusion

Conclusion

- It’s important to understand how Postgres organizes its data
- Sequential scan isn’t always bad
- Sequential scan is the best method to traverse a table
- Sequential scan can be done in parallel
- Indexes are tools built to help improve performance but,
    - Indexes are expensive to traverse (random I/O)
    - There are many ways to go wrong
    - Understanding the operator classes and data types are keys
    - Not all indexes are the same
    - Different indexes work better with different data, cardinality and distribution
    - Indexes are expensive to maintain and update data
    - Indexes should be used with caution

PERCONA LIVE 2023

Thank you!

Charly Batista

https://www.linkedin.com/in/charlybatista
https://github.com/elchinoo/

PERCONA

Want to know more about

Percona Software For Postgres?

Text goes here

FLOWCODE

PRIVACY.FLOWCODE.COM