EPAM Systems, RD Dep., RD Dep.

# POSTGRESQL DB FOR DWH AND ETL BUILDING

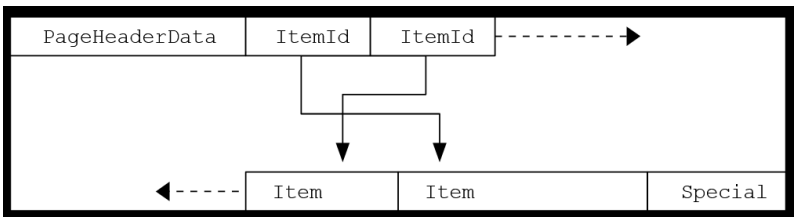
## PostgreSQL Relational Structures

# CONTENTS

# OVERVIEW OF OBJECT STORAGE

Each table or index whose size is less than 1GB is a single file stored under the database directory it belongs to. Tables and indexes as database objects are internally managed by individual OIDs. When the file size of tables and indexes exceeds 1GB, PostgreSQL creates a new file and uses it. If the new file has been filled up, next new file will be created, and so on.

Inside the data file, it is divided into pages (or blocks) of fixed length, the default is 8192 byte (8 KB). Those pages within each file are numbered sequentially from 0, and such numbers are called as block numbers. If the file has been filled up, PostgreSQL adds a new empty page to the end of the file to increase the file size.




Page Layout

Internal layout of pages depends on the data file types. An empty space between the end of line pointers and the beginning of the newest tuple is referred to as free space or hole.

To identify a tuple within the table, tuple identifier (TID) is internally used. A TID comprises a pair of values: the block number of the page that contains the tuple, and the offset number of the line pointer that points to the tuple.

In addition, heap tuple whose size is greater than about 2 KB (about 1/4 of 8 KB) is stored and managed using a method called **TOAST** (The Oversized-Attribute Storage Technique).

Looking carefully at the database subdirectories, you will find out that each table has two associated files suffixed respectively with '_fsm' and '_vm'. Those are referred to as **free space map** and **visibility map**, storing the information of the free space capacity and the visibility on each page within the table file, respectively. Indexes only have individual free space maps and don't have visibility map.

## 1.1 FREE SPACE MAP

Each heap and index relation, except for hash indexes, has a Free Space Map (FSM) to keep track of available space in the relation. It's stored alongside the main relation data in a separate relation fork, named after the filenode number of the relation, plus a _fsm suffix. For example, if the filenode of a relation is 12345, the FSM is stored in a file called 12345_fsm, in the same directory as the main relation file.

The Free Space Map is organized as a tree of FSM pages. The bottom level FSM pages store the free space available on each heap (or index) page, using one byte to represent each such page. The upper levels aggregate information from the lower levels.

Within each FSM page is a binary tree, stored in an array with one byte per node. Each leaf node represents a heap page, or a lower level FSM page. In each non-leaf node, the higher of its children's values is stored. The maximum value in the leaf nodes is therefore stored at the root.

## 1.2 VISIBILITY MAP

Each heap relation has a Visibility Map (VM) to keep track of which pages contain only tuples that are known to be visible to all active transactions; it also keeps track of which pages contain only frozen tuples. It's stored alongside the main relation data in a separate relation fork, named after the filenode number of the relation, plus a _vm suffix. For example, if the filenode of a relation is 12345, the VM is stored in a file called 12345_vm, in the same directory as the main relation file. Note that indexes do not have VMs.

The visibility map stores two bits per heap page. The first bit, if set, indicates that the page is all-visible, or in other words that the page does not contain any tuples that need to be vacuumed. This information can also be used by index-only scans to answer queries using only the index tuple. The second bit, if set, means that all tuples on the page have been frozen. That means that even an anti-wraparound vacuum need not revisit the page.

The map is conservative in the sense that we make sure that whenever a bit is set, we know the condition is true, but if a bit is not set, it might or might not be true. Visibility map bits are only set by vacuum, but are cleared by any data-modifying operations on a page.

# TABLES AND INDEXES

Tables constitute the building blocks of relational database storage. Structuring tables so that they form meaningful relationships is the key to relational database design. In PostgreSQL, constraints enforce relationships between tables. To distinguish a table from just a heap of data, we establish indexes. Much like the indexes you find at the end of books or the tenant list at the entrances to grand office buildings, indexes point to locations in the table so you don’t have to scour the table from top to bottom every time you’re looking for something.

In addition to ordinary data tables, PostgreSQL offers several kinds of tables that are rather uncommon: temporary, unlogged, inherited, typed, and foreign.

## 2.1 TABLES

To create a table, you use the aptly named CREATE TABLE command. In this command you specify at least a name for the new table, the names of the columns and the data type of each column.

CREATE TABLE will create a new, initially empty table in the current database. The table will be owned by the user issuing the command.

If a schema name is given (for example, CREATE TABLE myschema.mytable ...) then the table is created in the specified schema. Otherwise, it is created in the current schema. Temporary tables exist in a special schema, so a schema name cannot be given when creating a temporary table. The name of the table must be distinct from the name of any other table, sequence, index, view, or foreign table in the same schema.

CREATE TABLE also automatically creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

The optional constraint clauses specify constraints (tests) that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience for use when the constraint only affects one column.

To be able to create a table, you must have USAGE privilege on all column types or the type in the OF clause, respectively.

Every table has several system columns that are implicitly defined by the system. Therefore, these names cannot be used as names of user-defined columns. (Note that these restrictions are separate from whether the name is a key word or not; quoting a name will not allow you to escape these restrictions.) You do not really need to be concerned about these columns; just know they exist.

**tableoid** - The OID of the table containing this row. This column is particularly handy for queries that select from inheritance hierarchies (see Section 5.10), since without it, it's difficult to tell which individual table a row came from. The tableoid can be joined against the oid column of pg_class to obtain the table name.

**xmin** - The identity (transaction ID) of the inserting transaction for this row version. (A row version is an individual state of a row; each update of a row creates a new row version for the same logical row.)

**cmin** - The command identifier (starting at zero) within the inserting transaction.

**xmax** - The identity (transaction ID) of the deleting transaction, or zero for an undeleted row version. It is possible for this column to be nonzero in a visible row version. That usually indicates that the deleting transaction hasn't committed yet, or that an attempted deletion was rolled back.

**cmax** - The command identifier within the deleting transaction, or zero.

**ctid** - The physical location of the row version within its table. Note that although the ctid can be used to locate the row version very quickly, a row's ctid will change if it is updated or moved by VACUUM FULL. Therefore ctid is useless as a long-term row identifier. A primary key should be used to identify logical rows.

### 2.1.1 Inherited Table

PostgreSQL stands alone as the only database product offering inherited tables. When you specify that a table (the child table) inherits from another table (the parent table), PostgreSQL creates the child table with its own columns plus all the columns of the parent table.

PostgreSQL will remember this parent-child relationship so that any subsequent structural changes to the parent automatically propagate to its children. Parent-child table design is perfect for partitioning your data.

When you query the parent table, PostgreSQL automatically includes all rows in the child tables. Not every trait of the parent passes down to the child.

Notably, primary key constraints, foreign key constraints, uniqueness constraints, and indexes are never inherited. Check constraints are inherited, but children can have their own check constraints in addition to the ones they inherit from their parents.

### 2.1.2 Partitioned Table

Partitioned tables are much like inherited tables in that they allow partitioning of data across many tables and the planner can conditionally skip tables that don’t satisfy a query condition. Internally they are implemented much the same, but use a different DDL syntax.

Although partitioned tables replace the functionality of inherited tables in many cases, they are not complete replacements. Here are some key differences between inherited tables and partition tables:

- A partitioned table group is created using the declarative partition syntax CREATE TABLE ..PARTITION BY RANGE ...
- When partitions are used, data can be inserted into the core table and is rerouted automatically to the matching partition. This is not the case with inherited tables, where you either need to insert data into the child table, or have a trigger that reroutes data to the child tables.
- All tables in a partition must have the same exact columns. This is unlike inherited tables, where child tables are allowed to have additional columns that are not in the parent tables.

- Each partitioned table belongs to a single partitioned group. Internally that means it can have only one parent table. Inherited tables, on other hand, can inherit columns from multiple tables.
- The parent of the partition can't have primary keys, unique keys, or indexes, although the child partitions can. This is different from the inheritance tables, where the parent and each child can have a primary key that needs only to be unique within the table, not necessarily across all the inherited children.
- Unlike inherited tables, the parent partitioned table can't have any rows of its own. All inserts are redirected to a matching child partition and when no matching child partition is available, an error is thrown.

### 2.1.3 Unlogged Table

For ephemeral data that could be rebuilt in the event of a disk failure or doesn't need to be restored after a crash, you might prefer having more speed than redundancy. The UNLOGGED modifier allows you to create unlogged tables. These tables will not be part of any write-ahead logs. The big advantage of an unlogged table is that writing data to it is much faster than to a logged table—10–15 times faster in our experience.

If you accidentally unplug the power cord on the server and then turn the power back on, the rollback process will wipe clean all data in unlogged tables. Another consequence of making a table unlogged is that its data won't be able to participate in PostgreSQL replication.

You can use unlogged tables in following scenarios:

- Large data sets that take a lot of time to import and are only used a couple of times (finance, scientific computing, and even big data).
- Dynamic data that after a server crashes will not be that useful anyway, such as user sessions.
- Static data that you can afford losing and re-importing in the unlikely event of a server crash.

Finally, unlike temporary tables, unlogged ones are not dropped at the end of a the session or the current transaction, and under normal operations (that is, no crashes) they are, in fact, persistent and operate normally — but faster.

### 2.1.4 Clustered Table

In PostgreSQL, there is a command called CLUSTER that allows us to rewrite a table in the desired order. It is possible to point to an index and store data in the same order as the index.

The CLUSTER command has been around for many years and serves its purpose well. However, there are some things to consider before blindly running it on a production system:

- The CLUSTER command will lock the table while it is running. You cannot insert or modify data while CLUSTER is running. This might not be acceptable on a production system.
- Data can only be organized according to one index. You cannot order a table by zip code, name, ID, birthday, and so on, at the same time. This means that CLUSTER will make sense if there are search criteria that are used most of the time.
- Keep in mind that the example outlined in this book is more of a worst-case scenario. In reality, the performance difference between a clustered and a nonclustered table will depend on the workload, the amount of data retrieved, cache hit rates, and a lot more.
- The clustered state of a table will not be maintained as changes are made to a table during normal operations. Correlation will usually deteriorate as time goes by.

In cases where you are accessing single rows randomly within a table, the actual order of the data in the table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using CLUSTER. If you are requesting a range of indexed values from a table, or a single indexed value that has multiple rows that match, CLUSTER will help because once the index identifies the table page for the first row that matches, all other rows that match are probably already on the same table page, and so you save disk accesses and speed up the query.

## 2.2 INDEXES

PostgreSQL comes with a lavish framework for creating and fine-tuning indexes. The art of PostgreSQL indexing could fill a tome all by itself. PostgreSQL is packaged with several types of indexes. If you find these inadequate, you can define new index operators and modifiers to supplement. If still unsatisfied, you're free to invent your own index type.

PostgreSQL also allows you to mix and match different index types in the same table with the expectation that the planner will consider them all. For instance, one column could use a BTree index while an adjacent column uses a GiST index, with both indexes contributing to speed up the queries.

Indexing a table is as much a programming task as it is an experimental endeavor. A misappropriated index is worse than useless. Not all indexes are created equal. Algorithmists have devised different kinds of indexes for different data types and different query types, all in an attempt to scrape that last morsel of speed from a query.

You can create indexes on tables (with the exception of foreign tables) as well as materialized views.

### 2.2.1 B-tree

The standard index type is the B-tree, where the B stands for balanced. A balanced tree is one where the amount of data on the left and right side of each split is kept even, so that the amount of levels you have to descend to reach any individual row is approximately equal.

The B-tree can be used to find a single value or to scan a range, searching for key values that are greater than, less than, and/or equal to some value. They also work fine on both numeric and text data. Recent versions of PostgreSQL can also use an index to find (or avoid) null values in a table.

B-trees have a few important traits:

- B-trees are balanced, that is, each leaf page is separated from the root by the same number of internal pages. Therefore, search for any value takes the same time.
- B-trees are multi-branched, that is, each page (usually 8 KB) contains a lot of (hundreds) TIDs. As a result, the depth of B-trees is pretty small, actually up to 4–5 for very large tables.
- Data in the index is sorted in nondecreasing order (both between pages and inside each page), and same-level pages are connected to one another by a bidirectional list. Therefore, we can get an ordered data set just by a list walk one or the other direction without returning to the root each time.

The very first page of the index is a metapage, which references the index root. Internal nodes are located below the root, and leaf pages are in the bottommost row. Down arrows represent references from leaf nodes to table rows (TIDs).

### 2.2.2 Hash

PostgreSQL includes an implementation of persistent on-disk hash indexes, which are fully crash recoverable. Any data type can be indexed by a hash index, including data types that do not have a well-defined linear ordering. Hash indexes store only the hash value of the data being indexed, thus there are no restrictions on the size of the data column being indexed.

The hash index type can be useful in cases where you are only doing equality (not range) searching on an index, and you don't allow null values in it. Hash indexes support only single-column indexes and do not allow uniqueness checking.

Each hash index tuple stores just the 4-byte hash value, not the actual column value. As a result, hash indexes may be much smaller than B-trees when indexing longer data items such as UUIDs, URLs, etc. The absence of the column value also makes all hash index scans lossy. Hash indexes may take part in bitmap index scans and backward scans.

Hash indexes are best optimized for SELECT and UPDATE-heavy workloads that use equality scans on larger tables. In a B-tree index, searches must descend through the tree until the leaf page is found. In tables with millions of rows, this descent can increase access time to data. The equivalent of a leaf page in a hash index is referred to as a bucket page.

The search by Hash indexes is faster than the search performed by B-tree indexes even if hash indexes do not support the index-only scan method.

### 2.2.3 BRIN

Block range index (BRIN) is an index type introduced in PostgreSQL 9.4. It’s designed specifically for very large tables where using an index such as B-Tree would take up too much space and not fit in memory. The approach of BRIN is to treat a range of pages as one unit. BRIN indexes are much smaller than B-Tree and other indexes and faster to build. But they are slower to use and can’t be used for primary keys or certain other situations.

The first (more exactly, zero) page contains the metadata.

Pages with the summary information are located at a certain offset from the metadata. Each index row on those pages contains summary information on one range.

Between the meta page and summary data, pages with the reverse range map are located. Actually, this is an array of pointers (TIDs) to the corresponding index rows. For some ranges, the pointer in range map can lead to no index row (one is marked in gray in the figure). In such a case, the range is considered to have no summary information yet.

The algorithm is simple. The table is split into ranges that are several pages large (or several blocks large, which is the same) - hence the name: Block Range Index, BRIN. The index stores summary information on the data in each range. As a rule, this is the minimal and maximal values, but it happens to be different, as shown further. Assume that a query is performed that contains the condition for a column; if the sought values do not get into the interval, the whole range can be skipped; but if they do get, all rows in all blocks will have to be looked through to choose the matching ones among them.

The map of ranges is sequentially scanned (that is, the ranges are went through in the order of their location in the table). The pointers are used to determine index rows with summary information on each range. If a range does not contain the value sought, it is skipped, and if it can contain the value (or summary information is unavailable), all pages of the range are added to the bitmap. The resulting bitmap is then used as usual.

### 2.2.4 GIN

Regular indexes are optimized for the case where a row has a single key value associated with it, so that the mapping between rows and keys is generally simple. The Generalized Inverted Index (GIN) is useful for a different sort of organization. GIN stores a list of keys with what's called a posting list of rows, each of which contain that key. A row can appear on the posting list of multiple keys too.

With the right design, GIN allows efficient construction and search of some advanced key/value data structures that wouldn't normally map well to a database structure. It leverages the ability to customize the comparison operator class used for an index, while relying on a regular B-tree structure to actually store the underlying index data.

GIN is useful for indexing array values, which allows operations such as searching for all rows where the array column contains a particular value, or has some elements in common with an array you're matching against. It's also one of the ways to implement full-text search.

Anytime you're faced with storing data that doesn't match the usual single key to single value sort of structure regular database indexes expect, GIN might be considered as an alternative approach. It's been applied to a wide range of interesting indexing problems.

### 2.2.5 GiST

A Generalized Search Tree (GiST) provide a way to build a balanced tree structure for storing data, such as the built-in B-tree, just by defining how keys are to be treated. This allows using the index to quickly answer questions that go beyond the usual equality and range comparisons handled by a B-tree.

For example, the geometric data types that are included in PostgreSQL include operators to allow an index to sort by the distance between items and whether they intersect.

GiST can also be used for full-text search. When you need a really customized type of index, with operators that go beyond the normal comparison ones, but the structure is still like a tree, consider using GiST when a standard database table doesn't perform well.

### 2.2.6 SP-GiST

Space-Partitioned Generalized Search Tree (SP-GiST) can be used in the same situations as GiST but can be faster for certain kinds of data distribution. PostgreSQL's native geometric data types, such as point and box, and the text data type, were the first to support SP-GiST.

# VIEWS

## 3.1 BASIC VIEW

PostgreSQL provides views that are stored queries. The query used to create a view can be very simple (query a single table with a WHERE clause) or pretty complex (join a number of tables and apply many filters). Views do not store data. When we execute a query against a view, the underlying tables are queried.

Views can serve quite a few useful purposes. We can use views as a security measure to ensure that users get to see only a subset of data (specific columns or records). For this, we can create views and provide users access to the views, and not to the underlying tables.

Views can also be used to make it easy to retrieve data in cases where it might be necessary to write complex queries. We can create a view using the complex query and then a simple SELECT option against the view is all it takes to get the data we want.

Access to tables referenced in the view is determined by permissions of the view owner. In some cases, this can be used to provide secure but restricted access to the underlying tables. Therefore the user of a view must have permissions to call all functions used by the view.

Simple views are automatically updatable: the system will allow INSERT, UPDATE and DELETE statements to be used on the view in the same way as on a regular table. A view is automatically updatable if it satisfies all of the following conditions:

- The view must have exactly one entry in its FROM list, which must be a table or another updatable view.
- The view definition must not contain WITH, DISTINCT, GROUP BY, HAVING, LIMIT, or OFFSET clauses at the top level.
- The view definition must not contain set operations (UNION, INTERSECT or EXCEPT) at the top level.
- The view's select list must not contain any aggregates, window functions or set-returning functions.

An automatically updatable view may contain a mix of updatable and non-updatable columns. A column is updatable if it is a simple reference to an updatable column of the underlying base relation; otherwise the column is read-only, and an error will be raised if an INSERT or UPDATE statement attempts to assign a value to it.

## 3.2 MATERIALIZED VIEW

Materialized views are similar to views because they also depend on other tables for their data, although there are a couple of differences. SELECTs against views will always fetch the latest data, whereas SELECTs against materialized views might fetch stale data.

The other key difference is that materialized views actually contain data and take up storage space (proportionate to the volume of data), whereas views do not occupy significant space on disk. Materialized views are used mostly to capture summaries or snapshots of data from base tables. Some latency/staleness is acceptable.

When we use materialized views to store precalculated aggregates, there are two advantages.

First, we avoid the overhead of doing the same calculation multiple times. Materialized views that provide summaries tend to be small compared to the base tables. So, we will save on the cost of scanning big tables. This is the second advantage.

When we use materialized views to store data from foreign tables, we make the query performance more predictable. We also eliminate the data transfer that occurs when we access foreign tables multiple times.

Current limitations of materialized views include:

- You can’t use CREATE OR REPLACE to edit an existing materialized view. You must drop and re-create the view even for the most trivial of changes. Use DROP MATERIALIZED VIEW name_of_view. Annoyingly, you’ll lose all your indexes.
- You need to run REFRESH MATERIALIZED VIEW to rebuild the cache. PostgreSQL doesn’t perform automatic recaching of any kind. You need to resort to mechanisms such as crontab, pgAgent jobs, or triggers to automate any kind of refresh.
- Refreshing materialized views is a blocking operation, meaning that the view will not be accessible during the refresh process. You can lift this quarantine by adding the CONCURRENTLY keyword to your REFRESH command, provided that you have established a unique index on your view. The trade-off is concurrent refreshes could take longer to complete.

# FOREIGN DATA WRAPPERS

Foreign data wrappers are a standardized way of handling access to remote objects from SQL databases.

We can use PostgreSQL's foreign data wrappers to access data from various data sources including:

- Relational databases (such as Oracle and MySQL)
- NoSQL databases (such as MongoDB, Redis, and CouchDB)
- Various types of files

Queries against such sources do not have predictable performance. We can create materialized views on the foreign tables created on such data sources.

There are two foreign data wrappers that ship with PostgreSQL:

- file_fdw to create foreign tables that represent flat files (Postgres 9.1 and later),

- postgres_fdw to create foreign tables that represent tables in another PostgreSQL database (Postgres 9.3 and later).

You can also define your own wrapper, or use a third party wrapper.

The steps usually involve:

1. Installing the extension.

2. Creating a server object. This provides details of the server (such as IP and

port or path).

3. Creating user mapping. This involves providing the authentication information to connect to other databases. This step may or may not be necessary depending on the type of server object.

4. Creating the foreign table. This creates a table that maps to another table or file on the server object.

# SOURCE BOOKS AND ARTICLES

1. PostgreSQL 13.4 Documentation, The PostgreSQL Global Development Group, 1996–2021
2. Regina O. Obe and Leo S. Hsu, PostgreSQL: Up and Running THIRD EDITION, O'Reilly Media, 2018
3. Jayadevan Maymala, PostgreSQL for Data Architects, Packt Publishing, 2015
4. Hans-Jürgen Schönig, Mastering PostgreSQL 12 Third Edition, Packt Publishing, 2019
5. PostgreSQL 10 High PerformanceIbrar Ahmed, Gregory Smith, Enrico Pirozzi, 2018
6. Indexes in PostgreSQL, postgrespro.com/blog/pgsql/