**Index Optimization.**

B-tree indexes:

B-tree is the default index in Postgres and is best used for specific value searches, scanning ranges, data sorting or pattern matching.

***Syntax:***

CREATE INDEX idx\_tree1 on table(column\_name);

Hash indexes:

Hash indexes are best suited to work with equality operators. The equality operator looks for the exact match of data.

***Syntax:***

CREATE INDEX idx\_hash1 on table using HASH(column\_name);

GiST indexes:

The Generalized Search Tree (GiST) is balanced, and it implements indexing schemes for new data types in a familiar balanced tree structure. It can index complex data such as geometric data and network address data.

***Syntax:***

create index on table using gist(column\_name);

SP-GiST indexes:

The SP-GiST index refers to a space partitioned GiST index. It is useful for indexing non-balanced data structures using the partitioned search tree.

***Syntax:***

create index on table using spgist(column\_name);

GIN indexes:

The Generalized Inverted Index (GIN) is beneficial for indexing columns that have composite types. It is best suited for data types such as JSONB, Array, Range types and full-text search.

***Syntax:***

CREATE EXTENSION pg\_trgm;

CREATE INDEX IDX\_GIN on table USING gin(column\_name gin\_trgm\_ops);

BRIN index:

The BRIN index is also known as Block Range Index. It stores the summary of blocks (minimum value and maximum value).

***Syntax:***

create index idx\_brin1 on table USING BRIN(column\_name);

Common Mistakes When Using Indexes in PostgreSQL:

1. Creating Indexes on Every Column: Indexes are not free. They consume additional disk space and add overhead to write operations. When a new row is inserted or an existing one is updated, all indexes on the affected columns must be updated. This can lead to increased transaction times and reduced throughput. It’s crucial to analyze the query patterns and create indexes only on columns that are frequently used in WHERE clauses, JOIN conditions, or ORDER BY statements.
2. Not Considering the Selectivity of Columns: The selectivity of a column refers to the proportion of unique values it contains. High selectivity means more unique values, making the index more effective in narrowing down search results. Conversely, indexing columns with low selectivity, such as those with many repeated values, will not be as beneficial. For instance, indexing a column that stores gender, which typically has very few unique values, is unlikely to improve performance and is a waste of resources.
3. Using the Wrong Index Type: PostgreSQL provides several index types, each optimized for different data patterns and query types. The default B-tree index is suitable for general purposes, but for specific use cases, other types like Hash, GiST, SP-GiST, GIN, or BRIN may be more appropriate. For example, GIN indexes are ideal for indexing array data and full-text search, while BRIN indexes are efficient for large tables with naturally ordered data. Using the wrong index type can lead to suboptimal performance and increased storage requirements.
4. Ignoring the Cost of Index Maintenance: Indexes need to be maintained as data changes. This maintenance has a cost, particularly in write-heavy databases where the write amplification due to indexes can be significant. Frequent updates and deletions can lead to index bloat, where the index takes up more space than necessary, and can degrade performance. Regular maintenance tasks like VACUUM and REINDEX can mitigate some of these issues, but the cost-benefit ratio of each index should always be considered.
5. Overlooking the Importance of Statistics: PostgreSQL uses statistics to determine the most efficient way to execute a query. These statistics, collected by the ANALYZE command, provide information about the distribution of data within a table. If the statistics are not up-to-date, PostgreSQL might choose a less-than-ideal execution plan. For example, it might use a sequential scan instead of an index scan if it underestimates the number of rows returned by a query. Regularly updating statistics ensures that the query planner has accurate information to work with.

***Indexes on Expressions:***

index is defined on the result of a function applied to one or more columns of a single table. This feature is useful to obtain fast access to tables based on the results of computations.

Example:

postgres=# create index test1\_lower\_col1\_idx ON cust(lower(firstname));

CREATE INDEX

postgres=# explain analyze select \* from cust where lower(firstname)='jose';

***Considerations:***

1. Index expressions are relatively expensive to maintain, beca derived expression(s) must be computed for each row insertion and [non-HOT update.](https://www.postgresql.org/docs/16/storage-hot.html)
2. index expressions are not recomputed during an indexed search since they are already stored in the index.
3. indexes on expressions are useful when retrieval speed is more important than insertion and update speed.

***Partial Indexes:***

A partial index is an index built over a subset of a table; the subset is defined by a conditional expression (called the predicate of the partial index).  This selective indexing strategy is particularly useful for queries that target a specific subset of data, offering a more efficient alternative to full-table indexes.

Example:

create table orders (custid int,billed boolean not null,amount int);

CREATE INDEX orders\_unbilled\_index ON orders (amount) WHERE billed is not true;

explain analyze SELECT \* FROM orders WHERE billed is not TRUE AND amount < 75000;

explain analyze SELECT \* FROM orders WHERE billed is not TRUE AND custid>100;

***Considerations:***

1. Reduced Index Size: Partial indexes index fewer rows, resulting in a smaller index size and less disk space usage.
2. Faster Index Maintenance: Smaller indexes require less time to update when data changes, leading to better overall performance.
3. Use Partial Indexes for Frequent Conditions: Identify common query conditions and create partial indexes to support them.

***Multi-Column Indexes:***

Multi-column indexes, also known as composite indexes, are created on two or more columns of a table. They are effective when queries involve conditions on multiple columns, allowing the database engine to quickly filter and sort data based on the combined index keys. Currently, only the B-tree, GiST, GIN, and BRIN index types support multiple-key-column indexes.

Example:

create table staff(firstname varchar,lastname varchar,salary int,email varchar);

create index idx\_staff1 on staff(firstname,salary);

explain analyze select \* from staff where firstname='kasey' and salary=20880;

explain analyze select \* from staff where firstname='kasey' and salary=20880 AND email='alonso@gmail.com';

explain analyze select \* from staff where salary=20880;

explain analyze select \* from staff where salary=20880 and firstname='kasey';

***Considerations:***

* Improved Query Speed: Multi-column indexes can eliminate the need for separate single-column index lookups, speeding up query execution.
* Combine Frequently Used Columns: When multiple columns are often used together in queries, consider creating a multi-column index.
* Balance Index Benefits and Costs: While indexes can speed up queries, they also require maintenance. m judiciously to avoid unnecessary overhead.
* When constructing a multi-column index, the sequence in which columns are arranged plays a pivotal role. The database engine prioritizes the leading (leftmost) columns when executing filters.
* Limit the number of columns in a multi-column index to prevent performance degradation due to increased index size.
* For tables with extensive rows, contemplate creating several multi-column indexes on varying column subsets.
* Continuously evaluate index performance and reconstruct them as needed.

***Null Value Considerations for B-Tree:***

B-tree indexes are structured to store data in a sorted manner and NULL values are considered to be less than any other value, resulting in their placement at the start of the index. The presence of NULL values at the beginning of a B-tree index can introduce inefficiencies, particularly when queries predominantly target non-NULL entries.

Example:

CREATE INDEX test2\_info\_nulls\_low ON test2 (info NULLS FIRST);

CREATE INDEX test3\_desc\_index ON test3 (id DESC NULLS LAST);

***Considerations:***

* Index Type Selection: opt for index types that manage NULL values more effectively. For example, hash indexes or GiST indexes might be preferable over B-tree indexes for columns with numerous NULL values.
* Column Order in Multi-Column Indexes: In multi-column indexes, position columns likely to contain NULL values towards the end. This arrangement reduces the performance impact since the index can leverage the non-NULL columns more efficiently.
* Partial Indexes: Create partial indexes that exclude NULL values altogether. This can be particularly useful when queries frequently exclude NULL values.
* Default Values: If applicable, consider setting a default value for columns instead of allowing NULL, ensuring all rows contribute to the index’s order.

***Covering Index:***

 An index specifically designed to include the columns needed by a particular type of query that you run frequently.  Since queries typically need to retrieve more columns than just the ones they search on, PostgreSQL allows you to create an index in which some columns are just “payload” and are not part of the search key. This is done by adding an INCLUDE clause listing the extra columns.

Example:

explain analyze select bid,tid,tbalance from pgbench\_tellers where bid=24;

create index idx\_test1 on pgbench\_tellers(bid) INCLUDE(tid,tbalance);

***Considerations:***

1. INCLUDE clause can also be written in UNIQUE and PRIMARY KEY constraints the uniqueness condition applies to just the main column (bid) and not to (tid,balance).
2. Planner could handle these queries as index-only scans, because tid,balance can be obtained from the index without visiting the heap.
3. only B-tree, GiST and SP-GiST indexes currently support included columns.