“Develop a passion for learning.”

5. Performance Optimization

Introduction to PostgreSQL

AGENDA

- SQL raw storage format
- The query planner
- Queries and code optimization
- Indexing
- Schema design

RAW SQL STORAGE

- For certain database objects like views, rules, and functions
  - PostgreSQL stores the original SQL text used to define these objects as raw SQL strings in specific system catalog tables.
- This raw SQL text is preserved so that PostgreSQL can reproduce the object definitions
- For many objects, the raw SQL is not stored, tables for example because
  - Storing tables as structured metadata allows faster access and managing of table information without parsing raw SQL every time.
  - Changes to table structures, such as adding columns or modifying constraints, can be efficiently managed at the metadata level without the need to re-parse or reconstruct SQL.
  - Managing table structures through metadata ensures that all dependencies and internal data structures are consistently maintained.

RAW SQL STORAGE

- Why use raw SQL storage?
  - Primarily for object not stored like tables in an internal schema format
  - Tools like pg_dump rely on raw SQL definitions to accurately describe some objects
    - *When exporting a database or when it is restored or replicated, the objects are recreated exactly as they were originally defined.*
  - Objects like views, rules, and functions often contain complex logic which is straightforward to reference, modify, recompile or duplicate from the raw SQL.

RAW SQL STORAGE

- What is not in raw SQL – some examples
- Table definitions
- Indexes
  - Indexes are highly optimized data structures that rely on detailed internal representation to support efficient lookups.
- Constraints and Triggers
  - Storing constraints as metadata, allows enforcing data integrity directly at the storage level without needing to refer back to SQL definitions.
- DML Commands (INSERT, UPDATE, DELETE, SELECT)
  - DML commands are intended to manipulate data and do not define persistent database objects, so there's no need to store their SQL.

PROCESSING QUERIES

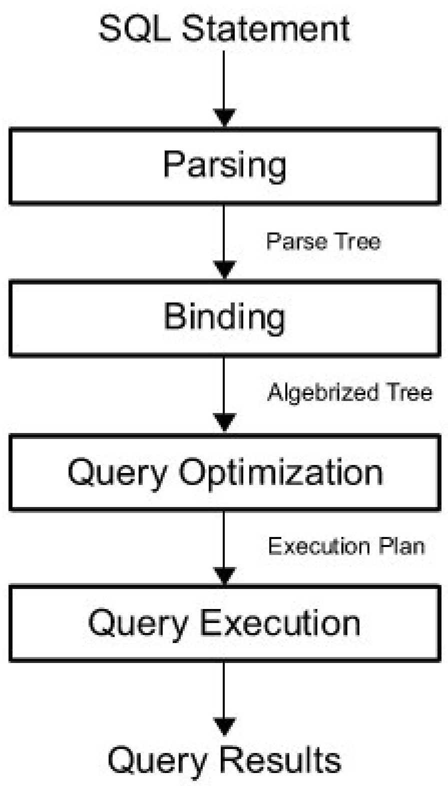
- The main stages of processing a query are:
- Parsing
  - The parser, handles examines the SQL text and verifies whether it is correct
  - If the statement gas any syntax errors, the processing aborts.
  - Otherwise, the parser disassembles the statement into its main parts
    - *The list of tables in the query, the referenced columns, the clauses to filter data, sorting,etc*
  - The output is a parse tree, a structure representing the query
- Binding or Rewriting
  - This applies any syntactic rules to rewrite the original SQL statement into what will be effectively executed.
    - *For example, views do not support updates, so updates to a view are redirected to update the underlying table*
  - Rewrite rules only apply to views and tables

PROCESSING QUERIES

- Optimization
  - Responsible for finding the fastest path to the data.
  - Decides which of the available access methods, such as indexes or direct access, should be used to get to the data.
  - Uses cost based optimization where each possible query strategy is assigned a cost based on time and resource usage
- Execution.
  - Handled by the executor component
  - Responsible for effectively going to the storage and retrieving (or inserting) the data

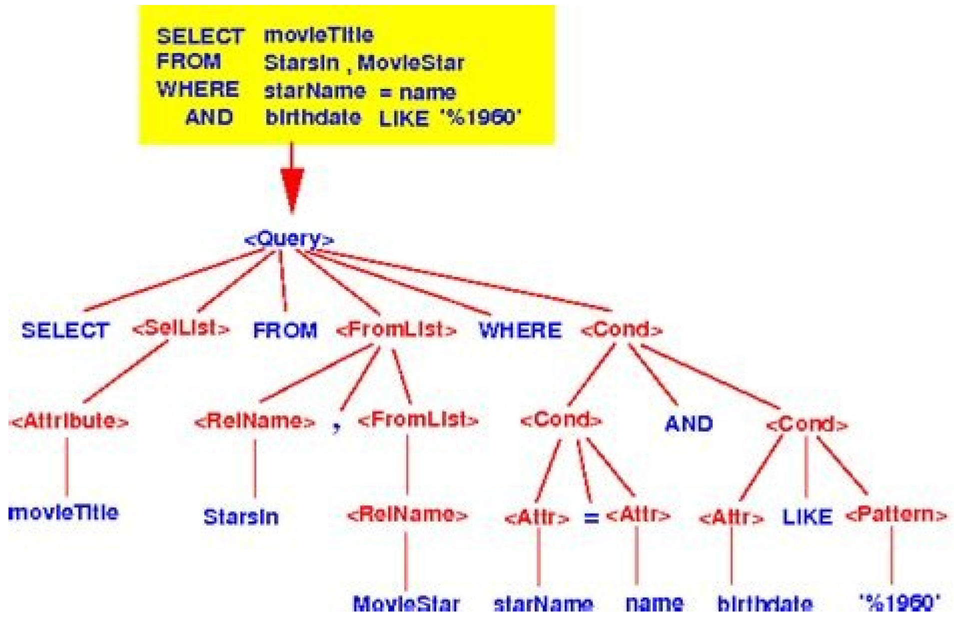
QUERY OPTIMIZATION

- Query optimizers analyze SQL queries and find the best execution plan
- Best means minimizing the resources used, such as time, memory, and CPU
- And minimizing the time return a result
- This is a standard component of any SQL or similar system that queries data
- For example
  - A join on 100,000 records followed by a select that only keep 1,000 records
  - Reversing the order of operations means much less data is used in the join



OPTIMIZATION STEPS

- Parsing the Query:
  - SQL is parsed into a tree to understand its structure and what data is being requested.
  - This step includes checking for syntax errors and creating a preliminary plan called a parse tree.
- Generating Possible Execution Plans
  - The optimizer generates several possible logical execution plans.
  - The plans try different methods to access and join tables, such as using indexes, performing full table scans, or joining tables in different orders.
  - The query may be rewritten to make it more efficient



OPTIMIZER NODES

- The optimizer divides the actions for the executor into nodes
  - A node is an action to execute to produce the final or an intermediate result
  - A different nodes exists for every operation that can be performed and access method
  - Stackable: The output of a node can be used as the input to another node.
  - Allows complex execution plans made by assembling nodes
- Sequential nodes
  - Executed sequentially
    - *Sequential Scan*
    - *Index Scan, Index-Only Scan, and Bitmap Index Scan*
    - *Nested Loop, Hash Join, and Merge Join*
    - *The Gather and Merge parallel nodes.*
  - Sequential Scan (Seq Scan)
    - *Used when there is no other effective alternative.*
    - *Executor will read all the data one block after the other in sequential order.*

OPTIMIZER NODES

- Index Scan
  - *Reads from the chosen index, then reads data from the storage.*
- Index-Only Scan
  - *If the requested data only involves columns in the index, the second read from storage is omitted*
- Bitmap Index Scan
  - *retrieves row locations (pointers to rows, called ctid) using an index but doesn't fetch the actual table rows immediately.*
  - *The results are stored in a bitmap, where each bit corresponds to a potential row in a table. A bit is set to 1 if the corresponding row satisfies the condition.*
  - *More fficient than a regular index scan for queries where multiple indexes or large ranges of rows are involved.*

- Join nodes
  - For joins between two (or more) tables, there are three possible nodes.
  - Nested Loop:
    - *Both tables are scanned and every tuple is checked to see whether there is a match.*
    - *Essentially, the scan is a set of nested for-loops*
    - *Used only if the inner table is small enough so that looping every time over it is efficient*

OPTIMIZER NODES

- Hash Join:
  - *The inner table is mapped into a set hash buckets*
  - *The outer table is then walked and to see if each tuple matches a hash.*
- Merge Join:
  - *Both tables are first sorted by the join key(s), and then walked sequentially.*
  - *For every tuple of the outer table, all the tuples that match in the inner table are extracted.*
  - *Since both tables are sorted, a non-matching tuple increments the join key.*

- Parallel nodes
  - Can distribute the work among parallel processes
  - Setup time is required
  - Parallel execution of certain nodes is usually only if the estimated parallel version will provide a benefit over sequential execution

OPTIMIZER NODES

- Gather nodes
  - Used to consolidate the results of parallel nodes
  - Gather nodes just assemble the sets of results
  - Gather Merge nodes requires each set to be sorted so it can assemble a sorted result
- Parallel scans
  - Parallel version of the sequential scans
- Parallel joins
  - Tries to keep the inner table accessed in a non-parallel way and performs parallel access to the outer table
    - *Uses one of the nodes presented in the last section.*
  - Hash Join
    - *A;lows a hash map of the inner table to be computed in parallel by every process working on the outer table.*
- Parallel aggregations
  - Final result set is made by the aggregation of different parallel subqueries, there must
    - *There is a Partial Aggregate node , for each parallel process for a partial result set*
    - *A Gather node (or Gather Merge) collects all the partial results*
    - *Passed to the Finalize Aggregate node, which sequentially assembles the final result.*

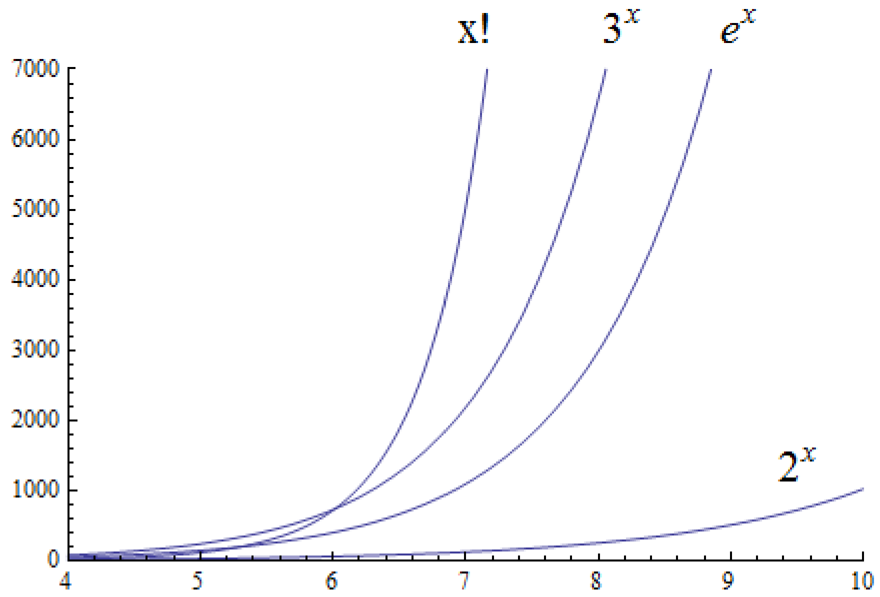
NODE COSTS

- Every node has a cost
  - Estimation of how expensive, in terms of computational resources, the execution of the node will be.
  - PostgreSQL provides a list of costs, expressed in arbitrary units, for the main type of operations that a node can perform.
  - Node cost is the per-unit cost of the single operations node performs multiplied the number of times performed
    - *Depends on the size of the data that the node has to evaluate.*
  - The unit costs can be set in n the postgresql.conf main file or the pg_settings catalog.
  - The costs can be queried

```
rod=# SELECT name, setting
FROM pg_settings
WHERE name LIKE 'cpu%\_cost'
OR name LIKE '%page\_cost'
ORDER BY setting DESC;
         name         | setting
----------------------+---------
 random_page_cost     | 4
 seq_page_cost        | 1
 cpu_tuple_cost       | 0.01
 cpu_index_tuple_cost | 0.005
 cpu_operator_cost    | 0.0025
(5 rows)
```

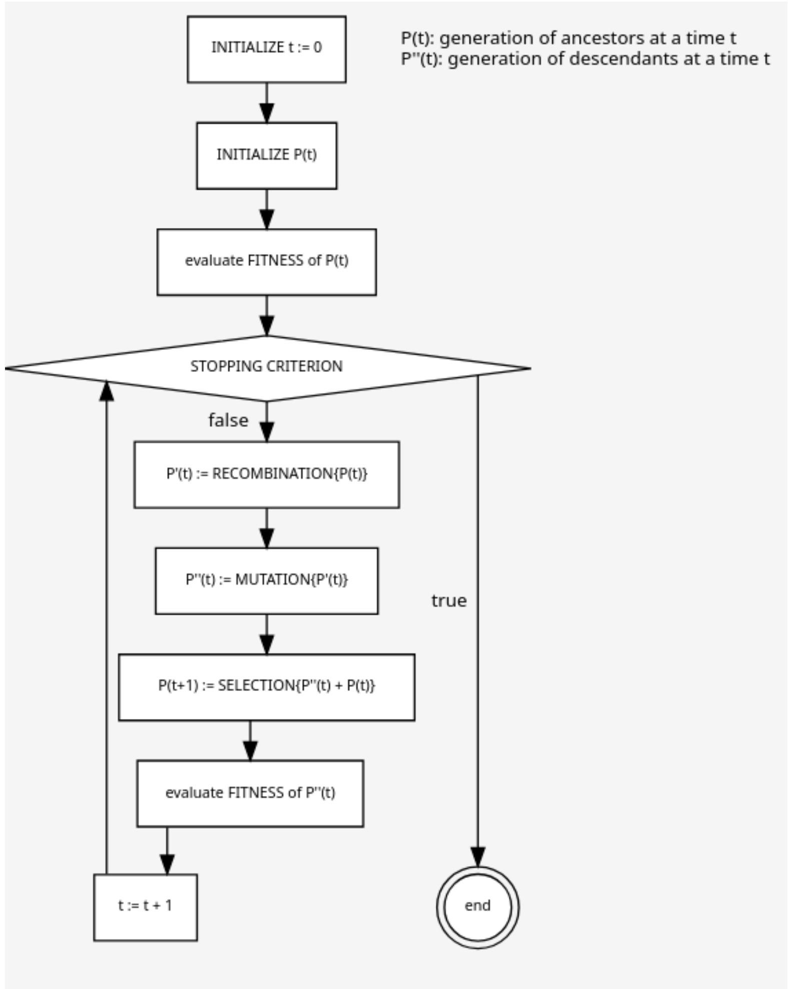
THE JOIN PROBLEM

- The problem of efficiently executing SQL queries that involve joins between multiple tables
  - As the number of tables in a join increases, the number of possible join orders grows at a factorial rate (x! In the diagram)
  - With three tables, there are only a few join orders to consider, but with ten tables, there are over three million possible join combinations.
  - Many potential join orders need to be considered to find the most efficient one
  - This can become computationally expensive and time-consuming



GENETIC QUERY OPTIMIZATION (GEQO)

- The GEQO planning process uses the standard planner code to generate plans for scans of individual relations.
  - Then join plans are developed using the genetic approach.
  - This process is inherently nondeterministic, because of the randomized choices made during both the initial population selection and subsequent “mutation” of the best candidates.



SQL QUERY PLANNING

- The application program transmits a query to the server and waits to receive the results sent back by the server.
- The parser stage checks the query transmitted by the application program for correct syntax and creates a query tree.
- The rewrite system takes the query tree created by the parser stage and looks for any rules (stored in the system catalogs) to apply to the query tree.
  - It performs the transformations given in the rule bodies.
  - One application of the rewrite system is in the realization of views. Whenever a query against a view (i.e., a virtual table) is made, the rewrite system rewrites the user's query to a query that accesses the base tables given in the view definition instead.
- The planner/optimizer takes the (rewritten) query tree and creates a query plan that will be the input to the executor.
  - It does so by first creating all possible paths leading to the same result.
  - For example if there is an index on a relation to be scanned, there are two paths for the scan.
  - One possibility is a simple sequential scan and the other possibility is to use the index.
  - Next the cost for the execution of each path is estimated and the cheapest path is chosen. The cheapest path is expanded into a complete plan that the executor can use.

SQL QUERY PLANNING

- The executor recursively steps through the plan tree and retrieves rows in the way represented by the plan.
  - The executor makes use of the storage system while scanning relations, performs sorts and joins, evaluates qualifications and finally hands back the rows derived.

DEEP DIVE

- This is an optional deep dive into the PostgreSQL rule system

EXPLAIN

- The EXPLAIN command shows the steps that the database will take to retrieve or modify the data as requested by the query.
  - Using EXPLAIN can identify potential performance issues and optimize queries accordingly.
- What EXPLAIN does?
  - Shows the execution plan chosen by the query optimizer, including the sequence of operations (e.g., scans, joins, sorts) and their order.
  - Shows estimated costs for each operation, including startup and total costs, which reflect resource usage (like CPU and I/O)
  - Provides estimates of the number of rows processed by each step.
  - When used with ANALYZE, it provides the actual execution times and row counts, which are useful for comparing the optimizer's estimates with real performance.
- The lab works with an actual example.

LAB 5

- The lab description and documentation is in the Lab directory in the class repository

OPTIMIZATION TIPS

- Use Proper Indexing:
  - Create Indexes on frequently queried columns: Index columns used in WHERE clauses, joins, and ORDER BY clauses.
  - Use Composite Indexes: When multiple columns are frequently queried together, consider composite indexes.
  - Use Partial Indexes: Index only the relevant subset of rows to reduce index size and improve performance.
  - Avoid Over-Indexing: Too many indexes can slow down INSERT, UPDATE, and DELETE operations.
- Optimize Joins:
  - Choose the Right Join Type: Use inner joins for matching rows and outer joins only when necessary.
  - Use Indexed Columns in Joins: Ensure join columns are indexed to speed up join operations.
  - Reduce Join Complexity: Simplify joins by breaking down complex queries or reducing the number of tables involved.

OPTIMIZATION TIPS

- Leverage Query Planning and Execution Tools:
  - Use EXPLAIN and EXPLAIN ANALYZE: Review execution plans to understand how queries are executed and identify performance bottlenecks.
  - Check for Sequential Scans: Ensure that large table scans are necessary and consider indexing to avoid them.
- Write Efficient SQL:
  - Avoid SELECT *: Specify only the columns you need to reduce I/O and processing time.
  - Use Subqueries and CTEs Judiciously: While useful, complex subqueries and Common Table Expressions (CTEs) can sometimes be replaced with simpler joins or views.
  - Avoid Unnecessary Calculations: Avoid repetitive calculations within queries by computing values once or using derived columns.

OPTIMIZATION TIPS

- Use Appropriate Data Types:
  - Choose the Right Data Type: Use appropriate data types for columns to optimize storage and performance.
  - Avoid Over-Sized Data Types: Use INTEGER instead of BIGINT when appropriate, and avoid using large text types if not necessary.
- Optimize Filtering and Sorting:
  - Push Down Filters: Apply filters as early as possible to reduce the data set size in intermediate steps.
  - Use Indexes for Sorting: Ensure that sorting operations (ORDER BY) can utilize indexes when possible.

OPTIMIZATION TIPS

- Improve Transaction and Locking Strategies:
  - Minimize Lock Contention: Use shorter transactions and minimize the scope of locks to reduce contention.
  - Use Appropriate Isolation Levels: Choose the correct transaction isolation level (READ COMMITTED, REPEATABLE READ, etc.) based on application requirements.
- Optimize Updates and Inserts:
  - Batch Inserts and Updates: Use bulk operations instead of individual row operations to reduce overhead.
  - Use COPY for Bulk Loads: The COPY command is much faster than INSERT for loading large volumes of data.

OPTIMIZATION TIPS

- Tune Memory and Configuration Settings:
  - Adjust work_mem: Set appropriate work_mem for complex queries to allow enough memory for sorting and joining.
  - Set shared_buffers: Allocate sufficient memory to shared_buffers to store frequently accessed data in memory.
  - Configure maintenance_work_mem: Set a higher value for maintenance operations like vacuuming and indexing.
- Regular Maintenance:
  - Vacuum and Analyze Regularly: Run VACUUM and ANALYZE to update statistics and reclaim space.
  - Reindex as Needed: Periodically reindex tables to maintain index efficiency, especially after large data changes.

OPTIMIZATION TIPS

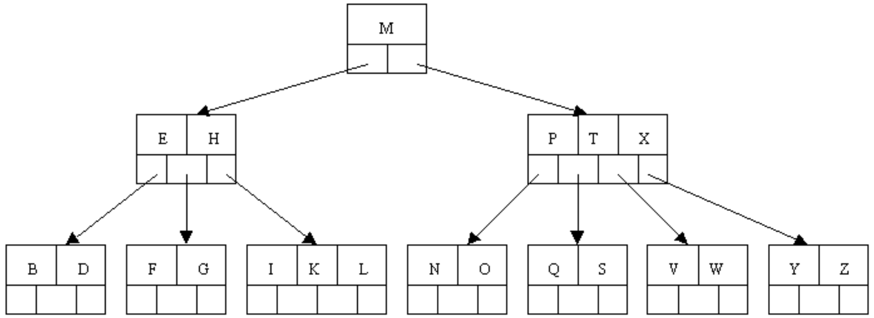
- Partition Large Tables:
  - Use Table Partitioning: Divide large tables into smaller, manageable partitions to improve query performance and manageability.
- Leverage Materialized Views:
  - Use Materialized Views: Precompute and store results of complex queries that do not change frequently to speed up access.
- Optimize Network and I/O:
  - Reduce Network Overhead: Minimize data transferred over the network by fetching only necessary rows and columns.
  - Use Connection Pooling: Use connection pooling to reduce the overhead of establishing database connections.

OPTIMIZATION TIPS

- Avoid Using Loops in PL/pgSQL:
  - Use Set-Based Operations: Prefer set-based SQL operations over looping row by row in PL/pgSQL functions.
- Monitor and Profile Queries:
  - Use PostgreSQL Monitoring Tools: Utilize tools like pg_stat_statements, pgBadger, or other performance monitoring solutions to identify slow queries and resource bottlenecks.

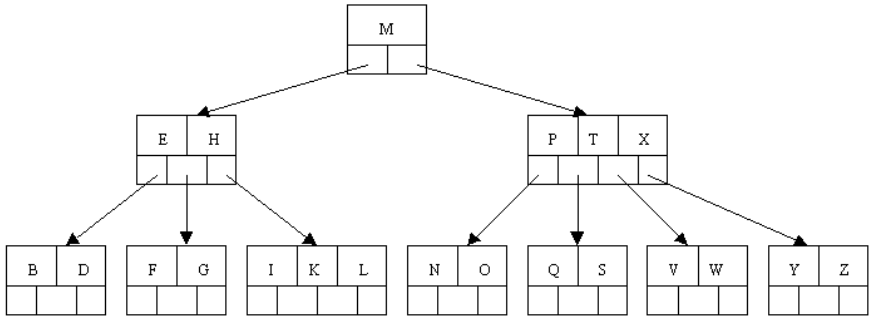
INDEXES

- PostgreSQL has several types of indexes, each suited to different use cases and query patterns
- B-Tree index
  - Are the default and most commonly used index type in PostgreSQL.
  - Work well for r equality (=), range (<, >, BETWEEN), and pattern matching (LIKE) queries with non-wildcard patterns
  - Optimal for primary key, unique constraints, and general-purpose indexing on columns used in WHERE, ORDER BY, and join conditions.



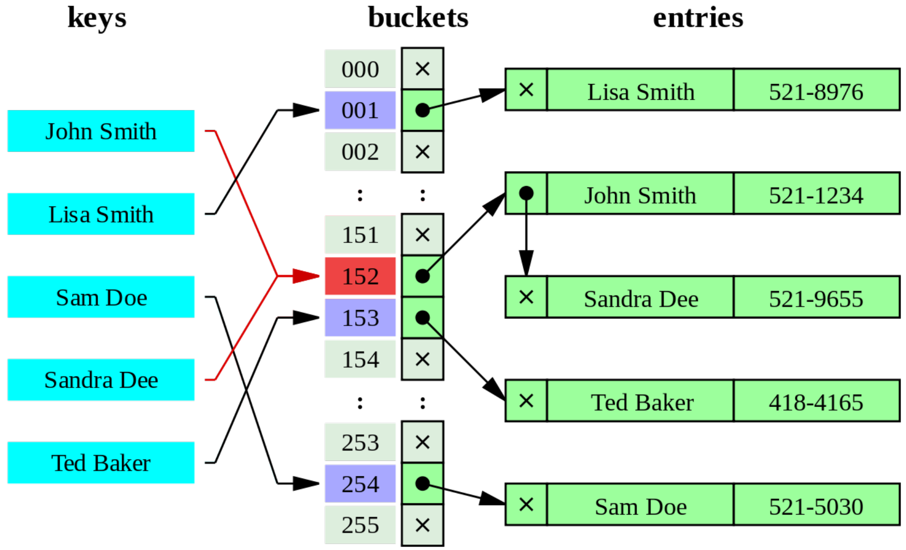
INDEXES

- Performance Tips:
  - Use B-tree indexes on frequently queried columns with a high level of distinct values.
  - Avoid using B-tree indexes on columns with many repeated values which will impact performance.



INDEXES

- Hash Index
  - Hash indexes are used specifically for equality comparisons (=).
  - Generally faster than B-tree indexes for equality lookups but do not support range queries.
  - Ideal for indexing columns that are frequently queried with equality conditions.
- Performance Tips:
  - Use hash indexes when the primary use case involves equality searches.
  - Hash indexes require additional maintenance operations (e.g., REINDEX) after crashes, as they are not WAL-logged.



INDEXES

- Generalized Inverted Index
  - Useful for containment queries (@>, <@) and for searching elements within arrays or documents.
  - Optimal for full-text search (tsvector), JSONB data types, and columns storing arrays or other composite data types.
  - Use GIN indexes when querying JSONB, array elements, or for full-text search scenarios.
  - Be mindful of the higher maintenance cost compared to B-tree indexes.

INDEXES

- Generalized Search Tree
  - GiST indexes support a wide range of queries, including geometric data types (e.g., points, polygons), full-text search, and custom data types with complex queries.
  - Use for nearest-neighbor searches, spatial data, and range queries on complex data types.
  - Use GiST indexes for spatial and geometric queries.
  - Custom operator classes allow GiST indexes to be used for various complex data types, but they may have higher space usage.

INDEXES

- Block Range Index
  - BRIN indexes are compact and efficient for indexing large, sequential data sets.
  - They work by summarizing data within physical blocks, making them suitable for columns with ordered or clustered data.
  - Best for very large tables where columns exhibit a natural ordering (e.g., timestamps, sequential IDs).
  - Use BRIN indexes for very large datasets with naturally ordered data.
  - They are space-efficient and have low maintenance costs but are less precise than B-tree indexes.

PARTIAL INDEXES

- Partial indexes are indexes that cover only a subset of rows based on a specified condition. They can be any of the previous index types but with a WHERE clause to limit indexed rows.
  - Ideal when only a subset of the data is frequently queried.
  - CREATE INDEX idx_active_employees ON employees (name) WHERE active = true;
  - Use partial indexes to reduce index size and maintenance by focusing on the most relevant data.
- Ensure the index condition matches frequently queried conditions to maximize performance benefits.

PERFORMANCE OPTIMIZATION

- Keep Statistics Updated:
  - Regularly run ANALYZE to update table and index statistics so the query planner can make informed decisions.
- Monitor Index Usage:
  - Use tools like pg_stat_user_indexes to monitor index usage and identify unused or redundant indexes.
- Balance Index Overhead:
  - Avoid excessive indexing, which can slow down write operations (inserts, updates, deletes).
- Use EXPLAIN to Analyze Queries: Regularly use EXPLAIN or EXPLAIN ANALYZE to check if your queries are using indexes effectively

SCHEMA OPTIMIZATION

- Schema optimization is designing and refining the database schema to improve the performance, scalability, and maintainability of the database.
- Normalization and Denormalization:
  - Normalization involves organizing tables and columns to reduce data redundancy and improve data integrity by dividing large tables into smaller, related tables.
  - Denormalization pre-joins often joined tables to reduce the number of joins required in queries.
- Proper Indexing:
  - Identifying which columns need indexing (e.g., primary keys, foreign keys, frequently queried columns) and choosing the right type of index (e.g., B-tree, hash, GIN, GiST).
  - Over-indexing can degrade performance on write operations (INSERT, UPDATE, DELETE) which might make indexes stale and need to recreated
- Data Types and Constraints:
  - Use data types for columns can save storage space and improve query performance.
  - For example, using INTEGER instead of BIGINT where appropriate, or VARCHAR(50) instead of TEXT when lengths are predictable.
  - Constraints (e.g., NOT NULL, UNIQUE, CHECK) ensure data integrity and can sometimes optimize query execution plans.

SCHEMA OPTIMIZATION

- Partitioning:
  - Dividing large tables into smaller, more manageable pieces along some value like date range improves query performance by limiting the amount of data that needs to be scanned.
- Foreign Keys and Relationships
  - Properly defining foreign keys and relationships between tables ensures data consistency and helps the optimizer understand the relationships between tables, which can influence query planning.
  - In some high-performance scenarios, foreign keys might be omitted to avoid the overhead of maintaining referential integrity, but this requires careful manual management.
- Schema Design for Query Patterns
  - Understanding common query patterns can guide schema design.
  - For example, if a table is frequently joined on a specific column, ensuring that column is indexed and optimizing the data model for those joins is crucial.
  - Designing tables with consideration for read-heavy vs. write-heavy workloads can also lead to optimizations specific to the expected load.

SCHEMA OPTIMIZATION

- Avoid Over-Engineering:
  - Use simple, clear schema rather than an overly complex one.
  - Over-engineering (e.g., too many tables, unnecessary normalization) can lead to complicated joins and maintenance challenges.
- Using Materialized Views:
  - For frequently accessed, complex queries, using materialized views can improve performance.
  - These are precomputed views stored on disk that can be refreshed periodically.

End Module