CS60203: Design Optimization of Computing Systems *October 27, 2024*

Database and Optimizing Storage

Instructor: Mainack Mondal *Scribe-By: <Tanishq Prasad >*

1 When to use a Database?

- When the underlying data has some inherent structure
- When you want to do some computation depending on the data (recent as well as historical)
- When you want some guarantees on the data (e.g., consistency, durability)

Start out with flat files then proceed in the order: In-Memory Database, SQLite and finally SQL to decide the best fit for your application.

Question: *Imagine that you are building a payments app, you store every transaction in some storage system. Need: average of amounts of all the transfers from India to Canada. Do you think it'll be good to store all the amounts together or maybe all the data of a transaction together?*
Amounts together (column-oriented storage) *as each attribute of a transaction (e.g., amount, sender country, receiver country, timestamp) is stored separately in a column, which means more efficient storage for queries focused on aggregates.*

Concluding, the operations you perform on stored data should determine how it is stored.

2 Components of a Database

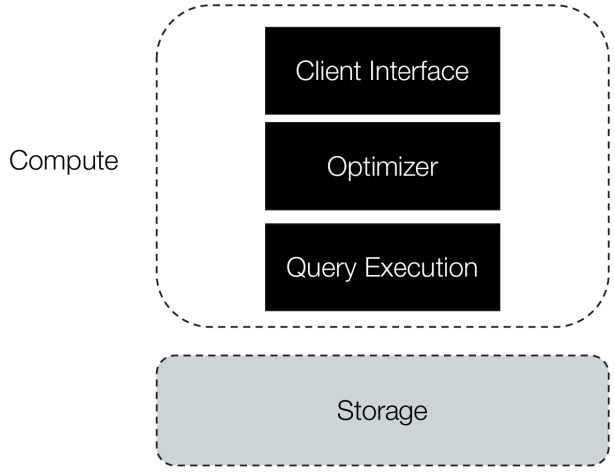

Figure 1: Components of a Database

- **Client Interface:** A user interface that allows for the ability to input queries to a database so it can be anything according to the need.

- **Optimizer:** The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans.
- **Query Execution:** Query executors are components of databases that execute queries and return the results to the application that submitted the query.
- **Storage:** Refers to the way information is organized and stored in a database. Some innovations include object stores for durable storage and distributed storage.

Note: *For a database to be transactional it need not be relational. For example, MongoDB has ACID properties but is not relational.*

3 Database Classification

Database depends on the application it is geared towards (i.e. the kind of guarantees, and queries to be supported). Storage, compute and optimising paradigms are based on that. Majorly they are categorized into:

1. **Transactional Databases (OLTP):** They allow all operations like read, insert, update and delete. They provide strong guarantees like ACID properties. It is application-oriented. Used for business tasks. They help in increasing users and transactions which helps in real-time access to data.

2. **Analytical Databases (OLAP):** OLAP databases are optimized for large scale READs, and aggregation based queries. In general strong guarantees are not necessary, especially not for "ALL" operations. This relaxation can give very strong performance upsides. It is subject-oriented. Used for Data Mining, Analytics, Decisions making, etc. [1]

4 PostgreSQL

4.1 Overview

PostgreSQL is an object-relational database management system. A PostgreSQL server/cluster manages data in multiple databases. Each database contains tables, views and other objects. This is analogous to hierarchical organization in a file system. `CREATE DATABASE` actually works by copying an existing database. PosgtreSQL contains 3 databases on startup:

1. **template1:** By default, PostgreSQL copies the standard system database named `template1`. Thus that database is the "template" from which new databases are made. If you add objects to `template1`, these objects will be copied into subsequently created user databases. This behavior allows site-local modifications to the standard set of objects in databases.

2. **template0:** This database contains the same data as the initial contents of `template1`, that is, only the standard objects predefined by your version of PostgreSQL. `template0` should never be changed after the database cluster has been initialized. By instructing `CREATE DATABASE` to copy `template0` instead of `template1`, you can create a "pristine" user database (one where no user-defined objects exist and where the system objects have not been altered) that contains none of the site-local additions in `template1`.

3. **postgres:** A regular database that you can use at your discretion. It is not a template database. The `postgres` database is also created when the database cluster is initialized.

4.2 System Catalogs

- Metadata of all cluster objects (such as tables, indexes, data types, or functions) is stored in tables that belong to the system catalog. Each database has its own set of tables (and views) that describe the objects of this database.
- Several system catalog tables are common to the whole cluster, but can be accessed from all of them. Most system catalogs are copied from the template database during database creation and are thereafter database-specific.
- The system catalog can be viewed using regular queries, while all modifications in it are performed by DDL commands. The psql client also offers commands for this purpose.
- Names of all system catalog tables begin with `pg_`. Column names start with a three-letter prefix that usually corresponds to the table name, like in `datname`.

4.3 Schemas (Namespaces)

Schemas are namespaces that store all objects of a database. A database contains one or more named schemas, which in turn contain tables. Schemas also contain other kinds of named objects, including data types, functions, and operators. Within one schema, two objects of the same type cannot have the same name. Furthermore, tables, sequences, indexes, views, materialized views, and foreign tables share the same namespace, so that, for example, an index and a table must have different names if they are in the same schema. The same object name can be used in different schemas without conflict. Apart from user schemas, PostgreSQL offers several predefined ones:

- **public:** By default such tables (and other objects) are automatically put into a schema named "public". Every new database contains such a schema.
- **pg_catalog:** It which contains the system tables and all the built-in data types, functions, and operators.
- **pg_toast:** Contains out-of-line TOAST tables. TOAST is the mechanism PostgreSQL uses to store large table attributes "out of line".
- **pg_temp:** These are temporary schemas that exist for the duration of a database session.

By default, users cannot access any objects in schemas they do not own. To allow that, the owner of the schema must grant the `USAGE` privilege on the schema. By default, everyone has that privilege on the schema public. To allow users to make use of the objects in a schema, additional privileges might need to be granted, as appropriate for the object.

A user can also be allowed to create objects in someone else's schema. To allow that, the `CREATE` privilege on the schema needs to be granted. In databases upgraded from PostgreSQL 14 or earlier, everyone has that privilege on the schema public.

4.4 Concurrency [2]

PostgreSQL provides a rich set of tools for developers to manage concurrent access to data. Internally, data consistency is maintained by using a multiversion model (Multiversion Concurrency Control, MVCC). This means that each SQL statement sees a snapshot of data (a database version) as it was some time ago, regardless of the current state of the underlying data. This prevents statements from viewing inconsistent data produced by concurrent transactions performing updates on the same data rows, providing transaction isolation for each database session. MVCC, by

eschewing the locking methodologies of traditional database systems, minimizes lock contention in order to allow for reasonable performance in multiuser environments.

The main advantage of using the MVCC model of concurrency control rather than locking is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading. PostgreSQL maintains this guarantee even when providing the strictest level of transaction isolation through the use of an innovative Serializable Snapshot Isolation (SSI) level.

Table- and row-level locking facilities are also available in PostgreSQL for applications which don't generally need full transaction isolation and prefer to explicitly manage particular points of conflict. However, proper use of MVCC will generally provide better performance than locks. In addition, application-defined advisory locks provide a mechanism for acquiring locks that are not tied to a single transaction.

The SQL standard defines four levels of transaction isolation. The most strict is Serializable, which is defined by the standard in a paragraph which says that any concurrent execution of a set of Serializable transactions is guaranteed to produce the same effect as running them one at a time in some order. The other three levels are defined in terms of phenomena, resulting from interaction between concurrent transactions, which must not occur at each level. The standard notes that due to the definition of Serializable, none of these phenomena are possible at that level.

The phenomena which are prohibited at various levels are:

1. `dirty read`: A transaction reads data written by a concurrent uncommitted transaction.
2. `nonrepeatable read`: A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).
3. `phantom read`: A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.
4. `serialization anomaly`: The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

The SQL standard and PostgreSQL-implemented transaction isolation levels are described in the following table:

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Figure 2: Transaction Isolation Levels

4.5 Performance Tuning[2]

4.5.1 `EXPLAIN`

PostgreSQL devises a query plan for each query it receives. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so

the system includes a complex planner that tries to choose good plans. You can use the `EXPLAIN` command to see what query plan the planner creates for any query.

4.5.2 `JOIN` clauses

The execution order of `JOIN` clauses can have a significant impact on performance. When a query only involves two or three tables, there aren't many join orders to worry about. Therefore, the planner will explore all of them to try to find the most efficient query plan. But the number of possible join orders grows exponentially as the number of tables expands. Beyond ten or so input tables it's no longer practical to do an exhaustive search of all the possibilities, and even for six or seven tables planning might take an annoyingly long time. When there are too many input tables, the PostgreSQL planner will switch from exhaustive search to a genetic probabilistic search through a limited number of possibilities. The switch-over threshold is set by the `geqo_threshold` run-time parameter. The genetic search takes less time, but it won't necessarily find the best possible plan.

4.6 Performance Tips for Populating the Database

- **Disable Autocommit:** Use a single transaction to minimize overhead and ensure atomicity during bulk inserts.
- **Use COPY:** Load data in bulk with the `COPY` command, which is faster than multiple `INSERT` statements.
- **Remove Indexes:** For a new table, load data first, then add indexes to optimize insertion speed.
- **Remove Foreign Key Constraints:** Temporarily remove foreign keys to avoid row-by-row checks, then re-add them after loading.
- **Increase `maintenance_work_mem`:** This speeds up index creation and foreign key constraint addition.
- **Increase `max_wal_size`:** Reduces checkpoint frequency, minimizing disk write overhead during large loads.
- **Disable WAL Archival and Streaming Replication:** Disable these to avoid excessive write-ahead log processing during data loads.
- **Run ANALYZE Afterwards:** Refreshes table statistics, which helps the planner make efficient query decisions.
- **Using `pg_dump`:** For restoring large dumps, use `COPY`, increase memory settings, disable `WAL` features if needed, and consider parallel processing for performance.

References

[1] GFG. Difference between olap and oltp in dbms. https://www.geeksforgeeks.org/difference-between-olap-and-oltp-in-dbms/, September 2024.

[2] The PostgreSQL Global Development Group. Postgresql 17.0 documentation. https://www.postgresql.org/docs/current/index.html.