**SECURITY**

**Role attributes**

**Graphical user interface, text, application

Description automatically generated**

Things to consider: roles, inheritance, privileges, Cascadian revoke

**Password hashing**

* old version Md5 (insecure but kept for compatibility)
* current version SCRAM( Salted Challenge Response Authentication Mechanism)

**PRINCIPLE OF LEAST PRIVILEGE: each user should have the minimum privileges necessary to do the job they need to do**

**AUDITING**

CIA 🡪 **C**onfidentiality **I**ntegrity **A**vailability

Auditing is the monitoring and recording of selected user database actions. It can be based on individual actions, such as the type of SQL statement executed, or on combinations of factors that can include user name, application, time, and so on.

Auditing options:

* Statement-level auditing (postgresql.conf)
* pgAudit extension

\*\* Logging: Logging is an important tool to enable us to conduct effective audits

**Trigger**

Postgres triggers allow to create an audit log for every table in a system.

* A trigger on the table runs whenever data is manipulated
* The trigger writes a record to a separate audit table, which stores all the information about the edit you set to know.

**INDEXING**

An index is a separated data structure that speeds up the data retrieval on a table, it is a pointer to the data in a table.

An index helps to speed up SELECT queries and WHERE clauses; however, it slows down data input, with UPDATE and INSERT statements.

Creating an index involves the CREATE INDEX statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order.

**When to Avoid index?**

Although indexes are intended to enhance a database's performance, there are times when they should be avoided.

* Indexes should not be used on small tables.
* Tables that have frequent, large batch update or insert operations.
* Indexes should not be used on columns that contain a high number of NULL
* Columns that are frequently manipulated should not be indexed

**Hash index**

A PostgreSQL Hash index can perform a faster lookup than a B-Tree index. However, the key downside of the Hash index is that its use is limited to equality operators that will perform matching operations.

🡪 Easier to maintain / only suitable for equality operations

*When to use Hash?*

speedy lookup while utilizing less space

**B-tree index**

The PostgreSQL B-tree index creates a self-balanced tree– in other words, it sorts itself. It will maintain its balance throughout operations such as insertions, deletions and searches. Using a B-tree index speeds up scan operations because it doesn’t have to scan pages or records sequentially in a linear fashion.

🡪 Requires more maintenance/ Suitable for sorted information operations

*When to use B-tree?*

* General purpose indexing (info that is frequently query)
* highly efficient for operations performed on large blocks of data

**CONCURRENTLY**

build index on existing table/ can be used in production to avoid downtime

**Slow queries (3 Ways to identify them)**

|  |  |
| --- | --- |
| Slow Query Log | postgresql.conf file allows to set query execution time as a trigger for logging queries log\_min\_duration\_statement determines the minimum time a query must run for in order to be logged. This can then be queried in the log file and the slow query can be easily identified |
| EXPLAIN Analyze | The EXPLAIN command breaks down an SQL query and estimates how long each step will take. This is an excellent way to identify columns which may need to be indexed • EXPLAIN gives an estimate, EXPLAIN ANALYZE actually executes the query |
| pg\_stat\_statements | The pgstat\_statements extension provides a view of all SQL statements executed in a database |

**PARTITIONING**

Partitioning allows a table, index, or index-organized table to be subdivided into smaller pieces, where each piece of such a database object is called a partition.

Benefits:

* They can improve performance dramatically
* Make easier to manage and archive data
* Seldom-Accessed data can be transferred to cheaper slower storage

Currently, PostgreSQL supports partitioning via table inheritance. Each partition must be created as a child table of a single parent table. The parent table itself is normally empty; it exists just to represent the entire data set.

**Types of partitions:**

**List partition**: The table is partitioned by explicitly listing which key values appear in each partition  (e.g., a global sales table divided into regional partitions.)

**Range partitioning**: The table is partitioned into "ranges" defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions (E.g.  table containing sales data that is divided into monthly partitions, key = country code)

**When to use partitioning ?**

if a lot of data is going to be written on a single table at some point, users need partitioning. Apart from data, there may be other factors users should consider, like update frequency of the data, use of data over a time period, how small a range data can be divided,

(\*) Indexes are used to speed the search of data within tables. Partitions provide segregation of the data creating sub-directories for each partition.

**TABLESPACE**

A TABLESPACE IS A DATABASE OBJECT REPRESENTING A DIRECTORY ON YOUR FILESYSTEM

(\*) found in PG\_DEFAULT TABLESPACE

* Tablespaces can be used to control where on disk a file lives
* They are an important part of the DB
* They should not be created in removable media( e.g USB) or temp filesystems (RAM)

**BACKUP AND RECOVERY**

**DBA's Role**

* increase the MTBF (Mean Time Between Failures), 100% Availability
* reduce the MTTR (Mean Time to Recover), downtime after failure
* reduce data loss

**Categories of failures (NUMIS)**

* **N**etwork Failure

Configure multiple databases and make use of load balancing

(Time out, high volumes of connections )

* **U**ser Errors
* **M**edia Failure

Prevented by proper config, warm standby on another disk/machine, archiving

(disk damage, sysment /db admin deleted files)

* **I**nstance Failure

Instance failure is particularly dangerous because databases often keep important information in RAM

(Shutdown or crash, power cut, rebooting server, corrupted db )

* **S**tatement Failure

The server rolls back the statement when failure is detected

(Invalid data, Logic Errors, insufficient privileges, Space management )

3 different types of databse backup in PgSQL :

1. sql\_dump is simple, flexible and effective
2. filesystem backups can be done but are difficult, error-prone and not generally

recommended

1. pg\_start\_backup leverages the WAL in conjunction with a filesystem backup to ensure a consistent snapshot is taken

**INSOLATION LEVELS**

**Locks and Mutexes** - Most database systems allow table-level locks and rowlevel locks

A lock prevents other sessions from accessing data until it is released

[Mutexes](http://en.wikipedia.org/wiki/Mutex) are *mutually exclusive locks* and are designed to prevent concurrent access to resources that in doing so may result in unsafe conditions.

**SQL Lock Types**

* Access Share: read-only queries
* Share Row Exclusive: prevents concurrent updates to table
* Access Exclusive: no other process is allowed access to the table

**TRANSACTION 🡪 Is a group of queries which are executed as a single atomic unit**

A transaction is a sequence of operations performed (using one or more SQL statements) on a database as a single logical unit of work. The effects of all the SQL statements in a transaction can be either all committed (applied to the database) or all rolled back (undone from the database). Transaction must be ACID

**ISOLATION**

Graphical user interface, text, application, email

Description automatically generatedIsolation determines how transaction integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system.

**4 levels of isolation** (A BIT SHIT TBH)

Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system

**Read Uncommitted**

Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transactions, thereby allowing dirty reads. At this level, transactions are not isolated from each other.

**Read Committed**

This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevents other transactions from reading, updating, or deleting it.

**Repeatable Read**

This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on referenced rows for update and delete actions. Since other transactions cannot read, update or delete these rows, consequently it avoids non-repeatable read.

**Serializable**

This is the highest isolation level. A serializable execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

**A transaction isolation level is defined by the following phenomena:**

**Dirty Read**:  Dirty read is a situation when a transaction reads data that has not yet been committed

**Nonrepeatable read**: occurs when a transaction reads the same row twice and gets a different value each time

**Phantom reads**: occurs when two same queries are executed, but the rows retrieved by the two, are different

**Serialization Anomaly:**When the result of successfully committing a group of concurrent transactions is different from all possible combinations of running those transactions one after the other.

**EVENTUAL CONSISTENCY**

ughh

**SCALING DB**

ughh