Postgres SQL Tips

**Postgres Simple Tutorial:**

*Ref:*<https://www.postgresqltutorial.com/>

**VACUUM:**

*Ref :*<https://www.postgresql.org/docs/9.5/sql-vacuum.html>

VACUUM reclaims storage occupied by dead tuples. In normal PostgreSQL operation, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present until a VACUUM is done. Therefore, it's necessary to do VACUUM periodically, especially on frequently-updated tables.

**E.g.:** *VACUUM FULL FREEZE VERBOSE table\_name;*

**Indexing:**

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system, so they should be used sensibly.

*Ref :*<https://www.postgresqltutorial.com/postgresql-indexes/postgresql-index-types/>

E.g.:

*SELECT \* FROM pg\_indexes WHERE schemaname = ‘schema\_name’ and tablename='table\_name'*

*DROP INDEX vision.index\_name*

*CREATE INDEX index\_name ON table\_name (column\_name);*

PostgreSQL has several index types: **B-tree, Hash, GiST, SP-GiST, GIN, and BRIN**. Each index type uses a different storage structure and algorithm to cope with different kinds of queries.

When you use the CREATE INDEX statement without specifying the index type, PostgreSQL uses **B-tree index type** **by default** because it is best fit the most common queries.

**Note:** PostgreSQL automatically creates indexes on primary keys and unique constraints, but not on the referencing side of foreign key relationships.

**Common Table Expression:**

Introduction to PostgreSQL common table expressions or CTEs. A common table expression is a temporary result set which you can reference within another SQL statement including SELECT, INSERT, UPDATE or DELETE. Common Table Expressions are temporary in the sense that they only exist during the execution of the query

*E.g.:*

*with cte\_name as(select column1, column2, count(column3) from table\_name group by column1, olumn2)select \* from cte\_name where count >1*

**DB Connection State Details:**

*select \* from pg\_stat\_activity where pid in (select pid*

*from pg\_locks l, pg\_stat\_all\_tables t where l.relation=t.relid and schemaname=schema\_name’ order by relation asc)*

*SELECT pg\_terminate\_backend(pid)*

*Ref:*<https://www.postgresql.org/docs/9.3/monitoring.html>

<https://www.postgresql.org/docs/9.3/monitoring-stats.html>

**Window Functions:**

*Ref*: <https://www.postgresql.org/docs/9.3/functions-window.html#FUNCTIONS-WINDOW-TABLE>

Window functions provide the ability to perform calculations across sets of rows that are related to the current query row.

Note that these functions must be invoked using window function syntax; that is an OVER clause is required.

| **Function** | **Return Type** | **Description** |
| --- | --- | --- |
| **row\_number()** | bigint | number of the current row within its partition, counting from 1 |
| **rank()** | bigint | rank of the current row with gaps; same as row\_number of its first peer |
| **dense\_rank()** | bigint | rank of the current row without gaps; this function counts peer groups |
| **percent\_rank()** | double precision | relative rank of the current row: (rank - 1) / (total rows - 1) |
| **cume\_dist()** | double precision | relative rank of the current row: (number of rows preceding or peer with current row) / (total rows) |
| **ntile(num\_buckets integer)** | integer | integer ranging from 1 to the argument value, dividing the partition as equally as possible |
| **lag(value anyelement**  **[, offset integer**  **[, default anyelement ]])** | same type as value | returns value evaluated at the row that is offset rows before the current row within the partition; if there is no such row, instead return default (which must be of the same type as value). Both offset and default are evaluated with respect to the current row. If omitted, offset defaults to 1 and default to null |
| **lead(value anyelement**  **[, offset integer**  **[, default anyelement ]])** | same type as value | returns value evaluated at the row that is offset rows after the current row within the partition; if there is no such row, instead return default (which must be of the same type as value). Both offset and default are evaluated with respect to the current row. If omitted, offset defaults to 1 and default to null |
| **first\_value(value any)** | same type as value | returns value evaluated at the row that is the first row of the window frame |
| **last\_value(value any)** | same type as value | returns value evaluated at the row that is the last row of the window frame |
| **nth\_value(value any, nth integer)** | same type as value | returns value evaluated at the row that is the nth row of the window frame (counting from 1); null if no such row |

**Concurrency Control:**

*Ref:*

1. <https://vladmihalcea.com/how-does-mvcc-multi-version-concurrency-control-work/>
2. <https://devcenter.heroku.com/articles/postgresql-concurrency>
3. <https://www.postgresql.org/docs/9.1/explicit-locking.html>

Optimistic Locking is when you check if the record was updated by someone else before you commit the transaction.

Pessimistic locking is when you take an exclusive lock so that no one else can start modifying the record.

In Concurrency Control theory, there are two ways you can deal with conflicts:

* 1. You can avoid them, by employing a pessimistic locking mechanism (e.g. Read/Write locks, Two-Phase Locking)
  2. You can allow conflicts to occur, but you need to detect them using an optimistic locking mechanism (e.g. logical clock, MVCC)

When the [ACID transaction properties](https://vladmihalcea.com/a-beginners-guide-to-acid-and-database-transactions/) were first defined, Serializability was assumed.

And to provide a Strict Serializable transaction outcome, the [2PL (Two-Phase Locking)](https://vladmihalcea.com/2pl-two-phase-locking/) mechanism was employed.

When using 2PL, every read requires a shared lock acquisition, while a write operation requires taking an exclusive lock.

* 1. a shared lock blocks Writers, but it allows other Readers to acquire the same shared lock
  2. an exclusive lock blocks both Readers and Writers concurring for the same lock

However, locking incurs contention, and contention affects scalability.

The Amdhal’s Law or the Universal Scalability Law demonstrate how contention can affect response Time speedup.

For this reason, database researchers have come up with a different Concurrency Control model which tries to reduce locking to a bare minimum so that:

1. Readers don’t block Writers
2. Writers don’t block Readers

**Explicit Locking:**

PostgreSQL provides various lock modes to control concurrent access to data in tables. These modes can be used for application-controlled locking in situations where MVCC does not give the desired behaviour.

Ref: <https://www.postgresql.org/docs/9.1/explicit-locking.html>

<https://vladmihalcea.com/how-do-postgresql-advisory-locks-work/>

**Transaction, ACID Property,** **CAP Theorem, Isolation Level:**

*Ref:*<https://vladmihalcea.com/a-beginners-guide-to-acid-and-database-transactions/>

**Transaction:**

Transactions are a fundamental concept of all database systems.

The essential point of a transaction is that it bundles multiple steps into a single, all-or-nothing operation.

The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.

In a relational database, every SQL statement must execute in the scope of a transaction. Without defining the transaction boundaries explicitly, the database is going to use an implicit transaction which is wraps around every individual statement.

The implicit transaction begins before the statement is executed and end (commit or rollback) after the statement is executed.  
The implicit transaction mode is commonly known as [auto commit](http://en.wikipedia.org/wiki/Autocommit).

**ACID Property:**

* Atomicity
* Consistency
* Isolation
* Durability

**Atomicity** takes individual operations and turns them into an all-or-nothing unit of work, succeeding if and only if all contained operations succeed.

**Consistency** means that constraints are enforced for every committed transaction. That implies that all Keys, Data types, Checks and Trigger are successful, and no constraint violation is triggered.

**Isolation** brings us the benefit of hiding uncommitted state changes from the outside world, as failing transactions shouldn’t ever corrupt the state of the system. Isolation is achieved through concurrency control using pessimistic or optimistic locking mechanisms.

A successful transaction must permanently change the state of a system, and before ending it, the state changes are recorded in a persisted [transaction log](https://vladmihalcea.com/how-does-a-relational-database-work/). If our system is suddenly affected by a system crash or a power outage, then all unfinished committed transactions may be replayed.

**Note:** From a database perspective, the atomicity is a fixed property, but everything else may be traded off for performance/scalability reasons.

**Isolation Level:**

Although some database management systems offer [**MVCC**](https://vladmihalcea.com/how-does-mvcc-multi-version-concurrency-control-work/), usually concurrency control is achieved through locking. But as we all know, locking increases the serializable portion of the executed code, affecting [parallelization](http://en.wikipedia.org/wiki/Amdahl%27s_law#Parallelization).

1. READ\_UNCOMMITTED
2. READ\_COMMITTED
3. REPEATABLE\_READ
4. SERIALIZABLE

A **dirty read** happens when a transaction is allowed to read uncommitted changes of some other running transaction.

A **non-repeatable read** manifest when consecutive reads yield different results due to a concurring transaction that has just updated the record we’re reading.

This is undesirable since we end up using stale data. This is prevented by holding a shared lock (read lock) on the read record for the whole duration of the current transaction.

A **phantom read** happens when a second transaction inserts a row that matches a previous select criteria of the first transaction. This is prevented using range locks

|  |  |  |  |
| --- | --- | --- | --- |
| **ISOLATION LEVEL** | **DIRTY READ** | **NON-REPEATABLE READ** | **PHANTOM READ** |
| READ\_UNCOMMITTED | allowed | allowed | allowed |
| READ\_COMMITTED | prevented | allowed | allowed |
| REPEATABLE\_READ | prevented | prevented | allowed |
| SERIALIZABLE | prevented | prevented | Prevented |

**Partitioning:**

Partitioning splits a table into multiple tables, and generally is done in a way that applications accessing the table don’t notice any difference, other than being faster to access the data that it needs. By splitting the table into multiple tables, the idea is to allow the execution of the queries to have to scan much smaller tables and indexes to find the data needed. Regardless of how efficient an index strategy is, scanning an index for a table that’s 50GB will always be much faster than an index that’s for a table at 500GB. This applies to table scans as well, because sometimes table scans are just unavoidable.

*Ref:*<https://severalnines.com/database-blog/how-take-advantage-new-partitioning-features-postgresql-11>

**Function and Procedure:**

A drawback of user-defined functions is that they cannot execute [transactions](https://www.postgresqltutorial.com/postgresql-transaction/). In other words, inside a function you cannot [open a new transaction](https://www.postgresqltutorial.com/postgresql-transaction/), even commit or rollback the current transaction

Unlike a user-defined function, a stored procedure does not have a return value. If you want to end a procedure earlier, you can use the RETURN statement with no expression.

In case you want to return a value from a stored procedure, you can use output parameters. The final values of the output parameters will be returned to the caller.

To call a stored procedure, you use the **CALL** statement as follows:

***CALL*** *stored\_procedure\_name(parameter\_list);*

*Ref:*<https://www.postgresqltutorial.com/postgresql-create-procedure/>

<https://kb.objectrocket.com/postgresql/postgres-stored-procedures-with-input-and-output-parameters-sql-766>

**Note:** Increase application performance because the **user-defined functions and stored procedure are pre-compiled** and stored in the PostgreSQL database server.

**Trigger:**

A PostgreSQL trigger is a function invoked automatically whenever an event associated with a table occurs. An event could be any of the following: INSERT, UPDATE, DELETE or TRUNCATE.

A trigger is a special user-defined function associated with a table. To create a new trigger, you must define a trigger function first, and then bind this trigger function to a table. The difference between a trigger and a user-defined function is that a trigger is automatically invoked when an event occurs

*Ref:*<https://www.postgresqltutorial.com/postgresql-triggers/>