Concurrency Architecture

https://dl.acm.org/doi/10.1145/356842.356846

https://dspace.mit.edu/bitstream/handle/1721.1/16279/05331643-MIT.pdf

The main concurrency architecture for postgres is called Multiversion Concurrency Control, herein referred to as **MVCC**. MVCC solved many of the issues of concurrent access to a database. In addition to this postgres, also permits finer grained concurrency control mechanisms in order to get the desired results. The MVCC architecture is more of a procedural architecture, than a structural one, and focuses more on a high level how concurrent transactions are handled.

MVCC, Multiversion Concurrency Control

In general for concurrent systems that read and write a shared element, there needs to be some mechanism in order to prevent race conditions and deadlocks. MVCC accomplishes this using something called a snapshot. A snapshot is the state of the database at a particular point in time. When any transaction is sent to the database (when a transaction begins) it receives a time stamped snapshot of the database. This means that multiple concurrent transactions will see different versions of the database depending when the transaction began. This makes it so that at a particular time stamp when the transaction began, it is possible to get consistent data regardless of other concurrent transactions occurring. Additionally, MVCC does not require locks for reads or writes since snap shots are used. This provides considerable performance advantages on traditional locking only style concurrent architecture.

MVCC: Components.

MVCC requires some component that creates a snapshot when the transaction starts, another component to validate the requests from the transaction to ensure they are running within the parameters specified, once this validation is complete there needs to be some mechanism that commits these changes to the database. If postgres does not have mechanisms to emulate these components then MVCC would not be possible. These described components are purely conceptual to illustrate the mechanism in postgres required for MVCC, they may or may not exist as some concrete structure within the implementation. These components will be referred to in the following way as well as their purpose:

- **Snapshot**: The mechanism in postgres in which the database snapshot is taken when a transaction begins

- **Validator:** The mechanism in postgres that reviews the state of the requirements and database to ensure that transaction occurs within some defined parameters. It checks parameters set such as isolation levels or lock parameters, and initiates the procedure required to satisfy them.
- **Commiter:** The mechanism that writes the transaction to the actual database.

The following figure illustrates MVCC using the aforementioned architectural components:

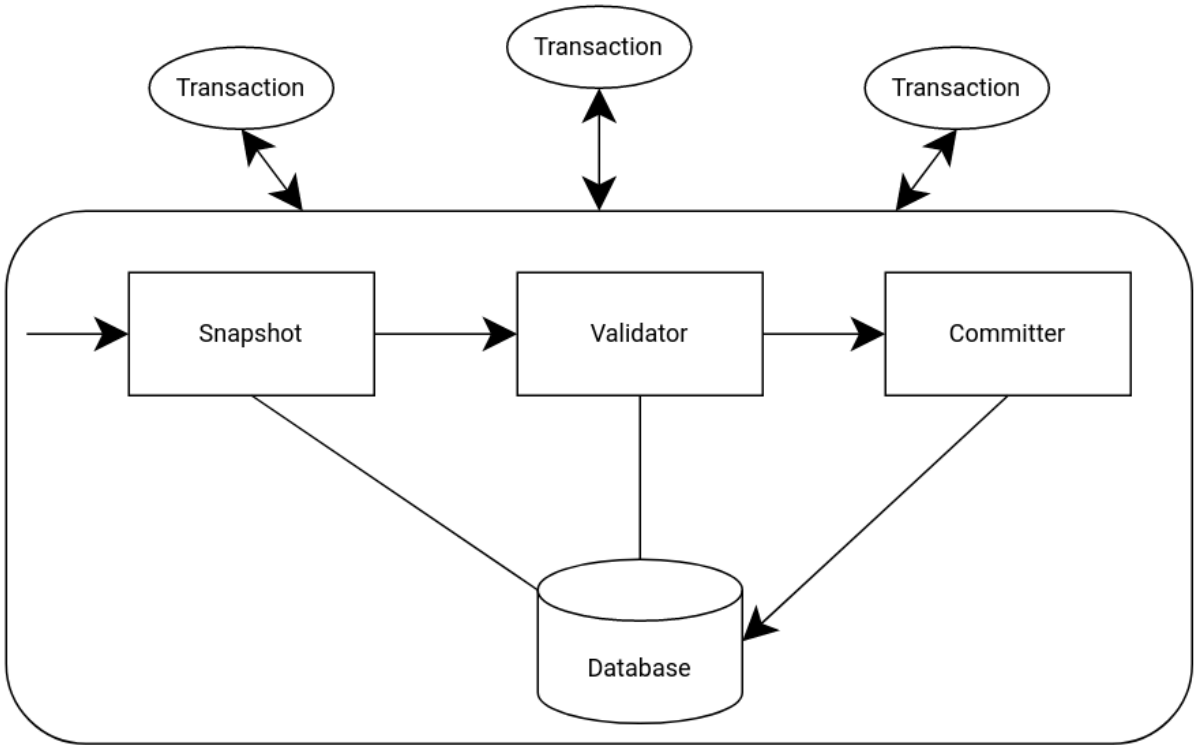


Figure.???. The overall architecture of MVCC. Derived by examining the general procedure of MVCC for any arbitrary transaction.

To better illustrate the flow of events and how these components interact, sequence diagrams will be used. The next section examines isolation levels for MVCC, and sequence diagrams will be used there to illustrate examples of concurrency phenomena, as these are more complex cases.

Isolation Levels

Using an MVCC architecture there are several phenomena that can occur when two concurrent transactions are attempting to access the same element. Defining an Isolation level allows for finer grained control of what should occur given these cases. The different phenomena that can occur are as follows:

Dirty read:

Occurs when one uncommitted transaction, lets call it ‘T1’ makes a write to an element, and another transaction, lets call it ‘T2’ reads this element some time after T1 completed the write, but before T1 commits. T2 would read the new T1 updated element before T1 commits. The following figure illustrates a dirty read using the previously mentioned architectural components.

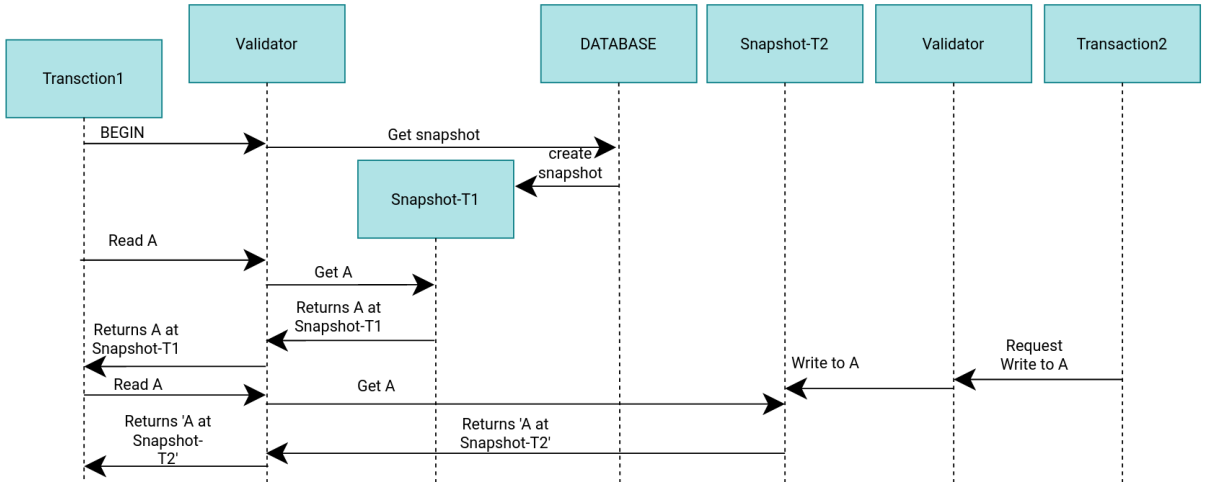


Figure ??? note that the Transaction2 is a concurrent transaction and how it began and its intermediate steps are omitted. This example is only concerned with Transaction2 writing to ‘A’. Transaction1 illustrates more details as it is the focus.

Nonrepeatable read:

An initial transaction, lets call it ‘T1’ reads an element, then at some point before it commits a separate transaction lets call it ‘T2’ commits a write to that element. T1 performs a read again on that same element, before it commits, it will get the newly committed change done by T2. Thus T1 would see two different values for the same element when it reads at different times:

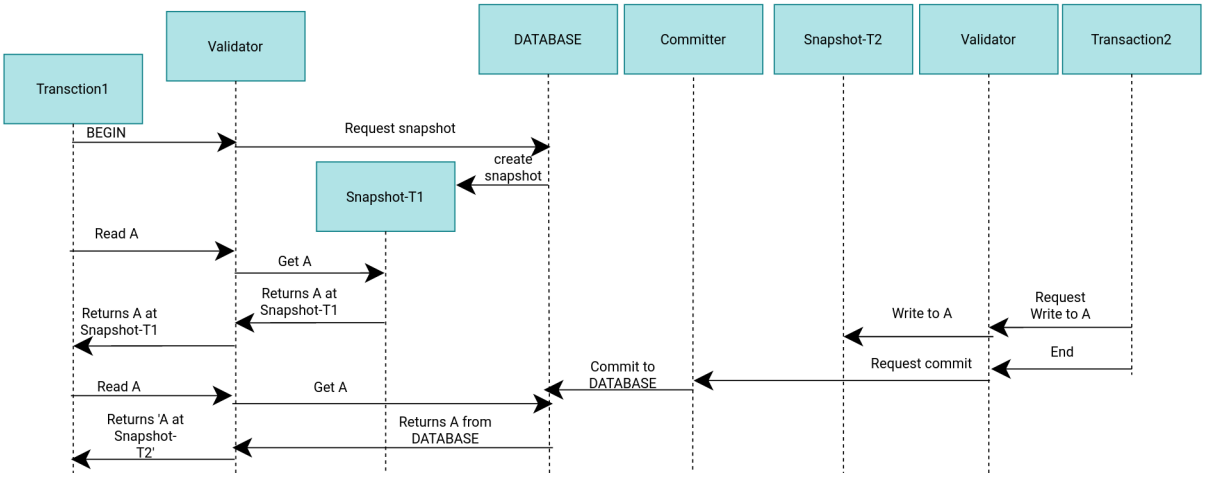


Figure ??? Transaction2 is a concurrent transaction and how it began and its intermediate steps are omitted. This example is only concerned with Transaction2 writing to ‘A’ and then commiting. Transaction1 illustrates more details as it is the focus.

Phantom read:

An initial transaction, lets call it ‘T1’ reads multiple rows based on some condition lets call it ‘C1’. Another transaction ‘T2’ commits changes to the database that effect the set membership of T1, before T1 commits. T1 reads that the set membership based on C1 has been changed because of the changes committed by T2. It is similar to non repeatable read, but applies to an aggregate transaction.

Serialization Anomaly:

The order of the successfully committed transactions has an effect on the database state. Different orders of committed transactions will lead to different database states.

Postgres supported isolation levels:

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

Table??, taken from SOURCE?? PG stands for postgres. Note that Read uncommitted is functionally the same as Read committed for post gres. Postgres does not completely implement the SQL spec in this regard. Additionally Serializable isolation level, basically mimics the functionality of 2 phase locking, so many of the MVCC benefits would not be witnessed in this level.

Using these defined isolation levels the MVCC architecture, will behave differently

Even Finer grained control: Locks

Sometimes the MVCC solution and its isolation levels may not exactly match the business requirements. Postgres allows a variety of different locks that can help achieve the

concurrent behaviour that is desired. Although it is not useful to completely understand the functionality of all the types of locks for the conceptual architecture, it is useful to know that this is a lower level control mechanism.

[SNAPSHOT DIAGRAM OVERALL]
Shows multiple concurrent transactions, and how they interact with each other.

Concurrency Phenomena

When concurrent transactions occur on the same element, several phenomenon can occur.

Postgres can control which phenomena can occur via setting an isolation level:
Read committed:
Serial:

Although the sql specification states these phenomenon, postgres only supports the following:

ACID Compliance

Atomicity
Consistency
Isolation
Durability

Generally for concurrent architectures locks are required when writing data. A writer that obtains a lock will not allow other processes to access it. When a more complex system like a database, which can have a large number of transactions with reads and writes, traditional locks, will cause a back log of transactions, as each batch of transactions that need to write to some element are scheduled waiting for each other to release the lock, one by one. This leads to much slower performance in general. The MVCC architecture was meant to be

more generalized, and its main selling point is that it does not require any locks. MVCC is centered on the concept of snapshots. Applied to a database, this meands that reads will never block writes and writes will never block reads.
Its main advantage is that it

In concurrent systems there can be multiple simultaneous transactions containing reads and writes, many phenomena can occur when reading and writing to the same element.
use snapshots at specific times in order to handle its concurrency. This means that read and writes do not lock each other up, which allows for performance gains. Each time a transaction starts, it can refer to a snapshot of the current database.
13.1. Introduction, pg 486

MVCC DIAGRAM (draw.io)

With this concurrency control system, postgres allows you multiple different ways to finely tune what occur within the MVCC upon a read and write.

STATE DIAGRAM EXAMPLE

Isolation Levels
Controls how concurrent reads and writes are processed

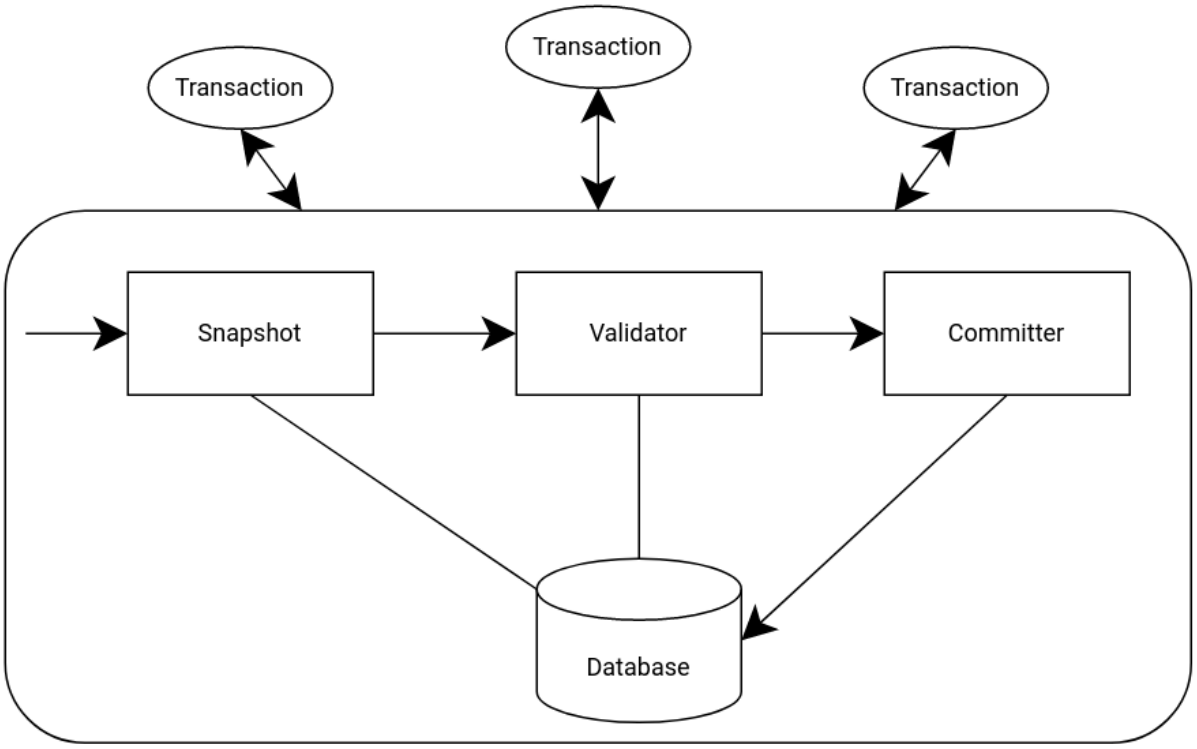
Concurrent Read and Write Phenomenon
When there are multiple concurrent transactions that can read and write there are a few phenomenon that can occur:
Dirty read: A transaction reads data written by a concurrent uncommitted transaction.
Nonrepeatable read: A transaction re-reads data it has previously read and finds that data has been modified by another transaction.
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-commited transaction.
Serialization Anomaly: Ther

3 distinct isolation levels:
Read Uncommitted, Read Committed
Repeatable Read

Serializable

A simple solution is to use a locking mechanism, to control, when process can read or write to an element. This locking mechanism functionally blocks access to some element under some conditions. A popular solution for databases is Two-Phase Locking which uses two types of locks. there are write-locks (Exclusive locks), in which any transaction that would write to an element in a database would obtain the lock, blocking all access to that element. The other type of lock is a read-lock (shared lock), a transaction that would read an element in the database would acquire this lock, and any other transactions that read the element would be able to access it, but all write transactions would be locked out. The main disadvantage of this solution is that in a concurrent database system, a large number of transactions can occur simultaneously and having transactions wait for lock access will cause delay and thus performance issues, especially for more complex longer running transactions.

These architectural components are purely conceptual used to more generally define, MVCC architecture. They may or may not exist as some class or process within postgres. They are named in a way that makes it clearer to understand what it does with respect to the MVCC architecture. They were derived by reviewing the functional components necessary in order to process an entire transaction and commit it. Firstly there must exist a mechanism for getting the snapshot of the database. There must be some mechanism in that watches other commits occurring during a transaction, or else, read committed cannot occur. There must be some validation mechanism in order to ensure that the transaction is valid before it is committed. And lastly in order to fully process the transaction there must be some mechanism to Commit these changes to the database.
MONITOR: monitors other commits during a transaction, represents the conceptual component that check concurrent commits within a transaction
VALIDATOR: Represents the conceptual component, that checks if a transaction, is valid. If not performs some action
COMMIT: commits to database


Transaction
Transaction
Transaction
Snapshot
Validator
Committer
Database