Isolation Levels and MVCC in SQL Databases: A Technical Comparative Study

Franck Pachot, Developer Advocate

yugabyteDB

@FranckPachot

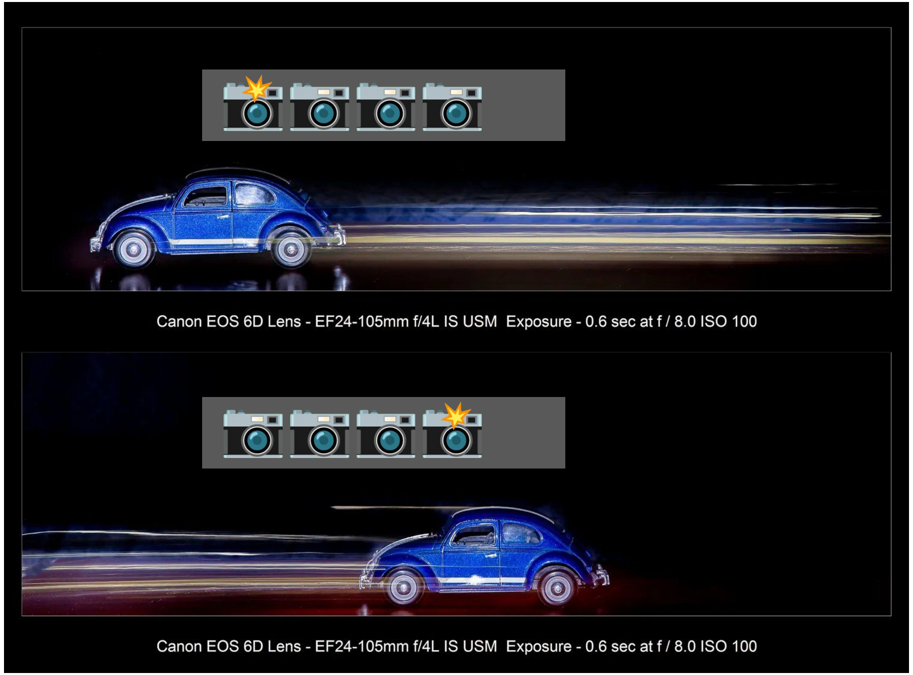
Question: Is this car moving forward or backwards?

Answer:

it is not moving
that's a picture (not a movie)

But the snapshot was taken
while the car was moving

Forward/Backward depends
on how the snapshot is taken



Canon EOS 6D Lens - EF24-105mm f/4L IS USM Exposure - 0.6 sec at f / 8.0 ISO 100

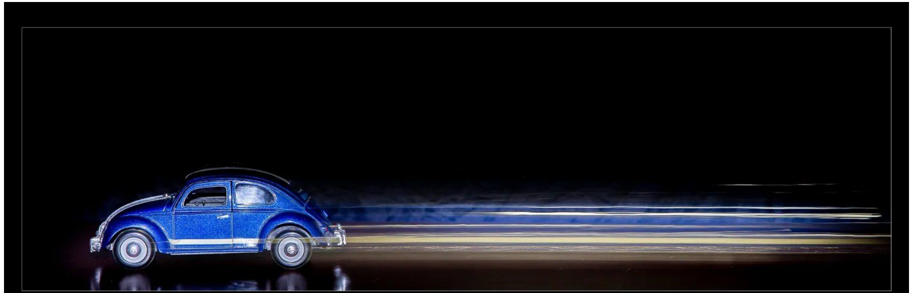
Canon EOS 6D Lens - EF24-105mm f/4L IS USM Exposure - 0.6 sec at f / 8.0 ISO 100

Question: How do we solve this anomaly?

Answer:

⏹️ Stop the car

⏸️ Take a movie



Canon EOS 6D Lens - EF24-105mm f/4L IS USM Exposure - 0.6 sec at f / 8.0 ISO 100

Canon EOS 6D Lens - EF24-105mm f/4L IS USM Exposure - 0.6 sec at f / 8.0 ISO 100

Databases are moving (others are writing to it)

To read on a consistent state

⏹️ Stop the car

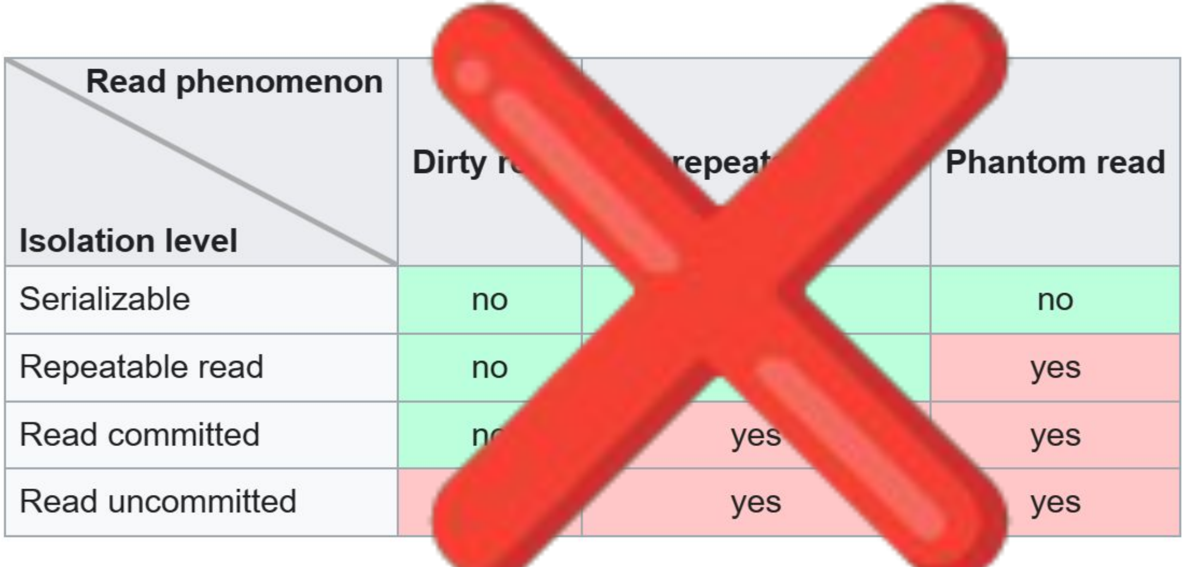
⏹️⏺️ Lock what intend to read

⏸️ Take a movie

⏪⏸️ Read a previous snapshot

Forget what you have learned (Isolation Levels & Phenomenon)

Serializable SI
Read-Only SI
Snapshot Isolation (SI)
Read Committed SI



Read phenomenon / Isolation level	Dirty r	repeat	Phantom read
Serializable	no	[illegible]	no
Repeatable read	no	[illegible]	yes
Read committed	n	yes	yes
Read uncommitted	[illegible]	yes	yes

This was defined in SQL-92 but is not how databases work!

Modern databases use MVCC snapshots and explicit locks

SQL ANSI vs. Real Life

SQL Standard:

- 🍿Users and application developers do not lock the rows explicitly
- 📽️The DB locks rows implicitly that are read to prevent anomalies

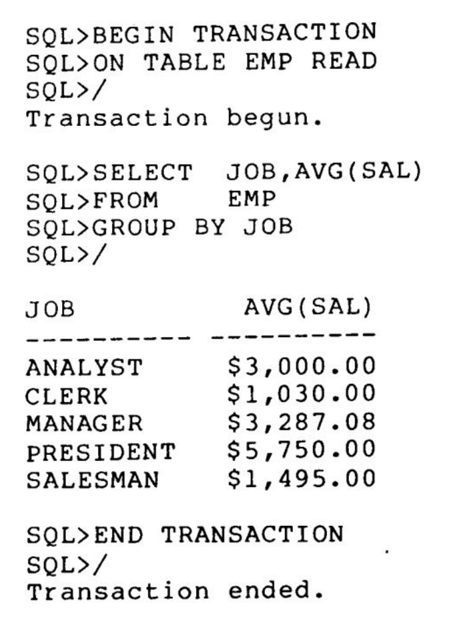
SQL Databases:

- ⏪⏸️Read from a past snapshot to avoid blocking reads
- ⏹️⏺️Developers declare their lock intent (SELECT FOR UPDATE) when the state they read must be frozen until the end of transaction

Some Concepts to understand Isolation Levels in modern DBs:

- SQL transactions and ACID
- Read and Write time during a transaction
- Optimistic and Pessimistic locking
- Explicit locking in SQL

SQL transactions are complex



```
SQL>BEGIN TRANSACTION
SQL>ON TABLE EMP READ
SQL>/
Transaction begun.

SQL>SELECT  JOB,AVG(SAL)
SQL>FROM    EMP
SQL>GROUP BY JOB
SQL>/

JOB              AVG(SAL)
----------  ----------
ANALYST       $3,000.00
CLERK         $1,030.00
MANAGER       $3,287.08
PRESIDENT     $5,750.00
SALESMAN      $1,495.00

SQL>END TRANSACTION
SQL>/
Transaction ended.
```

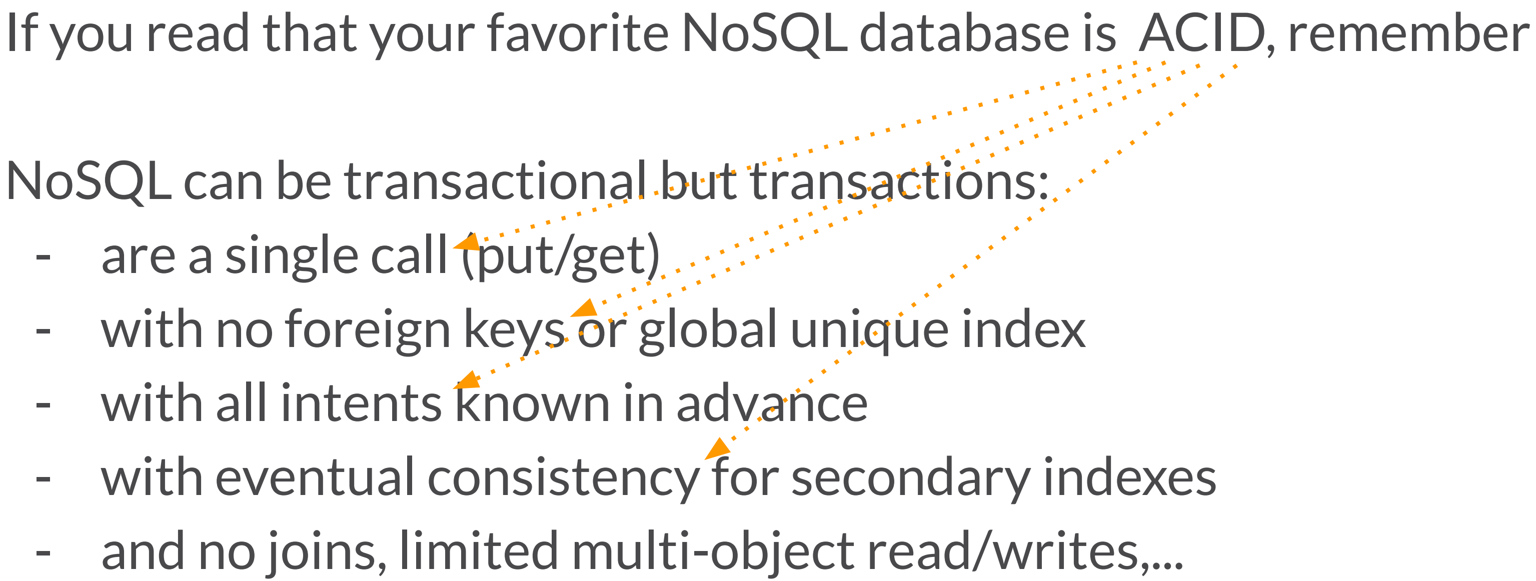
SQL Transactions are complex

- do multiple reads and writes,
  and writes depends on what was read
  example: book a seat that is free
- do not declare what they do before doing it

Even a single row insert in a SQL table is a complex transaction:

- check foreign keys
- update secondary indexes
- raise error if key already exists

SQL transactions are more complex than **NoSQL** transactions



If you read that your favorite NoSQL database is ACID, remember

NoSQL can be transactional but transactions:

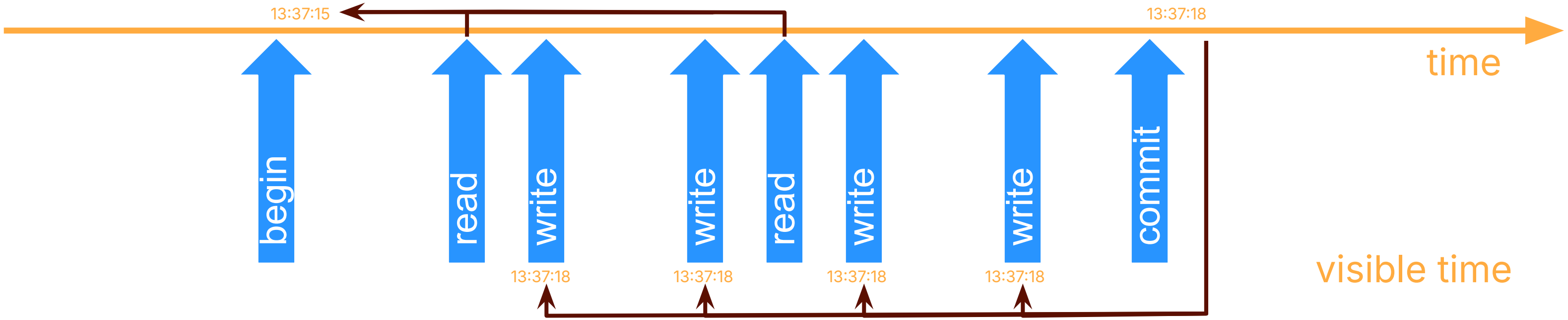
- are a single call (put/get)
- with no foreign keys or global unique index
- with all intents known in advance
- with eventual consistency for secondary indexes
- and no joins, limited multi-object read/writes,...

Read and Write time cannot be instantaneous

- Read and writes cannot happen simultaneously (can't be atomic)
- We can read from the past, not from the future

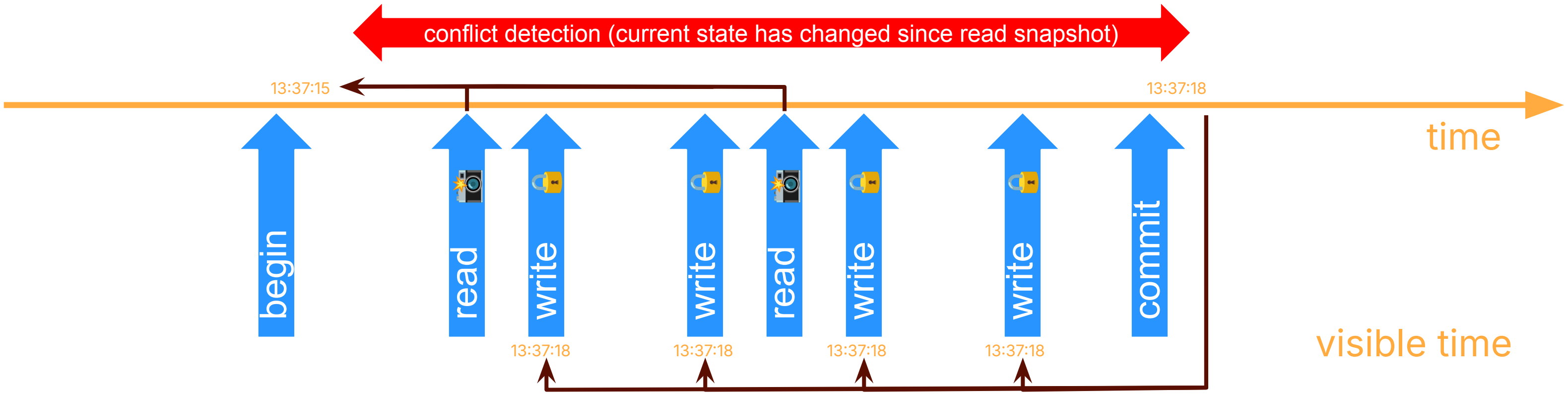
  except if there's no modifications, then past=future
- We cannot write to the past

  except if there's no modifications, then present=future



Read and Write time in MVCC databases

- Write time is the commit time: exclusive lock
- Read time is from a past snapshot
- Writes or Commit check if the two states conflict because of concurrent transactions committed in between



Optimistic and Pessimistic locking

Let's name them from their behavior:

- **Wait-on-Conflict** (enqueue)

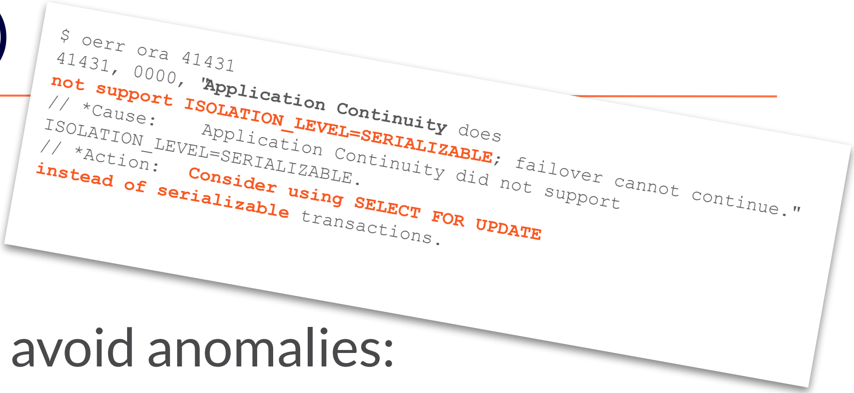
When a cause of conflict is detected, we wait for the conflicting transaction to end (commit or rollback) and continue

- **Fail-on-Conflict** (kill or die)

When a conflicting situation is detected, raise an error and retry
In some cases, the database can retry automatically, in some others the application must have a retry logic
Note: Optimistic Concurrency Control (OCC) is Fail-at-Conflict delayed to Commit time

- **Skip-on-Conflict** (ignore)

Explicit locking (by the application, in SQL)



```
$ oerr ora 41431
41431, 0000, "Application Continuity does
not support ISOLATION_LEVEL=SERIALIZABLE; failover cannot continue."
// *Cause:    Application Continuity did not support
ISOLATION_LEVEL=SERIALIZABLE.
// *Action:   Consider using SELECT FOR UPDATE
instead of serializable transactions.
```

This is ignored by the SQL Standard

But that's what most applications do to avoid anomalies:

```
SELECT ... FOR UPDATE
                          WAIT          -- Wait-on-Conflict
                          NOWAIT        -- Fail-on-Conflict
                          SKIP LOCKED   -- Skip-on-Conflict
LOCK TABLE ... SHARE/EXCLUSIVE... -- for serializable
```

Use this and ignore Isolation Levels 😁

Default Isolation Level and Explicit Locking is OK

but only if you understand it, and all databases behave differently

That's the reason for this presentation:
Isolation Levels and MVCC in SQL Databases: A Technical Comparative Study

👆

MVCC = Multi-Version Concurrency Control
aka Multi-Version Read Consistency
aka Multi Generational Architecture

What IBM said in 2002 about Multi-Version Read Consistency

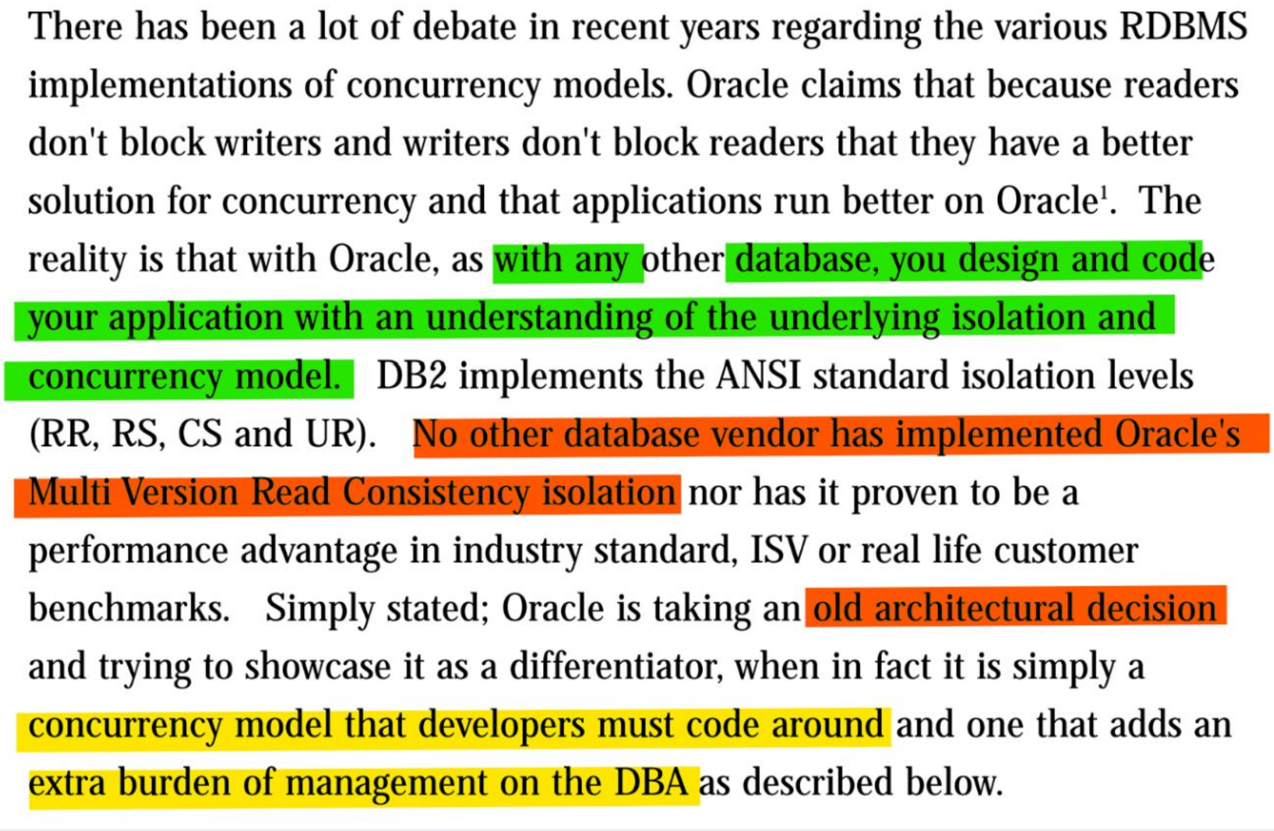
20 years later:

🟢 all DBs have different behavior on race condition (MVCC implementations)

🔴 all databases provide MVCC isolation levels

🟡 apps prefer explicit locking to isolation levels (SELECT FOR UPDATE)

🟡 TPC-C was built so that it doesn't require Serializable because Oracle didn't have it



There has been a lot of debate in recent years regarding the various RDBMS implementations of concurrency models. Oracle claims that because readers don't block writers and writers don't block readers that they have a better solution for concurrency and that applications run better on Oracle[1]. The reality is that with Oracle, as with any other database, you design and code your application with an understanding of the underlying isolation and concurrency model. DB2 implements the ANSI standard isolation levels (RR, RS, CS and UR). No other database vendor has implemented Oracle's Multi Version Read Consistency isolation nor has it proven to be a performance advantage in industry standard, ISV or real life customer benchmarks. Simply stated; Oracle is taking an old architectural decision and trying to showcase it as a differentiator, when in fact it is simply a concurrency model that developers must code around and one that adds an extra burden of management on the DBA as described below.

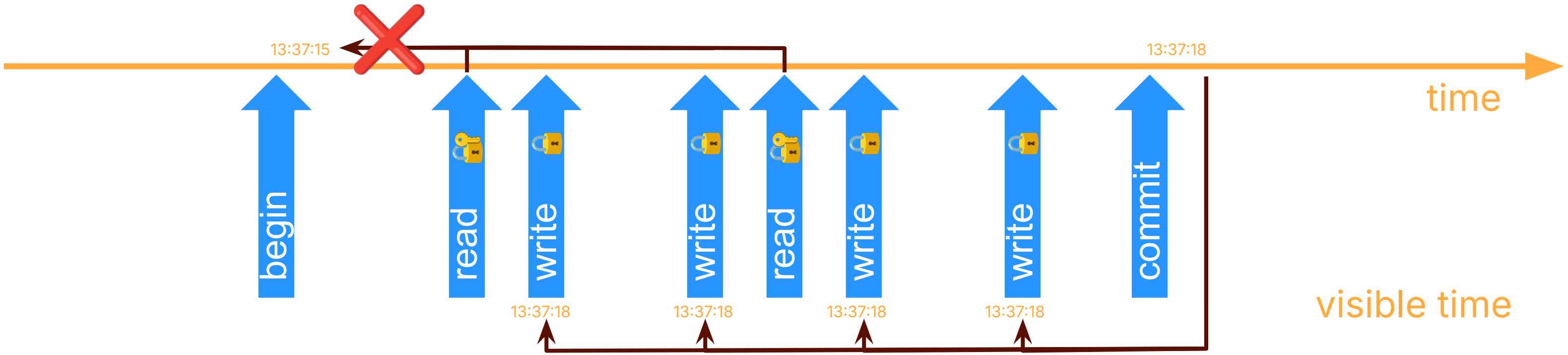
Non-MVCC databases

The cannot read as-of one state

They need to lock what they read to guarantee the same state

The Isolation Level is the duration of this lock

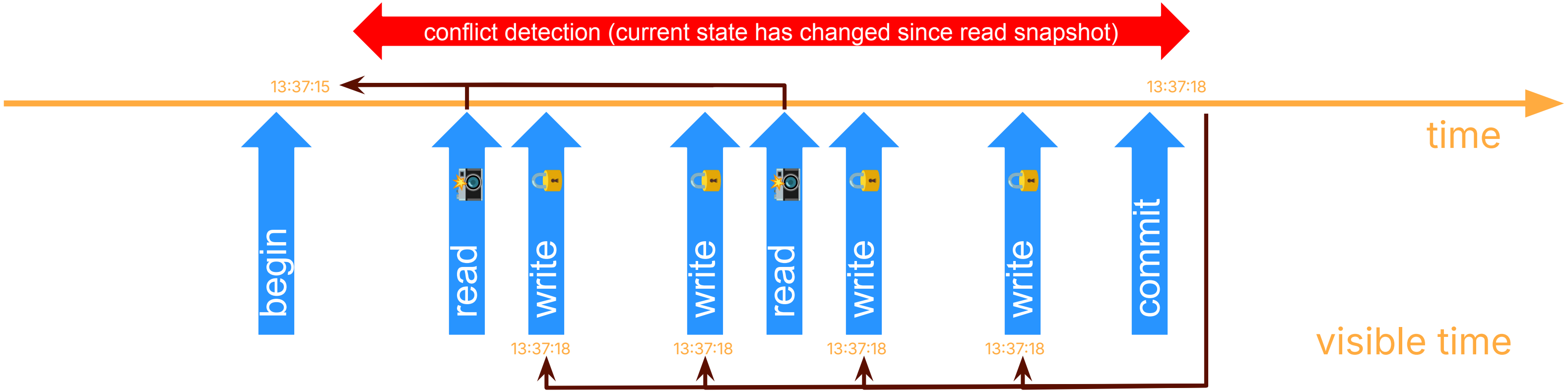
DB2, SQL Server (<2005), ...



MVCC read doesn't lock

- need to read from the past
- need to detect conflicts with the writes
- but 🎊 readers don't block writers (reports, dumps, read replicas)

Writes are the same (lock current state). MVCC is about reads



MVCC: Multi-Version Concurrency Control

Protocol defined in MIT 1978. First implementations at DEC as a solution to deadlocks.
Then VAX Rdb/ELN and InterBase with tuple versioning.

Changes are versioned (table rows, index entries, file blocks (pages)
Transactions can read AS-OF the start of transaction (start of statement)
Optimistic locking for reads (no locks), Pessimistic locking for writes
Great for mixed workloads (analytic/reporting on operational database)

Doesn't affect writes (they still have to lock) except for conflict detection

MVCC: implementation choices

How to version: timesamp (monotonic clock), sequence# (wraparound)
What to version: pages? rows? index entries?
Where to store past versions: in-place? undo log?
Which direction to scan: oldest to newest, newest to oldest
When to garbage collect: re-use, vacuum, compaction

Secondary indexes: must index all values, pointer to PK or TID

All databases are different: MVCC implementation

MVCC	**what is versioned**	**where is the past version**	**where is the current version**	**the storage is organized by**	**delayed commit and garbage collection**
PostgreSQL	**table rows** (heap)	**same place**, with a pointer to new one	**appended to heap** table (or maybe in same block if fillfactor <100 to avoid updating all indexes)	by **key in index** with all versions together (until vacuum) versions are **scattered in heap**	hint bits vacuum
Oracle	**table** or **index blocks**	**undo vectors** applied to current block, pointers to old ones	**in-place** with pointer to undo records (table/index row/entry lock flag, block ITL, transaction table, tx undo records)	b-tree indexes and heap table for current block. Past versions: per **transaction**, undo vectors both protected by redo logs (WAL)	delayed cleanout rollback expiration (but **ORA-1555**)
SQL Server	**table rows**	**tempdb**, now in Persistent Version Store (for ADR)	**in-place** with a pointer to old tables	by **key** in clustered and secondary index Past versions: new to old	ghost cleanup
MySQL InnoDB	**table rows** (PK)	**two logs** types: inserts and one for update/delete	**in-place** with pointer to transaction table and then to update log	by **key** indexes + delete marker, primary index points to transaction + delete marker, rollback segment	purge the delete markers
YugabyteDB	**table** or **index rows**/entries in LSM-Tree	**next to the curren**t: rocksdb key is pk/index + timestmp	new subdoc **in-place** (new packed row or new column value) in IntentsDB, moved to RegularDB after commit	by **key** for secondary indexes and table (primary key) and versions. In intents/regular Memtable + SST Files	deletion of provisional **records in IntentsDB** once in RegularDB SST compaction

yugabyteDB

Some Pros and Cons

Keep old versions in place

➖ bloat, need to vacuum, versions scattered in heap

➕ fast rollback (= fast recovery, = fast failover)

MVCC in heap table only

➖ heap fetches for Index Only Scan, need to rebuild indexes to free space

➕ simplicity (easy to add new index types)

Stop garbage collection when long transactions

➖ long transactions block vacuum

➕ readers do not fail with "snapshot too old" (on primary)

Store per key, per transaction... Chain versions from old to new or ...

Default isolation levels

Read Committed (PostgreSQL, YugabyteDB, Oracle, SQL Server)

- ✅ with statement read restart: Oracle, YugabyteDB
- ⚠️ different read times in stored procedure: Oracle
- ⚠️ with row re-read: PostgreSQL

Repeatable Read (MySQL)

- In MVCC databases, RR is Snapshot Isolation
- ⚠️ MySQL can show DELETE or UPDATE that are not isolated

Serializable (CockroachDB, Spanner)

- Need to implement Read Committed to be compatible with existing applications

Read Restart

Read Committed is not exempt from serializable errors

but the database can have its own retry logic

Read Restart is possible in Read Committed

because the spec allows per-statement read-time

Require savepoints before each statements

possible in Repeatable Read if first statement (can restart it)

Only if nothing has been returned to the application

cannot rollback if the application did something non-transactional (file, e-mail, queue)

Be careful

PostgreSQL Read Committed

- may read some rows at a different point in time (example)

MySQL Repeatable Read

- may see other's commits (doc)

Oracle Serializable is not serializable

- In the old times: non-default "_serializable"=true wich locked the tables

SQL Server escalates locks (UPDLOCK)

- prefer non-MVCC isolation levels

Long transaction blocking garbage collection, or snapshot too old

Read Restart may transform dirty writes into lost updates

Don't Panic, the solution is often to SELECT FOR UPDATE

SQL Isolation Levels vs. Implicit Locking

When SQL standard defined isolation levels (SQL-92)

- user interactions, long transactions, not declaring all intents beforehand

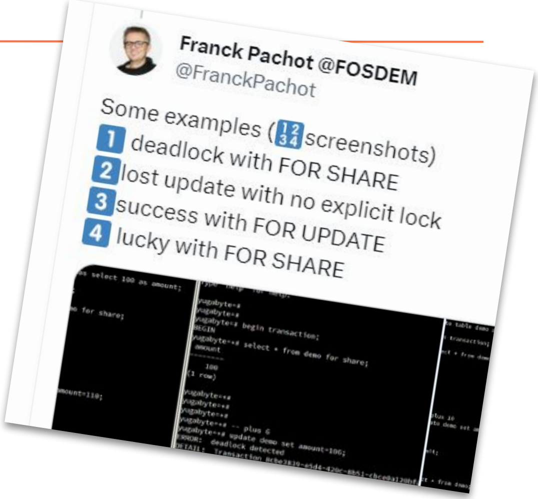
Modern SQL usage

- lot of single-statement, auto-commit
- transactions do not span multiple user interactions
- applications know the intent of a transaction
- not all databases have serializable, and when they do it may hurt performance

How to use Isolation Levels in MVCC databases - in short

Isolation Level	What the DB does	What your code can do
Read Committed	No read locks 🕰️ read-time = statement	Avoid non-repeatable reads with SELECT FOR SHARE/UPDATE but 👻 phantom reads and write skew possible
Repeatable Read = Snapshot Isolation	No read locks 🕰️ read-time = transaction	Retry logic for error 40001 Avoid write skew with: - Lock table 🥶 - SELECT FOR UPDATE on parent key - Index on foreign keys in Oracle
Serializable	Read locks (range or predicate) 🕰️ read-time = transaction	Add a retry logic and code like you are alone on the database 😎 Serializable read only doesn't need locks

Explicit locking



Franck Pachot @FOSDEM
@FranckPachot

Some examples (🔢screenshots)
1️⃣ deadlock with FOR SHARE
2️⃣ lost update with no explicit lock
3️⃣ success with FOR UPDATE
4️⃣ lucky with FOR SHARE

```
as select 100 as amount;

o for share;

amount=110;
```

```
yugabyte=#
yugabyte=#
yugabyte=# begin transaction;
BEGIN
yugabyte=*# select * from demo for share;
 amount
--------
    100
(1 row)

yugabyte=*#
yugabyte=*#
yugabyte=*#
yugabyte=*# -- plus 6
yugabyte=*# update demo set amount=106;
ERROR:  deadlock detected
DETAIL:  Transaction 8cbe3839-e5d4-420c-8851-cbce0a120bf
```

SELECT FOR SHARE

⚠️ sufficient to prevent lost updates but may deadlock on later update

https://x.com/FranckPachot/status/1721292232030880072?s=20

🅾️Oracle never implemented row shared locks

SELECT FOR NO KEY UPDATE

guarantees the possibility to update the row columns later

SELECT FOR UPDATE on a parent row to avoid phantom reads on the childs

LOCK TABLE IN SHARE MODE for full serializability (but blocks DML)

✅ With the choice of WAIT, NOWAIT, SKIP LOCKED

Franck Pachot
Developer Advocate YugabyteDB,
AWS Hero, SQL Dev & DBA (OCM)

Isolation Levels in Modern SQL Databases

blog series on dev.to:
https://dev.to/franckpachot/series/25468

E-mail:
fpachot@yugabyte.com

Blogs:
dev.to/FranckPachot
blog.yugabyte.com/author/fpachot

Twitter:
@FranckPachot

Youtube:
youtube.pachot.net

LinkedIn:
www.linkedin.com/in/franckpachot

🚀Community Slack / Github:
www.yugabyte.com/community

yugabyteDB

@FranckPachot

Franck Pachot

Developer Advocate at Yugabyte

Past:

20+ years in databases, dev and ops, consulting
Oracle ACE Director, AWS Data Hero
Oracle Certified Master, AWS Database Specialty

fpachot@yugabyte.com
dev.to/FranckPachot
@FranckPachot

ORACLE ACE Director
OakTable
ORACLE Certified Master Oracle Database 12c Administrator
ORACLE Certified Expert
aws certified Solutions Architect ASSOCIATE
aws certified Database SPECIALTY
aws ACADEMY Educator
aws HEROES