4
Concurrency Protocols –
Part 3
Wrap-Up

Reading: [Kl], chapter 7; [Ha] chapter 9; [Me] chapter 4

KUTAISI INTERNATIONAL UNIVERSITY

MVCC - Writes

Operation	Description	Version	Value	Created_by Xmin	Deleted_by Xmax
Insert	creates a version, created_by: TS (Ti), deleted_by: 0	A0	x	TS (Ti)	0
Delete	Mark for deletion	A0	x	TS (Ti)	TS(Td)
Update	delete and create	A0	x	TS (Ti)	TS(Tu)
		A1	y	TS(Tu)	0

because we add up all these versions we need garbage collection

Garbage Collection – in PostgreSQL called Vacuum Process: Needs to run regularly in order to remove old versions and deleted objects (rows) and to free up space for re-use

we can't run vacuum in transaction that yet has not commited because it might roll back

```
/* returns number of dead tuples in table teacher */
SELECT * FROM pgstattuple('teacher'::regclass);
/* returns number of dead tuples from updates / deletions and statistice about HOT updates */
SELECT relname, n_tup_upd, n_tup_del, n_tup_hot_upd, n_tup_newpage_upd  FROM pg_stat_user_tables WHERE relname = 'teacher';
/* updates Postgres statistics */
Analyze teacher;
```

KUTAISI INTERNATIONAL UNIVERSITY

2PL Write-Write

T1	T2	
Begin		
Read(A)		s-locka(a), read(a)
Write(A)		x-lock(a), write(a)
	Begin	
	Read(A)	wait
	Write(A)	
Read(A)		read(a)
Commit		release of all locks
	Commit	

read(a)

s-locka(a), read(a)
x-lock(a), write(a)
release locks with commit

in 2PL while we have write operation in one transaction, neither read nor write can happen in second transaction, so in this case T2 waits for T1 to commit and only after that reads and writes.

Serializability achieved by locks and letting transactions wait for each other.

KUTAISI INTERNATIONAL UNIVERSITY

MVCC – Snapshot Isolation - Concurrent Writes

each transaction sees which other transaction has commited

T1	T2	
Begin		
Read(A)		
Write(A)		
	Begin	
	Read(A)	
	Write(A)	wait A1
Read(A)		
Commit		
	Commit	

Version	Value	Created_by	Deleted_by
A0	100	00	1
A1	50	1	~~0~~ 2
A2	0	2	0

read commited

Result of this schedule under snapshot isolation READ COMMITTED → lost update

Result of this schedule under snapshot isolation Repeatable READ? →T2 rolls back

in MVCC only write operations block write operations
so in case of READ COMMITED since T1 does not commit after write(A), T2 will read initial old value
and will wait for T1 to complete the transaction so that T1 can write and when T2 writes version A2 is created.

but in case of REPEATABLE READ both transactions work with the snapshot of table and when T2 reads A0 it knows that it isn't valid anymore but doesn't see the change since it's working with initial snapshot of table before any change occured so it rolls back. This means that we won't have version A2 like we had in the READ COMMITED case.
But if T1 rolls back instead of commit then T2 reads valid value and can do write operations and commit.

MVCC – Write – Write Conflict

T1	T2
Begin	
Read(A)	
Write(A)	
	Begin
	Read(A)
	Write(A)
Read(A)	
Commit	
	Commit

Version	Value	Created_by	Deleted_by
A0	**100**	**00**	**01**
A1	**50**	**01**	**∞**

in contrast to strict2PL

- MVCC protocol does NOT guarantee serializability.
- MVCC has rules for READ-READ and READ-WRITE and WRITE-READ transactions.
- WRITE-WRITE conflicts may end in lost update or are "solved" with serialization errors and rollbacks – depending on the isolation level.
- For implementation of serializability, e.g. to prevent write-skew, MVCC needs additional protocol rules.

Some Quotes out of PostgreSQL Documentation

"

The Repeatable Read mode provides a rigorous guarantee that each transaction sees a completely stable view of the database. However, this view **will not necessarily always be consistent with some serial (one at a time) execution of concurrent transactions** of the same level ....

The ***Serializable*** **isolation level** provides the strictest transaction isolation. This level **emulates serial transaction execution** for all committed transactions;
"

https://www.postgresql.org/docs/current/tranaction-iso.html

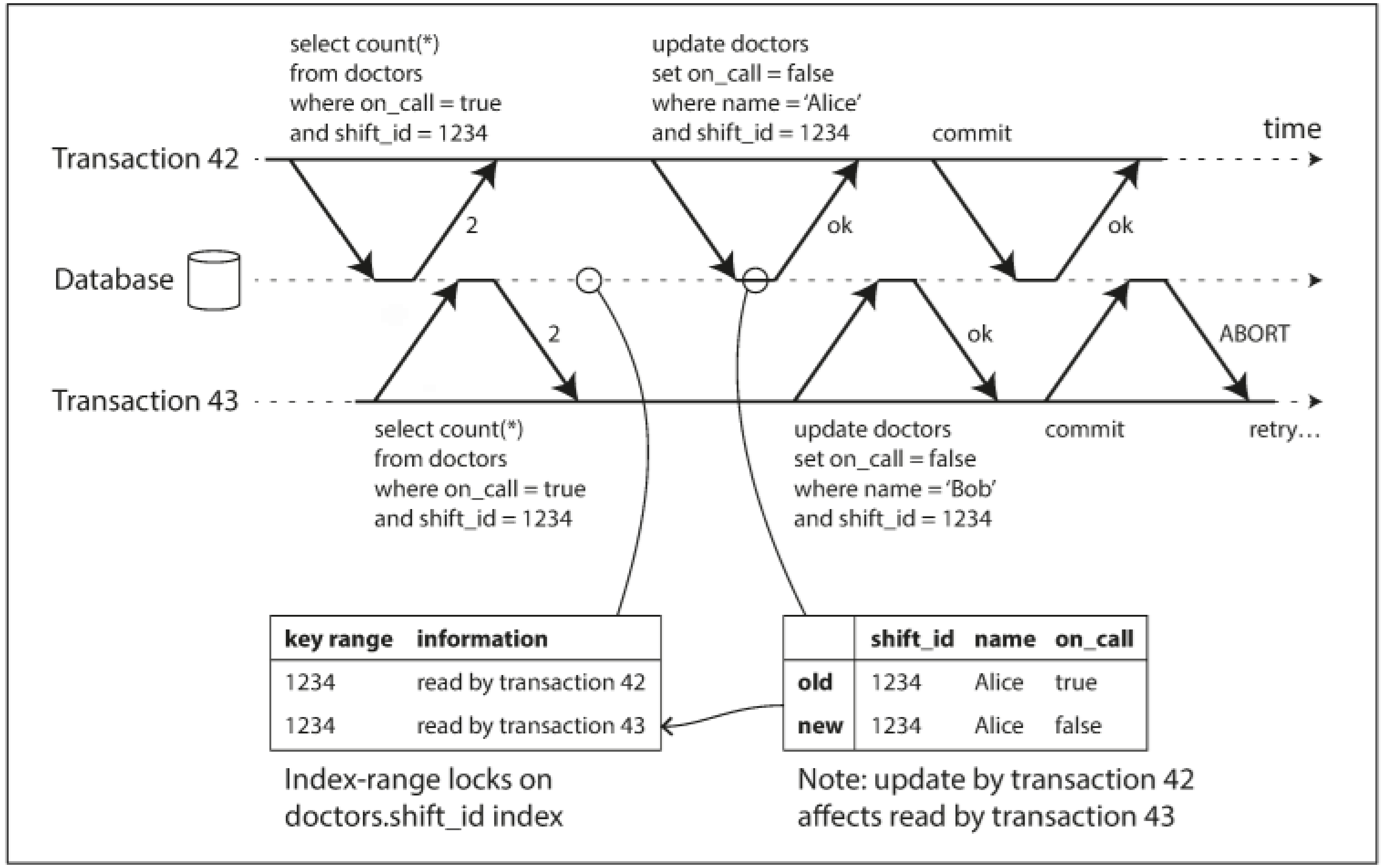
Write Skew: Anomaly Pattern RW Dependency
Ports/Grittner, SSI in Postgres, 2012 (uploaded in Teams)

SSI - Serializable Snapshot Isolation

- T1 reads some data from the database and writes – based on the result of the read.
- T2 reads some data from the database and writes – based on the result of the read.
- There is a causal dependency between the reads and the writes of the transaction.
- By the time T1 / T2 want to commit,
    - both transactions only see the read result caused by their own write.
    - but at the same time, the premise for the writes is no longer true.
- If both transactions commit, there is a violation of consistency.

- Preventing the anomaly:
- → **Detecting writes that affect prior reads**
  T2 (as second writer) needs to detect that the first writer T1 may have affected the result of its prior reads.

**Example:
Doctors on call:
At least 1 doctor
needs to be on duty;
Alice and Bob
are on duty**



key range	information
1234	read by transaction 42
1234	read by transaction 43

	shift_id	name	on_call
old	1234	Alice	true
new	1234	Alice	false

Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

[Kl], p.264ff

SSI Anomaly Detection Algorithm

- T1 Query: select count(*) from personnel where on_duty=TRUE
- Query in serialiazbale isolation level causes "predicate lock" on all rows that match search result.
- T2 Query also causes "predicate lock" on all rows that match search result.
- T1 updates, T2 updates
- T1 wants to commit
  → checks
  - whether any write happened on the predicate-locked rows that could affect its prior read result
  - whether the write is committed

  → there is a write (T2) that could affect its prios read result – but not committed yet.
  → T1 commits
- T2 wants to commit
  → checks whether
  - whether any write happened on the predicate-locked rows that could affect its prior read result
  - whether the write is committed

  → there is a write (T1) that could affect its prios read result – and it is committed.
  → T2 rolls back

sometimes postgre cancels a transaction just to be safe, even if there wasn't actually a real problem. This is false positive.

Explain, why under Postgres SSI "false positives" can happen

postgre watches how transactions interact to avoid conflicts. If it sees a pattern that might lead to a data inconsistency, it assumes the worst and cancels one of the transactions even if in reality nothing bad would've happened.

PostgreSQL Serializable Snapshot Isolation
Monitoring of read / write dependencies

- Isolation level serializable does NOT consider any data read from a table valid until the transaction which read it has successfully committed.
- PostgreSQL uses so called ***predicate locking* (*which is a monitoring mechanism*)**, which means that it keeps "locks" which allow it to determine when a write would have had an impact on the result of a previous read from a concurrent transaction, had it run first.
- In PostgreSQL these locks do not cause any blocking and therefore can *not* play any part in causing a deadlock. They are used to identify and **flag dependencies** among concurrent serializable transactions which in certain combinations can lead to serialization anomalies.
- These predicate locks often need to be kept past transaction commit, until overlapping read write transactions complete.
- If all transactions are run at the SERIALIZABLE transaction isolation level, business rules can be enforced in triggers or application code without ever having a need to acquire an explicit lock or to use SELECT FOR SHARE or SELECT FOR UPDATE.

https://www.postgresql.org/docs/10/transaction-iso.html#XACT-SERIALIZABLE

PostgreSQL Serializable Snapshot Isolation (SSI)

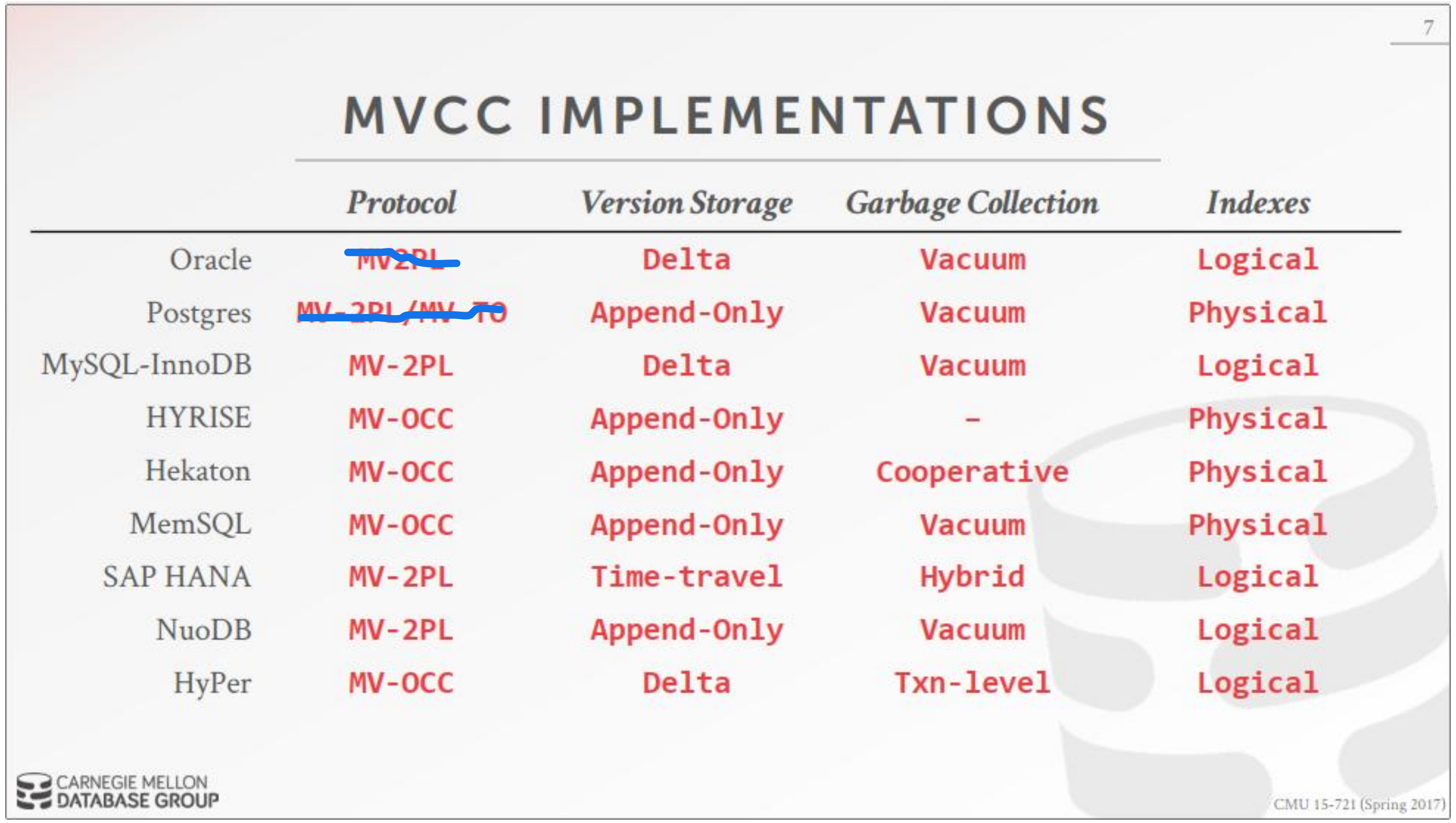
- PostgreSQL *Serializable* isolation level provides serializability, that is, the result is equivalent to a serial schedule. (Rolling back with an error also provides serializability.)
- It works like isolation level repeatable read but in addition monitors for conditions which could make the execution of a concurrent set of serializable transactions behave in a way that may be inconsistent with all possible serial executions of those transactions.
- The monitoring does not introduce any blocking beyond that present in repeatable read, but there is some overhead to the monitoring. Detection of conditions which could cause a *serialization anomaly* will trigger a *serialization failure with the error message:*

  ERROR: could not serialize access due to read/write dependencies among transactions

For comparison:
Repeatable Read error message for concurrent writes is:

ERROR: could not serialize access due to concurrent update

https://www.postgresql.org/docs/10/transaction-iso.html#XACT-SERIALIZABLE



7

MVCC IMPLEMENTATIONS

	Protocol	*Version Storage*	*Garbage Collection*	*Indexes*
Oracle	~~MV2PL~~	Delta	Vacuum	Logical
Postgres	~~MV-2PL/MV-TO~~	Append-Only	Vacuum	Physical
MySQL-InnoDB	MV-2PL	Delta	Vacuum	Logical
HYRISE	MV-OCC	Append-Only	–	Physical
Hekaton	MV-OCC	Append-Only	Cooperative	Physical
MemSQL	MV-OCC	Append-Only	Vacuum	Physical
SAP HANA	MV-2PL	Time-travel	Hybrid	Logical
NuoDB	MV-2PL	Append-Only	Vacuum	Logical
HyPer	MV-OCC	Delta	Txn-level	Logical

CARNEGIE MELLON DATABASE GROUP

CMU 15-721 (Spring 2017)

in postgres if we write isolation level serializable it internally switches to SSI which is a different protocol.

Table is neither up-to-date nor correct as of 2025. But it shows a couple of important points about MVCC:

- serializability under MVCC (including preventing write skew) needs additional protocol rules.
- MVCC writes tuple versions
- Version storage is handled differently
- Version storage demands for a regular garbage collections
- Index Issue: indexes must always point to the current version

Postgres MVCC Version Storage: Append-Only

Append-Only:

1. All physical versions of a logical tuple are stored in the same table (heap file).
2. New versions are appended into the table.
3. Versions are "mixed", versions of different rows follow each other.

page 0

row-offset	Version	Key	Balance	created_by	Deleted_by	Offset Pointer	
1	A0	Harry	0	1	~~0~~ 6	3	not hot update
2	B0	Rose	5	5	~~0~~ 7		
3	A1	HARRY	10	6	~~0~~ 8	5	hot update
4	B1	ROSE	4	7	0		
5	A2	HARRY	9	8	0		hot update

T6: Harry buys 10 lessons.
T7: Rose takes a lesson.
T8: Harry takes a lesson.

SELECT * FROM student where s_username='HARRY'

How is a select on key Harry executed?

1. PK BTREE -> 1
2. heap file: offset 1
3. heap file: offset 3
4. heap file: offset 5
5.

HOT Updates

HOT updates effectively speed up updates:

- HOT update version is written on the same page (But: not all update versions on the same page need to be HOT updates!)
- HOT updates cannot cross page boundaries.
  → if a page is full, no HOT updates are possible
- HOT update version are NOT referenced from outside the page but only inside the page via a HOT chain.
- → indexes can be left unchanged provides speed
- especially useful for tables with columns that get a lot of updates (like our teacher and student tables with counter columns)

Two necessary requirements for HOT Updates

1. Updated column MUST NOT be an indexed column
   (If the update is executed on an indexed column, the update changes the BTREE index itself: the index needs to get a new entry that references the updated tuple. As it is not a HOT update, the PK index and all other indexes are updated, as well.
   → do not put indexes on heavily updated attributes (see databases 1: Where to put indexes)

2. There must be space on the page for new versions
   By default, Postgres fills pages 100%. However, the fill factor parameter (for heap files and index files!) directs Postgres to initially leave extra space on a page that can be used for HOT updates. by doing this we can control how fast our database is

MVCC Version Storage

HOT-Update: Heap Onle Tuple Update

1. HOT Tuples are not referenced by the indexes directly but via the HOT chain.
2. Index points to the 1st offset (simplified!), then walks through the HOT chain.
3. If HOT chain gets long and contains dead versions (tuples that no transactions need to see anymore), intermediate offsets may be removed (freed up). Root offset is always kept as entrance point for index reference.

page 0

offset	Version	Key	Field Balance	created_by Xmin	Deleted_by Xmax	Offset-Pointer
1	A0	Harry	0	1	6	3
2	B0	Rose	5	5	7	4
3	A1	Harry	10	6	8	5
4	B1	Rose	4	7	0	
5	A2	Harry	9	8	0	

MVCC Version Storage

HOT-Update: Heap Onle Tuple Update

1. HOT Tuples are not referenced by the indexes directly but via the HOT chain.
2. Index points to the 1st offset (simplified!), then walks through the HOT chain.
3. If HOT chain gets long and contains dead versions (tuples that no transactions need to see anymore), intermediate offsets may be removed (freed up). Root offset is always kept as entrance point for index reference.

page 0

offset	Version	Key	Field Balance	created_by Xmin	Deleted_by Xmax	Offset-Pointer
1	A0	Harry	0	1	6	6
2	B0	Rose	5	5	7	4
3	A1	Harry	10	6	8	freed up
4	B1	Rose	4	7	0	
5	A2	Harry	9	8	9	freed up
6	A3	Harry	8	9	0	

so now A0 version directly points to the last offset.

Fill Factor

1. ALTER TABLE teacher SET (fillfactor = 60);
   Sets fillfactor for a table to a specific value

2. SELECT relname, reloptions FROM pg_class WHERE relname = 'teacher';
   Returns the filllfactor for table teacher

3. SELECT indexname, indexdef FROM pg_indexes WHERE tablename = teacher';
   Returns the indexes defined on table teacher

4. SELECT relname AS index_name, reloptions FROM pg_class WHERE relkind = 'i' AND relname = 'teacher_pkey';
   Returns the fillfactor of the index

5. ALTER INDEX teacher_pkey SET (fillfactor = 80);
   Sets fillfactor for an index to a specific value

HOT and COLD Updates

```
/* create two indexes in addition to PK */
CREATE INDEX i_t_name ON teacher (t_name);
CREATE INDEX i_t_postalcode ON teacher (t_postalcode);

SELECT pg_current_wal_lsn();

/* run an update on one indexed column and on one not indexed column */
update public.teacher SET t_name= 'Duerr' where t_id = 8;
update public.teacher SET t_payment = 20 where t_id = 7;

SELECT pg_current_wal_lsn();

Run pg_waldump
```

cold update needs to access different btrees and hot update only the heap

```
rmgr: Heap        len (rec/tot):     65/  2521, tx:       3286, lsn: 0/08D490F0, prev 0/08D490B8, desc: UPDATE old_xmax: 3286, old_off: 24,
old_infobits: [], flags: 0x60, new_xmax: 0, new_off: 26, blkref #0: rel 1663/16396/17198 blk 0 FPW
rmgr: Btree       len (rec/tot):     53/   573, tx:       3286, lsn: 0/08D49AD0, prev 0/08D490F0, desc: INSERT_LEAF off: 10, blkref #0: rel
1663/16396/17203 blk 1 FPW
rmgr: Btree       len (rec/tot):     53/   573, tx:       3286, lsn: 0/08D49D10, prev 0/08D49AD0, desc: INSERT_LEAF off: 4, blkref #0: rel
1663/16396/93062 blk 1 FPW
rmgr: Btree       len (rec/tot):     64/    64, tx:       3286, lsn: 0/08D49F50, prev 0/08D49D10, desc: INSERT_LEAF off: 4, blkref #0: rel
1663/16396/93063 blk 1
rmgr: Transaction len (rec/tot):     34/    34, tx:       3286, lsn: 0/08D49F90, prev 0/08D49F50, desc: COMMIT 2025-03-26
21:40:41.419252 RTZ 2 Normalzeit
rmgr: Standby     len (rec/tot):     50/    50, tx:          0, lsn: 0/08D49FB8, prev 0/08D49F90, desc: RUNNING_XACTS nextXid 3287
latestCompletedXid 3286 oldestRunningXid 3287
rmgr: Heap        len (rec/tot):     79/    79, tx:       3287, lsn: 0/08D49FF0, prev 0/08D49FB8, desc: HOT_UPDATE old_xmax: 3287, old_off:
25, old_infobits: [], flags: 0x60, new_xmax: 0, new_off: 27, blkref #0: rel 1663/16396/17198 blk 0
rmgr: Transaction len (rec/tot):     34/    34, tx:       3287, lsn: 0/08D4A058, prev 0/08D49FF0, desc: COMMIT 2025-03-26
21:40:50.942903 RTZ 2 Normalzeit
```

MVCC

MVCC Concepts – but no standard

1. Readers do not block writers and writers do not block readers.
2. Conflicting writes: may lead to lost updates or need to be rolles back.
3. Serializability: needs additional protocol rules
4. Updates create new versions of a row.
5. Version Storage: Pointers are used to create a version chain.

1. Append-Only:	Postgres
2. Delta:	Oracle / MYSQL: old versions kept in the undo log
3. Time-Travel:	SAP Hana: old versions kept in Time-Travel Storage

6. Garbage Collection to remove old versions.
7. Indexes: needs to point to or find the current version of a tuple

You need to understand the specific implementations of the database you use in order to set the isolation levels correctly and in order to know about and use database features that speed up performance.

Additional MVCC Implementation Information on MySQL / MariaDB / SAP Hana

this is not included in the exam so you can skip until dynamic sql statements

Mariadb Expand

MariaDB Expand row version format

Record Type
NULL-column indicator
Length of non-NULL variable length column values
Next Record Offset
Key
row-store of non-key / non-null fields
TransactionID
InvocationID
CommitID
Roll-Back Pointer (Log-Sequence-NumberID)

ID	Description
xID	is generated when the transaction begins (TS Start)
iID	is generated when a statement begins execution within the transaction (TS start statement)
cID	is generated when the transaction is committed (TS commit)

What requirement holds in regard to the cID in order to safely remove the row version?
Database keeps a list with all currently active transaction xIDs.

https://mariadb.com/docs/xpand/architecture/components/xpand/concurrency/mvcc/

MySQL / MariaDB / Oracle: Delta Version Storage

Delta-Storage

1. On every update, the old version of the row is written into the undo log.
2. The undo log is used to restore data after rollback and to read old versions.
3. Table itself contains current version only.
4. Every row version contains a pointer that points to the undo log and the previous version of the row.

RowID	Key	Field LessonBalance	xID	DeleteFlag	Roll-Bck-Ptr
1	Harry	0	1	0	1
2	Rose	5	5	0	15

INNODB MVCC

Internally, InnoDB adds three fields to each row stored in the database:

A 6-byte DB_TRX_ID field indicates the transaction identifier for the last transaction that inserted or updated the row. Also, a deletion is treated internally as an update where a special bit in the row is set to mark it as deleted.

A 7-byte DB_ROLL_PTR field called the roll pointer. The roll pointer points to an undo log record written to the rollback segment. If the row was updated, the undo log record contains the information necessary to rebuild the content of the row before it was updated.

A 6-byte DB_ROW_ID field contains a row ID that increases monotonically as new rows are inserted. If InnoDB generates a clustered index automatically, the index contains row ID values. Otherwise, the DB_ROW_ID column does not appear in any index.

https://dev.mysql.com/doc/refman/8.0/en/innodb-multi-versioning.html

SAP Hana: Time Travel Version Storage

TimeTravel Storage

1. On every update, the old version of the row is moved to a second table.
2. The main table always holds the current version.
3. Pointers in Time-Travel table go from New-to-Old
4. Pointers from main table to time-travel table need to be updated on every update.

MVCC Time-Travel Version Storage

Main Table

row ID	Version	Key	Balance	created_by	Deleted_by	Pointer
1	A0	Harry	0	T1	0	-
2	B0	Rose	5	T5	0	-

Time-Travel Table

row ID	Version	Key	Balance	created_by	Deleted_by	Pointer
1						
2						
3						

T6: Harry buys 10 lessons.
T7: Rose takes a lesson.
T8: Harry takes a lesson

What does the main table look like after the updates?

What does the time-travel table look like after the updates?

MVCC Time-Travel Version Storage

Main Table

row ID	Version	Key	Balance	created_by	Deleted_by	Pointer
1	A1	Harry	10	T6	0	TTT row1
2	B0	Rose	5	T5	0	-

Time-TravelTable

row ID	Version	Key	Balance	created_by	Deleted_by	Pointer
1	A0	Harry	0	T1	T6	-
2						
3						

T6: Harry buys 10 lessons.
T7: Rose takes a lesson.
T8: Harry takes a lesson

What does the main table look like after the updates?

What does the time-travel table look like after the updates?

KUTAISI INTERNATIONAL UNIVERSITY

MVCC Time-Travel Version Storage

Main Table

row ID	Version	Key	Balance	created_by	Deleted_by	Pointer
1	A1	Harry	10	T6	0	TTT row1
2	B1	Rose	4	T7	0	TTT row2

Time-TravelTable

row ID	Version	Key	Balance	created_by	Deleted_by	Pointer
1	A0	Harry	0	T1	T6	-
2	B0	Rose	5	T5	T7	
3						

T6: Harry buys 10 lessons.
T7: Rose takes a lesson.
T8: Harry takes a lesson

What does the main table look like after the updates?

What does the time-travel table look like after the updates?

MVCC Time-Travel Version Storage

Main Table

row ID	Version	Key	Balance	created_by	Deleted_by	Pointer
1	A2	Harry	9	T8	0	TTT row3
2	B1	Rose	4	T7	0	TTT row2

Time-TravelTable

row ID	Version	Key	Balance	created_by	Deleted_by	Pointer
1	A0	Harry	0	T1	T6	-
2	B0	Rose	5	T5	T7	
3	A1	Harry	10	T6	0	TTT row1

T6: Harry buys 10 lessons.
T7: Rose takes a lesson.
T8: Harry takes a lesson

What does the main table look like after the updates?

What does the time-travel table look like after the updates?

Dynamic SQL Statements

Dynamic SQL Statements

SQL statements are usually created at runtime, based on user input or other application needs.
→ using this input, a SQL string is dynamically constructed (concatenated) and sent to the database to execute.

PHP example:

```
// connect to the database
$conn = mysql_connect("localhost","username","password");
// dynamically build the sql statement with the input
$query = "SELECT * FROM teacher WHERE city  = '$_GET["val"]' " ;
// execute the query against the database
$result = mysql_query($query);
```

If the user inputs the value *Kutaisi,* then the query string to be executed is:

```
SELECT * FROM teacher WHERE city = 'Kutaisi';
```

Dynamic SQL Statements

SQL Dynamic statement for a user login:

```
// connect to the database
$conn = mysql_connect("localhost","username","password");
// dynamically build the sql statement with the input
$query = "SELECT userid FROM users WHERE user = '$_GET["user"]' " .
"AND password = '$_GET["password"]'";
// execute the query against the database
$result = mysql_query($query);
```

user inputs: username=foo
                password=bar

Query string to be sent to DB and executed:

```
SELECT userid FROM users WHERE user = 'foo' AND password = 'bar';
```

KUTAISI INTERNATIONAL UNIVERSITY

Dynamic SQL Statements

user inputs: username=foo

password=bar' OR '1'='1

Query string to be built and sent to DB and executed?

If the website is poorly coded, the hacker can send special input that modifies these queries and makes the database give up private data!

SELECT userid FROM users WHERE user = 'foo' AND

password = 'bar' OR '1'='1' [redacted]

we would get all users in browser [redacted]

[redacted]

Result: [redacted]

Dynamically built SQL strings can contain harmful content.

[redacted]

[redacted]

SQL Injection

SQL injection:
the application builds and sends malicious SQL strings to the database for execution.

Inserting rows into a student table:
INSERT INTO student (firstname) VALUES ('Elaine');

Here comes a student with a strange name: Robert');DROP TABLE students;--

What SQL string gets sent to the database?

```
INSERT INTO student (firstname) VALUES (' Robert');
DROP TABLE students;
');
```

Root of the problem: code is mixed into a field that should contain only data.

Defenses against SQL Injection

1. Sanitize user input manually ("escape" user input)

- modifing user input in such a way that input cannot be interpreted as code by database anymore.

- How would the input Robert');DROP TABLE Students;--
  would have to be escaped?

- The single quote after Robert must not interpreted as end of the data part but as part of the value.

..> adding another single quote means including the first single quote as part of the value string.

INSERT INTO student (firstname) VALUES

Defenses against SQL Injection

2. Parameterized queries (prepared statements)

- send query (code) and data separately,
- prepared statements – because the query is prepared and sent first and the values come in later
- when data comes in separate and later it is bound to the query and cannot be interpreted as code.

PHP example:

```
$con = new mysqli("localhost", "username", "password", "db");
$sql = "SELECT * FROM users WHERE username=? AND password=?";
$cmd = $con->prepare($sql);

// Add parameters to SQL query
$cmd->bind_param("ss", $username, $password); // bind parameters as strings
$cmd->execute();
```

When you use a prepared statement with ?, the database locks the query structure first, before adding values. At this stage, the database knows it will receive two separate values (not SQL code).

Here, $username and $password are sent as data, not code. This means even if a hacker enters OR 1=1. it is treated as a normal username, not part of the SQL query!

Defenses against SQL Injection

2. Parameterized queries (prepared statements)
- are the most important defense against SQL injection

Data values can be parameterized, not column names / identifiers
Invaliud parameterized examples:

```
SELECT * FROM ? WHERE username = 'john'
SELECT ? FROM users WHERE username = 'john'
SELECT * FROM users WHERE username LIKE 'j%' ORDER BY ?
```

Defenses against SQL Injection

3. Input Validation

- using regular expressions or business logic

Example:

Using Reg Expression Validation to validate that a username contains only letters (uppercase and lowercase) and is between eight and 12 characters long:

```
$username = $_POST['username'];
if (!preg_match("/^[a-zA-Z]{8,12}$/D", $username) {
// handle failed validation
}
```