HPI Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Concurrency Control in Databases

Motivation

- **A**tomicity
  - Transactions (Txs) either *commit* fully or not at all
- **C**onsistency
- **I**solation
  - Concurrent transactions must not affect each other
- **D**urability

Motivation – Textbook Example

CustomerName	Balance
John Smith	1,000.00 €
…	…

Transaction 1: John Smith pays 300 Euros for a flight

Tx 2: John Smith receives his 3000 Euro paycheck

Tx 1: UPDATE customers SET balance = balance – 300 …;

Tx 2: UPDATE customers SET balance = balance + 3000 …;

Tx 1: The DBMS reads the current balance

Tx 2: The DBMS reads the current balance

Tx 1: The DBMS calculates the new balance and writes it back

Tx 2: The DBMS calculates the new balance and writes it back

How much money does John have?

Types of Isolation Errors

- Lost Update
  - Two Tx modify the same row, but only one update is saved.
- Dirty Reads
  - A Tx reads changes that have not been committed yet.
- Non-Repeatable Reads
  - A Tx reads the same row twice and gets different results.
- Phantom Reads
  - A Tx sees rows that have been inserted during its lifetime.

Isolation Methods

- In general, databases have two ways to deal with concurrent modifications:
  - Pessimistic Two-Phase Locking (2PL)
  - Optimistic Concurrency Control (OCC)
- Hyrise and HANA use different flavors of OCC
  - Good if few conflicts are expected.

Two-Phase Locking (2PL)

- For each row, two types of locks can be held by transactions:
  - A read lock, which makes sure that the row's contents do not change while the Tx is ongoing.
  - Multiple transactions can hold a read lock on one row.
  - A write lock, which guarantees that the row is not modified or read by any other Tx.
  - If a write lock is held, no other Tx can have a lock.

Two-Phase Locking (2PL)

- Each transaction has two *phases*:
  - In the *expanding phase*, Txs may acquire new locks, but cannot release held locks.
  - Read locks cannot be upgraded to write.
  - In the *shrinking phase*, Txs may release locks, but cannot acquire new ones.
- It can be proven that this guarantees that no isolation conflicts can occur.
- Other flavors of 2PL (e.g., with upgrades) exist.

Example without 2PL
X sends money to Y in two transactions (5€ and 6€)

Transaction A if(X >= 5): Y = Y + 5 X = X - 5	Transaction B if(X >= 6): Y = Y + 6 X = X - 6	Value of X	Value of Y
Read X (= 7)		7	0
is sufficient			
Read Y (= 0)			
Write Y = 5			5
Read X (= 7)			
Write X = 2		2	
	Read X (= 2)		
	is insufficient		

2 + 5 = 7: no conflict

Example without 2PL
X sends money to Y in two transactions (5€ and 6€)

Transaction A if(X >= 5): Y = Y + 5 X = X - 5	Transaction B if(X >= 6): Y = Y + 6 X = X - 6	Value of X	Value of Y
Read X (= 7)		7	0
is sufficient			
Read Y (= 0)			
	Read X (= 7)		
	is sufficient		
Write Y = 5			5
Read X (= 7)			
Write X = 2		2	
	Read Y (= 5)		
	Write Y (= 11)		11
	Read X (= 2)		
	Write X = 1	1	

1 + 11 = 12: conflict

Example with 2PL
X sends money to Y in two transactions (5€ and 6€)

Transaction A if(X >= 5): Y = Y + 5 X = X - 5	Transaction B if(X >= 6): Y = Y + 6 X = X - 6	X	Y
Write**L**ock(X)		7	0
Read X (= 7)			
is sufficient			
	WL(X) - fails		
WL(Y)			
Read Y (= 0)			
Write Y = 5			
Un**L**ock(Y)			5
Write X = 2		2	
UL(X)			

Transaction A if(X >= 5): Y = Y + 5 X = X - 5	Transaction B if(X >= 6): Y = Y + 6 X = X - 6	X	Y
	WL(X)		
	Read X (= 2)		
	is insufficient		
	UL(X)		

2 + 5 = 7: no conflict

Example with 2PL
X sends money to Y in two transactions (5€ and 6€)

- Why do we have to obey the two-phase protocol?
- We could think about releasing X as soon as possible to reduce lock duration.

Example with 2PL
X sends money to Y in two transactions (5€ and 6€)

Transaction A if(X >= 5): Y = Y + 5 X = X - 5	Transaction B if(X >= 6): Y = Y + 6 X = X - 6	X	Y
WL(X)		7	0
Read X (= 7)			
is sufficient			
UL(x)			
	WL(X)		
	Read X (= 7)		
	is sufficient		
	UL(x)		
WL(Y)			
Read Y (= 0)			
Write Y = 5			5
UnLock(Y)			

Transaction A if(X >= 5): Y = Y + 5 X = X - 5	Transaction B if(X >= 6): Y = Y + 6 X = X - 6	X	Y
WL(X)			
Write X = 2		2	
UL(X)			
	WL(Y)		
	Read Y (= 5)		
	Write Y = 11		11
	UL(Y)		
	WL(X)		
	Read X = 2		
	Write X = 1	1	
	UL(X)		

1 + 11 = 12: conflict

Problems with 2PL

- Great care has to be taken to not hold locks longer than necessary.
- Deadlocks may occur – the DBMS has to actively monitor locks that are acquired and terminate Txs that run into a deadlock.
- Acquiring locks requires synchronization across all CPU cores and is expensive.
- Most of the time, we will not run into any conflicts – why not be more optimistic?

Optimistic Concurrency Control

- **Timestamp-Based Concurrency Control** gives each row two identifiers: The last transaction that read it and the last transaction that modified it.
- If the write timestamp was changed by a Tx started after “our” transaction, a conflict is detected and our Tx has to abort.
- What is the difference to 2PL? With 2PL, we are waiting for the lock. Here, we are detecting conflicts and roll back.

Optimistic Concurrency Control

- **Multiversion Concurrency Control (MVCC)** is different in that no rows are actually changed. Instead, old rows are invalidated and new rows are inserted. This method is called *insert-only*.
- Each transaction operates on a version of the database that corresponds to the "truth" when the transaction started. This version does not change and is always available.
- In case of conflicts (i.e., trying to invalidate an already invalidated row), the last transaction has to abort.

Multiversion Concurrency Control

- We do this by storing the commit id of the TX that added or deleted a row:

AccountId	Balance	BeginCid	EndCid
00119	1,000.00 €	0	5
00119	700.00 €	5	∞
00120	…		

- Additionally, we keep the transaction id to lock and mark transactions that are currently undergoing modifications.
  - We will not discuss this part in the lecture.

Example with MVCC

Account	Balance	BeginCid	EndCid
X	7	5	∞
Y	0	5	∞

Global LastCid	5

Stores the commit id of the last Tx that committed, i.e., the timestamp of „now"

- We use the same example as for 2PL.
- Both rows were inserted by the last Tx, commit id 5.
- They are visible to future Txs because the BeginCid is equal and the EndCid is larger than the last commit id of new Txs.

For this lecture, we leave out some parts of MVCC (such as the two-phase commit) and discuss MVCC only on a conceptual level. For more details, please refer to the HYRISE documentation and Schwalb et al.: Efficient Transaction Processing for Hyrise in Mixed Workload Environments

Example with MVCC

Account	Balance	BeginCid	EndCid
X	7	5	∞
Y	0	5	∞

Global LastCid	5
LastCid TxA	5

- Transaction A starts and retrieves the lastCid from the global store.

Example with MVCC

Account	Balance	BeginCid	EndCid
X	7	5	∞
Y	0	5	∞

Global LastCid	5
LastCid TxA	5

- It performs the check for sufficient funds in the account of X.

  Row 0 is visible because $5 \leq 5 < \infty$.

Example with MVCC

Account	Balance	BeginCid	EndCid
X	7	5	∞
Y	0	5	**6**
Y	5	6	∞

Global LastCid	6
LastCid TxA	5

- 5 € are added to the account of Y. This happens by invalidating the old row and adding a new row.
- For simplicity, let us assume that TxA has commit id 6.

Example with MVCC

Account	Balance	BeginCid	EndCid
X	7	5	**6**
Y	0	5	6
Y	5	6	∞
X	2	6	∞

Global LastCid	6
LastCid TxA	5

- 5 € are subtracted from the account of X. Again, the old row is invalidated and the updated version of the row is appended.

Example with MVCC

Account	Balance	BeginCid	EndCid
X	7	5	6
Y	0	5	6
Y	5	6	∞
X	2	6	∞

Global LastCid	6
LastCid TxA	completed
LastCid TxB	6

- Transaction B starts. It retrieves the global LastCID 6.
- Row 0 cannot be used for validating the balance, because its EndCid is not greater that TxB‘s LastCid. It is invisible.
- Instead, row 3 is used for verification.

Multiversion Concurrency Control

- It is impossible to recover from conflicts.
- If a transaction finds that the row that it wants to invalidate has already been invalidated by another Tx, it has to abort and rollback all its changes.
- There is no guarantee that a Tx that started first will win this conflict. For highly contested rows, starvation is possible.

Does this solve all problems?

- In the real world, you should not store the aggregated balance of a customer accounts.
- Instead, calculate the balance on-the-fly by looking at the bookings.
- What does this mean for our example?

Does this solve all problems?

Account	TransactionDate	Amount
X	2017-05-27 22:39:01	-5
Y	2017-05-27 22:39:01	5

Ok, but what if one transaction influences another one?

Does this solve all problems?

Flight	Seat	Passenger
…	…	…

Transaction 1: John Smith books seat 5E on flight AB 7211

Tx 2: Joseph Westport books the same seat

Tx 1: INSERT INTO bookings VALUES ('AB7211', '5E', 'John Smith')

Tx 2: INSERT INTO bookings VALUES ('AB7211', '5E', 'Joseph Westport')

Flight	Seat	Passenger
AB7211	5E	John Smith
AB7211	5E	Joseph Westport

No row conflicts, yet the database is inconsistent

Serializable Snapshot Isolation

- Serializable Snapshot Isolation (SSI) adds to a simple snapshot method (e.g., MVCC) by making sure that transactions leave the database in a state exactly equal to what would happen if they executed sequentially.
- It was first implemented in PostgreSQL as late as 2012:

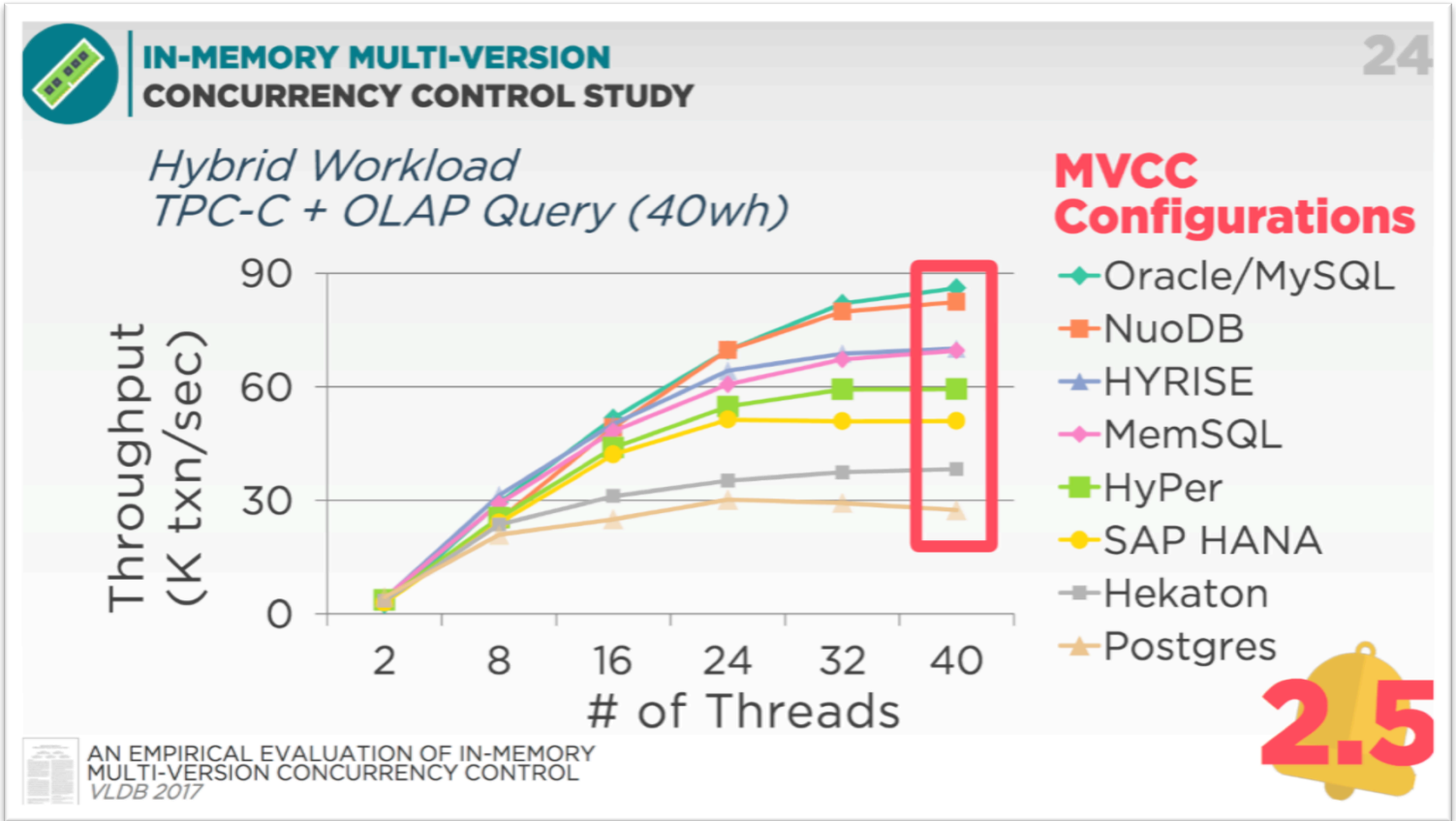
> PostgreSQL 9.1's serializable isolation level is effective: it provides true serializability but allows more concurrency than two-phase locking. Our experiments with a transaction processing and a web application benchmark show that our serializable mode has a performance cost of less than 7% relative to snapshot isolation, and outperforms two-phase locking significantly on some workloads.

Dan R. K. Ports and Kevin Grittner. 2012. Serializable snapshot isolation in PostgreSQL. Proc. VLDB Endow. 5, 12

Differences in Concurrency Control Schemes

- This all looks like a solved problem. Why should we care about this?
- With a growing degree of parallelism, choosing the right concurrency control scheme becomes increasingly important.
- Even within MVCC, subtle implementation changes can make a significant difference in performance.

Differences in Concurrency Control Schemes



Summary

- Databases need to take great care to guarantee that multiple transactions are correctly isolated from each other.
- Four types of row-level conflicts exist: Lost update, Dirty, Non-Repeatable, Phantom.
- Concurrency schemes are either pessimistic (e.g., 2PL) or optimistic (e.g., MVCC).
- 2PL works by having sharable read locks and exclusive write locks that are acquired in the first phase and released in the second phase.
- MVCC stores the timestamps at which rows became first visible and at which they were invalidated. This allows each transaction to work on its own copy of the database. In case of conflicts, transactions have to abort.
- In many cases, additional database constraints and/or application logic is needed to guarantee consistency.