Transactions: Overview

Christoph Koch
School of Computer & Communication Sciences, EPFL

Transactions

- Concurrent execution of user programs is essential for good DBMS performance.
- Users want to access database concurrently.

Abstraction: A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.

- *A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned with what data is read/written from/to the database.*

Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself (isolation).
- Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
  - *DBMS will enforce some integrity constraints.*
  - *Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).*

Atomicity of Transactions

- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are *atomic*. That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - *DBMS logs all actions so that it can undo the actions of aborted transactions.*

The ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability

Example

- Consider two transactions (*Xacts*):

```
T1:  BEGIN   A=A+100,   B=B-100   END
T2:  BEGIN   A=1.06*A,   B=1.06*B   END
```

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

Example (Contd.)

- Consider a possible interleaving (*schedule*):

```
T1: A=A+100,                 B=B-100
T2:          A=1.06*A,               B=1.06*B
```

- This is OK. But what about:

```
T1: A=A+100,                          B=B-100
T2:          A=1.06*A, B=1.06*B
```

- The system's view of the second schedule:

```
T1: R(A), W(A),                                 R(B), W(B)
T2:             R(A), W(A), R(B), W(B)
```

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

```
T1: R(A), W(A),               R(B), W(B), Abort
T2:             R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts):

```
T1: R(A),               R(A), W(A), C
T2:       R(A), W(A), C
```

- Overwriting Uncommitted Data (WW Conflicts):

```
T1: W(A),               W(B), C
T2:       W(A), W(B), C
```

Aborting a Transaction

- If a transaction *Ti* is aborted, all its actions have to be undone.
- Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
  - *Or: If Ti writes an object, Tj can read this only after Ti commits.*
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded.
  - *This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.*

Concurrency Control: Conflict Serializability and Locking

Christoph Koch

School of Computer & Communication Sciences, EPFL

Conflict Serializable Schedules

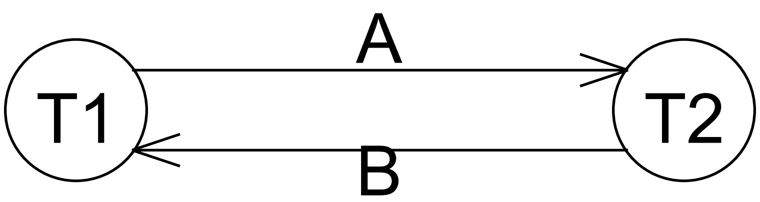
- Conflict cases: R-W, W-R, W-W (see anomalies).
- Two schedules are conflict equivalent if:
  - *They involve the same actions of the same transactions*
  - *Every pair of conflicting actions is ordered the same way*
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Example

- A schedule that is not conflict serializable:

```
T1: R(A), W(A),                        R(B), W(B)
T2:             R(A), W(A), R(B), W(B)
```

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.



Dependency graph

Dependency Graph

- Dependency graph: One node per Xact; edge from *Ti* to *Tj* if there is an object A such that
  - *Ti writes A before Tj reads or writes A or*
  - *Tj reads A before Tj writes A.*
- Theorem: A schedule is conflict serializable if and only if its dependency graph is acyclic.

Lock-Based Concurrency Control

- Two-Phase Locking Protocol
  - *Each Xact must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.*
  - *A transaction cannot request additional locks once it releases any locks.*
  - *If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.*
- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - *Like 2PL, but all locks held by a transaction are released at the same time when the transaction completes.*
- 2PL allows only schedules whose precedence graph is acyclic => serializable.
- Strict 2PL additionally simplifies transaction aborts
  - *(Non-strict) 2PL involves more complex abort processing.*

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - *Deadlock detection (build waits-for dependency graph)*
  - *Deadlock prevention (timestamp-based priorities make deadlocks impossible)*

Cascading aborts and unrecoverable schedules

- Unrecoverable schedule: A transaction T2 has committed after reading an object written by T1. If T1 wants to abort, there is a serious problem.
- Cascading abort. T1 writes an object before T2 reads the same object. Now T1 aborts. T2 must abort too. (In general, an arbitrarily long chain of aborts may be triggered.)

- Does strict 2PL feature either of these two issues?
- Does non-strict 2PL?

Dynamic Databases

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:
  - *T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor (say, age = 71).*
  - *Next, T2 inserts a new sailor; rating = 1, age = 96.*
  - *T2 also deletes oldest sailor with rating = 2 (and, say, age = 80), and commits.*
  - *T1 now locks all pages containing sailor records with rating = 2, and finds oldest (say, age = 63).*
- No consistent DB state where T1 is “correct”!

```
T1: S(A*) R(A*)                           S(B*) R(B*) W(C)
T2:             X(A') I(A') X(B) D(B)
```

The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1.
  - *Assumption only holds if no sailor records are added while T1 is executing!*
  - *Need some mechanism to enforce this assumption. (Index locking and predicate locking.)*
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!

Predicate Locking

- Grant lock on all records that satisfy some logical predicate, e.g. *age > 2*salary*.
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
  - *What is the predicate in the sailor example?*
- In general, predicate locking has a lot of locking overhead.

Serializable Multiversion Concurrency Control (MVCC)

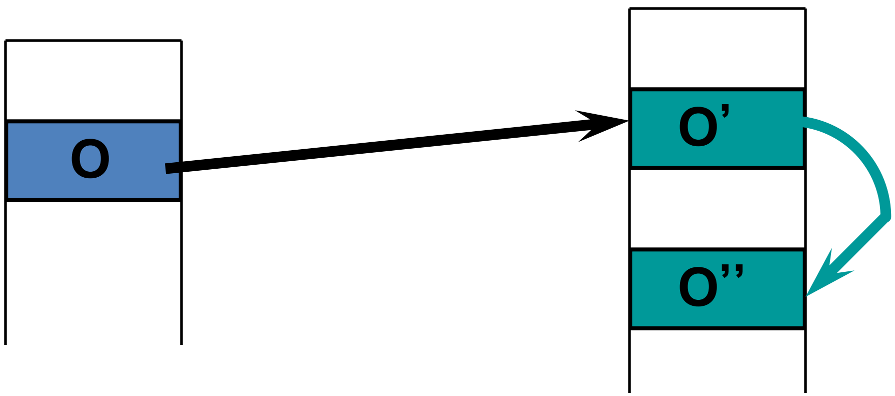
Christoph Koch

School of Computer & Communication Sciences, EPFL

Multiversion CC

- **Idea:** Let writers make a "new" copy while readers use an appropriate "old" copy:

MAIN SEGMENT (Current versions of DB objects)



VERSION POOL (Older versions that may be useful for some active readers.)

Multiversion CC (Contd.)

- Each version of an object has its writer's TS as its WTS, and the TS of the Xact that most recently read this version as its RTS.
- Versions are chained backward; we can discard versions that are "too old to be of interest" (garbage collection).
- Each Xact is classified as Reader or Writer.
  - *Writer may write some object; Reader never will.*
  - *Xact declares whether it is a Reader when it begins.*
- Readers are always allowed to proceed.
  - *But may be blocked until writer commits.*

Reader Xact

- For each object to be read:
  - *Finds **newest version** V with WTS(V) <= TS(T). (Starts with current version in the main segment and chains backward through earlier versions.)*
  - *Sets RTS(V) to max(RTS(V), TS(V))*
- Assuming that some version of every object exists from the beginning of time, Reader Xacts are never restarted.
  - *However, might block until writer of the appropriate version commits (to deal with unrecoverable schedules).*

Writer Xact

- To read an object, follows reader xact protocol (previous slide).
- To write an object:
  - *Finds **newest version V** s.t. WTS <= TS(T).*
  - *If RTS(V) <= TS(T), and WTS(V) < TS(T) T makes a copy V' of V, with a pointer to V, with WTS(V') = TS(T), RTS(V') = TS(T).*
  - *Else if RTS(V) = WTS(V) = TS(T), T overwrites the value of V with the new value.*
  - *Else, reject write.(Abort transaction)*

Example

```
T1: R(A)      W(B)
T2:      R(B)      W(A)
```

- Initially, we have versions A@0 (RTS=0, WTS=0), B@0 (RTS=0, WTS=0)
- Op T1:R(A) : read from A@0, set RTS=1
- Op T2:R(B): read from B@0, set RTS=2
- Op T1:W(B): RTS of B@0 is too high, abort T1 and restart as T3. We get

```
T2:      R(B)           W(A)
T3:           R(A) W(B)
```

- Op T3:R(A): read from A@0, set RTS=3
- Op T3: W(B): create new version B@3 (RTS=3, WTS=3)
- Op T2: W(A): RTS of A@0 is 3, abort T2 and restart as T3.

```
T3:           R(A) W(B)
T4:                     R(B) W(A)
```

- Op T4:R(B): read B@3, set RTS=4
- Op T4: W(A): : create new version B@4 (RTS=4, WTS=4)
-

Committing in MVCC

- Assume we want to avoid unrecoverable schedules.
- Avoid scenario that writer transaction gets aborted and another transaction has read a dirty value.
- Analysis at commit time.
- Example:

```
T1: W(A)        Abort
T2:        R(A)
```

  - *Here, the abort of T1 must trigger the abort of T2.*
- Optimization with dedicated reader transactions: If writer xacts are rare (and short running), read transactions may block already at read time to reduce aborts forced by protocol.
  - *Reader Xacts will never abort.*

Snapshot Isolation

Christoph Koch

School of Computer & Communication Sciences, EPFL

Transaction Support in SQL-92

- Each transaction has an access mode, a diagnostics size, and an isolation level.

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No

Snapshot isolation

- Snapshot isolation (SI) is the most popular mechanism in real DBMS.
  - *Implemented in Oracle, MS SQL Server, Postgres.*
- ISOLATION LEVEL "Serializable".

- But does not guarantee serializability!!!
  - *There is a patch ("serializable SI"). Implemented in recent versions of Postgres.*

Snapshot isolation

- Conceptually, Xact works on a copy made at Xact start time.
  - *Not implemented that way.*
  - *Guarantees that reads in the Xact see a consistent version of the database.*
- Commit only if no updates of Xact conflict with updates made since the snapshot.
- *Write skew* anomaly:

T1:	R(A)		W(B)		C
T2:		R(B)		W(A)	C

  - *Not serializable, but permitted by snapshot isolation!*

Discussion

- SI is related to optimistic CC, in that
  - *Conceptually, snapshots are created at Xact start.*
  - *There is an analysis phase at the end to decide whether a transaction may commit (do writesets overlap?).*
- Multiversion CC is a way to implement snapshot isolation.
  - *SI is MVCC where WR conflicts are not appropriately caught.*
  - *But do not need to maintain RTS, WTS.*
- SI does NOT use locks a priori
  - *But locks are there to support distribution – Postgres can get into a deadlock.*