Database Tuning and Physical Design: Execution of Transactions

Fall 2017

School of Computer Science
University of Waterloo

Databases CS348
Query (and update) processing converts requests for sets of tuples to requests for reads and writes of physical objects in the database.

database objects (depending on granularity) can be
- individual attributes
- records
- physical pages
- files (only for concurrency control purposes)

Goals
⇒ correct and concurrent execution of queries and updates
⇒ guarantee that acknowledged updates are persistent
ACID Requirements

Transactions are said to have the ACID properties:

- **Atomicity**: all-or-nothing execution
- **Consistency**: execution preserves database integrity
- **Isolation**: transactions execute independently (as if they were executed in the system alone)
- **Durability**: updates made by a committed transaction will not be destroyed by subsequent failures.

Implementation of transactions in a DBMS comes in two parts:

- **Concurrency Control**: committed transactions do not interfere
- **Recovery Management**: committed transactions are durable, aborted transactions have no effect on the database
Concurrency Control: assumptions

1. we fix a database: a set of objects read/written by transactions:
   \[ r_i[x] \]: transaction \( T_i \) reads object \( x \)
   \[ w_i[x] \]: transaction \( T_i \) writes (modifies) object \( x \)

2. a transaction \( T_i \) is a sequence of operations
   \[ T_i = r_i[x_1], r_i[x_2], w_i[x_1], \ldots, r_i[x_4], w_i[x_2], c_i \]
   \( c_i \) is the **commit request** of \( T_i \).

3. for a **set of transactions** \( T_1, \ldots, T_k \) we want to produce a **schedule** \( S \) of operations such that
   \[ \Rightarrow \] every operation \( o_i \in T_i \) appears also in \( S \)
   \[ \Rightarrow \] \( T_i \)’s operations in \( S \) are ordered the same way as in \( T_i \)

Goal:

produce a **correct schedule** with **maximal parallelism**
Transactions and Schedules

If $T_i$ and $T_j$ are concurrent transactions, then it is always correct to schedule the operations in such a way that:

- $T_i$ will appear to precede $T_j$ meaning that $T_j$ will “see” all updates made by $T_i$, and $T_i$ will not see any updates made by $T_j$, or
- $T_i$ will appear to follow $T_j$, meaning that $T_i$ will see $T_j$’s updates and $T_j$ will not see $T_i$’s.

Idea how to define Correctness:

it must appear as if the transactions have been executed sequentially (in some *serial* order).
Serializable Schedules

**Definition**

An execution of is said to be **serializable** if it is equivalent to a serial execution of the same transactions.

**Example:**

- An interleaved execution of two transactions:
  \[ S_a = w_1[x] \quad r_2[x] \quad w_1[y] \quad r_2[y] \]

- An equivalent serial execution \((T_1, T_2)\):
  \[ S_b = w_1[x] \quad w_1[y] \quad r_2[x] \quad r_2[y] \]

- An interleaved execution with no equivalent serial execution:
  \[ S_c = w_1[x] \quad r_2[x] \quad r_2[y] \quad w_1[y] \]
How do we determine if two schedules are equivalent?

⇒ cannot be based on any particular database instance

Conflict Equivalence:

- two operations conflict if they
  (1) belong to different transactions
  (2) access the same data item \( x \)
  (3) at least one of them is a write operation \( w[x] \).
- we require that in two conflict-equivalent histories all conflicting operations are ordered the same way.
- yields conflict-serializable schedules

⇒ conflict-equivalent to a serial schedule

View Equivalence:

allows more schedules, but it is harder (NP-hard) to compute
Other Properties of Schedules

Serializability guarantees correctness. However, we’d like to avoid other unpleasant situations.

Recoverable Schedules: (RC)

- Transaction $T_j$ reads a value $T_i$ has written, $T_j$ succeeds to commit, and $T_i$ tries to abort (in this order)
  - to abort $T_2$ we need to undo effects of a committed transaction $T_1$.
  - commits only in order of the read-from dependency

Cascadeless Schedules (ACA):

- if $T_j$ above didn’t commit we can abort it:
  - may lead to cascading aborts of many transactions
  - no reading of uncommitted data
How to Get a Serializable Schedule?

So how do we build schedulers that produce serializable and cascadeless schedules?

The scheduler receives requests from the query processor(s). For each operation it chooses one of the following actions:

- execute it (by sending to a lower module),
- delay it (by inserting in some queue), or
- reject it (thereby causing abort of the transaction)
- ignore it (as it has no effect)

Two main kinds of schedulers:

- conservative (favors delaying operations)
- aggressive (favors rejecting operations)
Two Phase Locking (2PL)

Transactions must have a lock on objects before access:
- a **shared lock** is required to read an object
- an **exclusive lock** is required to write an object

It is *insufficient* just to acquire a lock, access the data item, and then release it immediately...

2PL Protocol

A transaction has to **acquire** all locks before it **releases** any of them.

**Theorem**

*Two-phase locking guarantees that the produced transaction schedules are (conflict) serializable.*

In practice: **STRICT 2PL** (locks held till commit; this guarantees ACA)
Deadlocks and What to do

With 2PL we may end with a **deadlock**:

\[ r_1[x], r_2[y], w_2[x](\text{blocked by } T_1), w_1[y](\text{blocked by } T_2) \]

How do we deal with this:

- **deadlock prevention**:
  - \( \Rightarrow \) locks granted only if they can’t lead to a deadlock.
  - \( \Rightarrow \) ordered data items and locks granted in this order.

- **deadlock detection**:
  - \( \Rightarrow \) wait for graphs and cycle detection.
  - \( \Rightarrow \) resolution: the system **aborts** one of the offending transactions (involuntary abort).

**in practice**: detection (or often just a timeout) and abort
Variations on Locking

- Multi-granularity Locking
  ⇒ not all locked objects have the same size
  ⇒ advantageous in presence of bulk vs. tiny updates

- Predicate Locking
  ⇒ locks based on selection predicate rather than on a value

- Tree Locking
  ⇒ tries to avoid congestion in roots of (B-)trees
  ⇒ allows relaxation of 2PL due to tree structure of data

- Lock Upgrade protocols

...
Inserts and Deletes

We have been assuming a **fixed set** of data items.

⇒ what if we try to *insert* or *delete* an item?

- does plain 2PL (correctly) handle this situation? NO:
  ⇒ one transaction tries to count records in a table
  ⇒ second transactions adds/ deletes a record

- this situation is called the **phantom problem**.

  Solution: operations that ask for “all records” have to lock against insertion/deletion of a qualifying record

  ⇒ locks on tables
  ⇒ index locking and other techniques
Isolation Levels in SQL

The guarantee of serializable executions may carry a heavy price. Performance may be poor because of blocked transactions and deadlocks.

Four **isolation levels** are supported:

- **Level 3:** (Serializability)
  - essentially table-level strict 2PL
- **Level 2:** (Repeatable Read)
  - tuple-level strict 2PL; “phantom tuples” may occur
- **Level 1:** (Cursor Stability)
  - tuple-level exclusive-lock only strict 2PL
  - reading the same object twice: different values
- **Level 0:**
  - neither read nor write locks are acquired
  - transaction may read uncommitted updates
Recovery: Goals and Setting

Two goals:

1. allow transactions to be committed (with a guarantee that the effects are permanent) or aborted (with a guarantee that the effects disappear).

2. allow the database to be recovered to a consistent state in case on HW/power/... failure.

Input: a 2PL, ACA schedule of operations produced by TM.
Output: a schedule of reads/writes/forced writes.
Approaches to Recovery

Two essential approaches:

1. **Shadowing**
   - copy-on-write and merge-on-commit approach
   - poor clustering
   - used in system R, but not in modern systems

2. **Logging**
   - use of LOG (separate disk) to avoid forced writes
   - good utilization of buffers
   - preserves original clusters
A log is a read/append only data structure (a file) ⇒ transactions add log records about what they do

Log records contain several types of information:

- **UNDO information**: old versions of objects that have been modified by a transaction. UNDO information can be used to undo database changes made by a transaction that aborts.

- **REDO information**: new versions of objects that have been modified by a transaction. REDO records can be used to redo the work done by a transaction that commits.

- **BEGIN/COMMIT/ABORT** records are recorded whenever a transaction begins, commits, or aborts.
Example of a LOG

log head → $T_0$,begin
(oldest part) $T_0,X,99,100$
$T_1$,begin
$T_1,Y,199,200$
$T_2$,begin
$T_2,Z,51,50$
$T_1,M,1000,10$
$T_1$,commit
$T_3$,begin
$T_2$,abort
$T_3,Y,200,50$
$T_4$,begin

(newest part) $T_4,M,10,100$

log tail → $T_3$,commit
Write-Ahead Logging (WAL)

How do we make sure the LOG is consistent with the main database?

Write-Ahead Logging (WAL) approach requires:

1. **UNDO rule**: a log record for an update is written to log disk before the corresponding data (page) is written to the main disk (guarantees Atomicity).

2. **REDO rule**: all log records for a transaction are written to log disk before commit (guarantees Durability).
Summary

ACID properties of transactions guarantee correctness of concurrent access to the database and of data storage.

- consistency and isolation based on **serializability**
  - leads to definition of correct **schedulers**
  - responsibility of the **transaction manager**

- durability and atomicity
  - responsibility of the **recovery manager**
  - synchronous writing is too inefficient
    replaced by synchronous writes to a LOG and WAL