Transaction, ACID, concurrency tuning

Viet-Trung Tran
Information systems

Credit: Pei Li – Database management and performance tuning

Outline

- Introduction to transaction
- ACID
- Concurrency control

What is transaction

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- Example: transfer \$50 from account *A* to account *B*
  - 1. $R(A)$
  - 2. $A \leftarrow A - 50$
  - 3. $W(A)$
  - 4. $R(B)$
  - 5. $B \leftarrow B + 50$
  - 6. $W(B)$
- Two main issues:
  - 1. concurrent execution of multiple transactions
  - 2. failures of various kind (e.g., hardware failure, system crash)

ACID Properties

- Database system must guarantee ACID for transactions:
  - **Atomicity**: either all operations of the transaction are executed or none
  - **Consistency**: execution of a transaction in isolation preserves the consistency of the database
  - **Isolation**: although multiple transactions may execute concurrently, each transaction must be unaware of the other concurrent transactions.
  - **Durability**: After a transaction completes successfully, changes to the database persist even in case of system failure.

Atomicity

- Example: transfer \$50 from account *A* to account *B*
  1. $R(A)$
  2. $A \leftarrow A - 50$
  3. $W(A)$
  4. $R(B)$
  5. $B \leftarrow B + 50$
  6. $W(B)$
- **What if failure (hardware or software) after step 3?**
  - money is lost
  - database is inconsistent
- **Atomicity:**
  - either all operations or none
  - updates of partially executed transactions not reflected in database

Consistency

- Example: transfer \$50 from account *A* to account *B*
  1. $R(A)$
  2. $A \leftarrow A - 50$
  3. $W(A)$
  4. $R(B)$
  5. $B \leftarrow B + 50$
  6. $W(B)$
- **Consistency in example**: sum $A + B$ must be unchanged
- **Consistency in general:**
  - explicit integrity constraints (e.g., foreign key)
  - implicit integrity constraints (e.g., sum of all account balances of a
  - bank branch must be equal to branch balance)
- **Transaction:**
  - must see consistent database
  - during transaction inconsistent state allowed
  - after completion database must be consistent again

Isolation – Motivating Example

- Example: transfer \$50 from account *A* to account *B*
  - 1. $R(A)$
  - 2. $A \leftarrow A - 50$
  - 3. $W(A)$
  - 4. $R(B)$
  - 5. $B \leftarrow B + 50$
  - 6. $W(B)$
- **Imagine second transaction *T*2:**
  - $T2 : R(A), R(B), print(A + B)$
  - *T*2 is executed between steps 3 and 4
  - *T*2 sees an inconsistent database and gives wrong result

Isolation

- **Trivial isolation:** run transactions serially
- Isolation for concurrent transactions:
  - For every pair of transactions *Ti* and *Tj*,
  - it appears to *Ti* as if either *Tj* finished execution before *Ti* started
  - or *Tj* started execution after *Ti* finished.
- **Schedule**:
  - specifies the chronological order of a sequence of instructions from various transactions
  - equivalent schedules result in identical databases if they start with identical databases
- **Serializable schedule:**
  - equivalent to some serial schedule
  - serializable schedule of *T*1 and *T*2 is either equivalent to *T*1, *T*2 or *T*2, *T*1

Durability

- When a transaction is done it commits.
- **Example**: transaction commits too early
  - transaction writes *A*, then commits
  - *A* is written to the disk buffer
  - then system crashes
  - value of *A* is lost
- **Durability**: After a transaction has committed, the changes to the database persist even in case of system failure.
- Commit only after all changes are permanent:
  - either written to log file or directly to database
  - database must recover in case of a crash

CONCURRENCY CONTROL

Locks

- A lock is a mechanism to control concurrency on a data item.
- Two types of locks on a data item *A*:
  - **exclusive** – *xL*(*A*): data item *A* can be both read and written
  - **shared** – *sL*(*A*): data item *A* can only be read.
- Lock request are made to concurrency control manager.
- Transaction is blocked until lock is granted.
- **Unlock** *A* – *uL*(*A*): release the lock on a data item *A*

Lock Compatibility

- Lock compatibility matrix:

$T_1 \downarrow \quad T_2 \rightarrow$	shared	exclusive
shared	true	false
exclusive	false	false

- ***T*1 holds shared lock on *A*:**
  - shared lock is granted to *T*2
  - exclusive lock is not granted to *T*2
- ***T*2 holds exclusive lock on *A*:**
  - shared lock is not granted to *T*2
  - exclusive lock is not granted to *T*2
- **Shared locks can be shared by any number of transactions.**

Locking Protocol

- Example transaction *T*2 with locking:
  1. *sL*(*A*), *R*(*A*), *uL*(*A*)
  2. *sL*(*B*), *R*(*B*), *uL*(*B*)
  3. *print*(*A* + *B*)
- ***T*2 uses locking, but is not serializable**
  - *A* and/or *B* could be updated between steps 1 and 2
  - printed sum may be wrong
- **Locking protocol:**
  - set of rules followed by all transactions while requesting/ releasing locks
  - locking protocol restricts the set of possible schedules

Pitfalls of Locking Protocols – Deadlock

- **Example**: two concurrent money transfers
  - $T1$: $R(A)$, $A \leftarrow A + 10$, $R(B)$, $B \leftarrow B - 10$, $W(A)$, $W(B)$
  - $T2$: $R(B)$, $B \leftarrow B + 50$, $R(A)$, $A \leftarrow A - 50$, $W(A)$, $W(B)$
  - possible concurrent scenario with locks:
  - $T1.xL(A)$, $T1.R(A)$, $T2.xL(B)$, $T2.R(B)$, $T2.xL(A)$, $T1.xL(B)$, . . .
  - $T1$ and $T2$ block each other – no progress possible
- **Deadlock**: situation when transactions block each other
- **Handling deadlocks:**
  - one of the transactions must be rolled back (i.e., undone)
  - rolled back transaction releases locks

Pitfalls of Locking Protocols – Starvation

- **Starvation**: transaction continues to wait for lock
- **Examples**:
  - the same transaction is repeatedly rolled back due to deadlocks
  - a transaction continues to wait for an exclusive lock on an item while a
    sequence of other transactions are granted shared locks
- Well-designed concurrency manager avoids starvation.

Protocol: Two-Phase Locking

- Protocol that guarantees serializability.
- Phase 1: growing phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: shrinking phase
  - transaction may release locks
  - transaction may not obtain locks

Two-Phase Locking – Example

- Example: two concurrent money transfers
  - $T1$: $R(A)$, $A \leftarrow A + 10$, $R(B)$, $B \leftarrow B - 10$, $W(A)$, $W(B)$
  - $T2$: $R(A)$, $A \leftarrow A - 50$, $R(B)$, $B \leftarrow B + 50$, $W(A)$, $W(B)$
- Possible two-phase locking schedule:
  - 1. $T1$: $xL(A)$, $xL(B)$, $R(A)$, $R(B)$, $W(A \leftarrow A + 10)$, $uL(A)$
  - 2. $T2$: $xL(A)$, $R(A)$, $xL(B)$ *(wait)*
  - 3. $T1$: $W(B \leftarrow B - 10)$, $uL(B)$
  - 4. $T2$: $R(B)$, $W(A \leftarrow A - 50)$, $W(B \leftarrow B + 50)$, $uL(A)$, $uL(B)$
- Equivalent serial schedule: $T1$, $T2$

CONCURRENCY TUNING

Concurrency Tuning Goals

- **Performance goals:**
  - reduce blocking (one transaction waits for another to release its locks)
  - avoid deadlocks and rollbacks
- **Correctness goals:**
  - serializability: each transaction appears to execute in isolation
  - note: correctness of serial execution must be ensured by the programmer!
- Trade-off between performance and correctness!

Ideal Transaction

- Acquires few locks.
- Favors shared locks over exclusive locks.
  – only exclusive locks create conflicts
- Acquires locks with fine granularity.
  – granularities: table, page, row
  – reduces the scope of each conflict
- Holds locks for a short time.
  – reduce waiting time

Lock Tuning

1. Eliminate unnecessary locks
2. Control granularity of locking
3. Circumvent hot spots
4. Isolation guarantees and snapshot isolation
5. Split long transactions

1. Eliminate Unnecessary Locks

- Lock overhead:
  - memory: store lock control blocks
  - CPU: process lock requests
- Locks not necessary if
  - only one transaction runs at a time, e.g., while loading the database
  - all transactions are read-only, e.g., decision support queries on archival data

2. Control Granularity of Locking

- Locks can be defined at different granularities:
  - row-level locking (also: record-level locking)
  - page-level locking
  - table-level locking
- Fine-grained locking (row-level):
  - good for short online-transactions
  - each transaction accesses only a few records
- Coarse-grained locking (table-level):
  - avoid blocking long transactions
  - avoid deadlocks
    reduced locking overhead

Lock Escalation

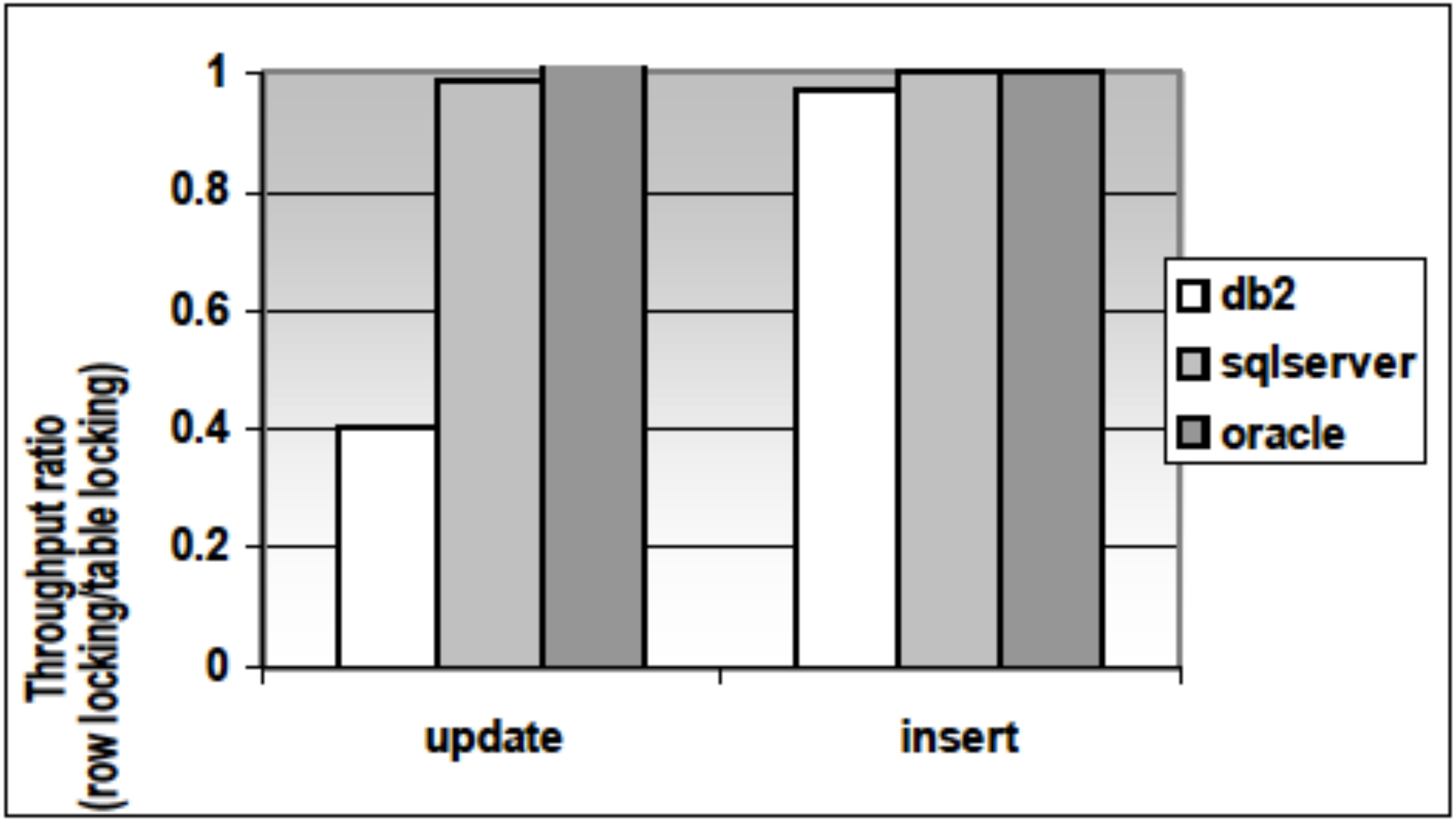
- Lock escalation: (SQL Server and DB2 UDB)
  - automatically upgrades row-level locks into table locks if number of row-level locks reaches predefined threshold
  - lock escalation can lead to deadlock
- Oracle does not implement lock escalation.

Granularity Tuning Parameters

- 1. **Explicit control of the granularity:**
  - within transaction: statement within transaction explicitly requests a table-level lock, shared or exclusive (Oracle, DB2)
  - across transactions: lock granularity is defined for each table; all transactions accessing this table use the same granularity (SQL Server)
- 2. **Escalation point setting:**
  - lock is escalated if number of row-level locks exceeds threshold (escalation point)
  - escalation point can be set by database administrator
  - rule of thumb: high enough to prevent escalation for short online transactions
- 3. **Lock table size:**
  - maximum overall number of locks can be limited
  - if the lock table is full, system will be forced to escalate

Overhead of table vs. row locking

- Experimental setting:
  - accounts(number, branchnum, balance)
  - clustered index on account number
  - 100,000 rows
  - SQL Server 7, DB2 v7.1 and Oracle 8i on Windows 2000
  - lock escalation switched off
- Queries: (no concurrent transactions!)
  - 100,000 updates (1 query)
    - example: update accounts set balance=balance*1.05
  - 100,000 inserts (100,000 queries)
    - example: insert into accounts values(713,15,2296.12)


1
0.8
0.6
0.4
0.2
0
Throughput ratio
(row locking/table locking)
update
insert
db2
sqlserver
oracle

3. Circumvent Hot Spots

- Hot spot: items that are
  - accessed by many transactions
  - updated at least by some transactions
- Circumventing hot spots:
  - access hot spot as late as possible in transaction
  - (reduces waiting time for other transactions since locks are kept to the end of a transactions)
  - use partitioning, e.g., multiple free lists
  - use special database facilities, e.g., latch on counter

Partitioning Example: Distributed Insertions

- **Insert contention:** last table page is bottleneck
  - appending data to heap le (e.g., log les)
  - insert records with sequential keys into table with B+ -tree
- **Solutions:**
  - use clustered hash index
  - if only B+ tree available: use hashed insertion time as key
  - use row locking instead of page locking

Partitioning Example: DDL Statements and Catalog

- **Catalog:** information about tables, e.g., names, column widths
- Data denition language (DDL) statements must access catalog
- Catalog can become hot spot
- **Partition in time:** avoid DDL statements during heavy system activity

4. Weaken Isolation Guarantees

- The main idea is about consistency tuning
- Your homework

5. Chopping Long Transactions

- Algorithms for splitting long transactions
- Your homework