Database Tuning

Concurrency Tuning

Nikolaus Augsten
nikolaus.augsten@sbg.ac.at
Department of Computer Sciences
University of Salzburg

database research group
http://dbresearch.uni-salzburg.at

Sommersemester 2019

Version March 1, 2019

Adapted from "Database Tuning" by Dennis Shasha and Philippe Bonnet.

Outline

What is a Transaction?[1]

- 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)

[1] Slides of section "Introduction to Transactions" are adapted from the slides "Database System Concepts", 6^{th} Ed., Silberschatz, Korth, and Sudarshan

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:
  - $T_2 : R(A), R(B), print(A + B)$
  - 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 T_i and T_j, it appears to T_i as if either T_j finished execution before T_i started or T_j started execution after T_i 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 $T1$ and $T2$ is either equivalent to $T1, T2$ or $T2, T1$

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

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:

$T_1 \downarrow$ $T_2 \rightarrow$	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
  - T_1: $R(A), A \leftarrow A + 10, R(B), B \leftarrow B - 10, W(A), W(B)$
  - T_2: $R(B), B \leftarrow B + 50, R(A), A \leftarrow A - 50, W(A), W(B)$
  - possible concurrent scenario with locks:
    $T_1.xL(A)$, $T_1.R(A)$, $T_2.xL(B)$, $T_2.R(B)$, $T_2.xL(A)$, $T_1.xL(B)$, ...
  - T_1 and T_2 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.

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
  - T_1: $R(A), A \leftarrow A + 10, R(B), B \leftarrow B - 10, W(A), W(B)$
  - T_2: $R(A), A \leftarrow A - 50, R(B), B \leftarrow B + 50, W(A), W(B)$
- Possible two-phase locking schedule:
  1. $T_1 : xL(A), xL(B), R(A), R(B), W(A \leftarrow A + 10), uL(A)$
  2. $T_2 : xL(A), R(A), xL(B)$ *(wait)*
  3. $T_1 : W(B \leftarrow B - 10), uL(B)$
  4. $T_2 : R(B), W(A \leftarrow A - 50), W(B \leftarrow B + 50), uL(A), uL(B)$
- Equivalent serial schedule: T_1, T_2

Outline

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

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.

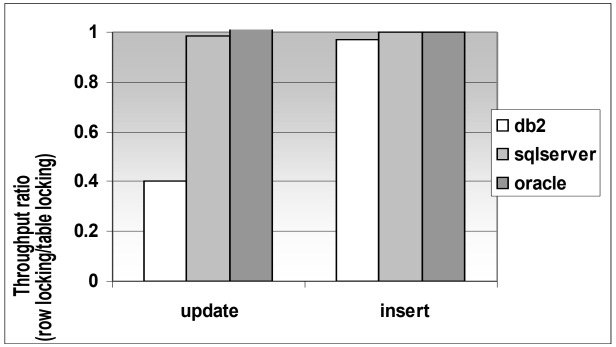
Granularity Tuning Parameters

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)`

Overhead of Table vs. Row Locking

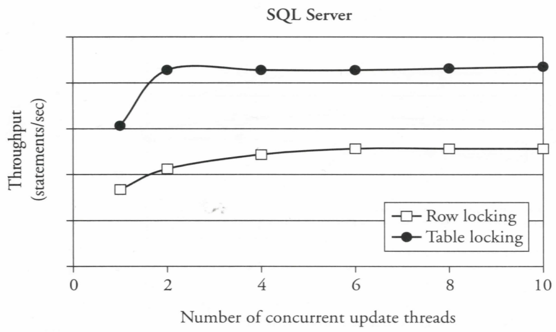


- Row locking (100k rows must be locked) should be more expensive than table locking (1 table must be locked).
- SQL Server, Oracle: recovery overhead (logging changes) hides difference in locking overhead
- DB2: low overhead due to logical logging of updates, difference in locking overhead visible

Experiment: Fine-Grained Locking

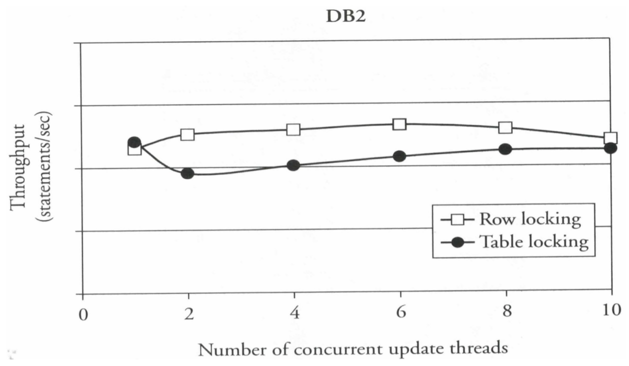
- Experimental setting:
  - table with bank accounts
  - clustered index on account number
  - long transaction (summation of account balances)
  - multiple short transactions (debit/credit transfers)
  - parameter: number of concurrent transactions
  - SQL Server 7, DB2 v7.1 and Oracle 8i on Windows 2000
  - lock escalation switched off

Experiment: Fine-Grained Locking



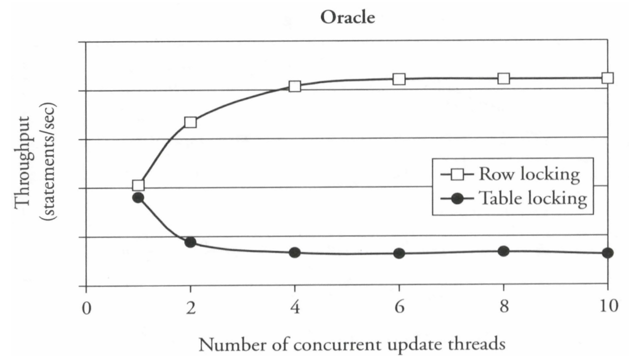
- Serializability with row locking forces key range locks.
- Key range locks are performed in clustered index.
- SQL Server: Clustered index is sparse, thus whole pages are locked.
- Row-level locking only slightly increases concurrency.
- Table-locking prevents rollback for summation query.

Experiment: Fine-Grained Locking



- Row locking slightly better than table locking.
- DB2 automatically selects locking granularity if not forced manually.
  - index scan in this experiment leads to row-level locking
  - table scan would lead to table-level locking

Experiment: Fine-Grained Locking



- Oracle uses snapshot isolation: summation query not in conflict with short transactions.
- Table locking: short transactions must wait.

3. Circumvent Hot Spots

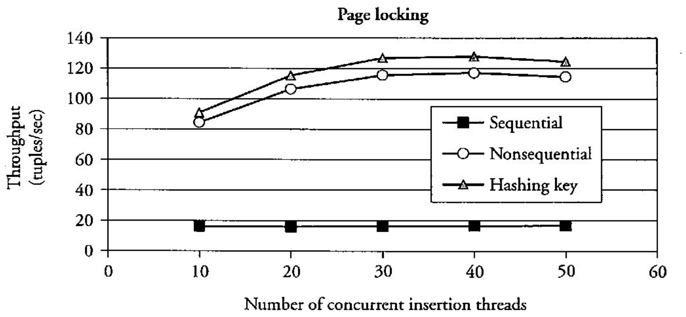
- Hot spot: items that are
  - accessed by many transactions
  - updated at least by some transactions
- Circumventing hot spots:
  - access hot spot as late as possible in transaction (reduces waiting time for other transactions since locks are kept to the end of a transaction[1])
  - use partitioning, e.g., multiple free lists
  - use special database facilities, e.g., latch on counter

[1] In 2-phase locking, the locks need only be held till the end of the growing phase; if the locks are held till the end of the transaction, the resulting schedule is *cascadeless* (in addition to *serializable*), which is desirable.

Partitioning Example: Distributed Insertions

- Insert contention: last table page is bottleneck
  - appending data to heap file (e.g., log files)
  - 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
  - if reads are always table scans: define many insertion points (composite index on random integer $(1..k)$ and key attribute)

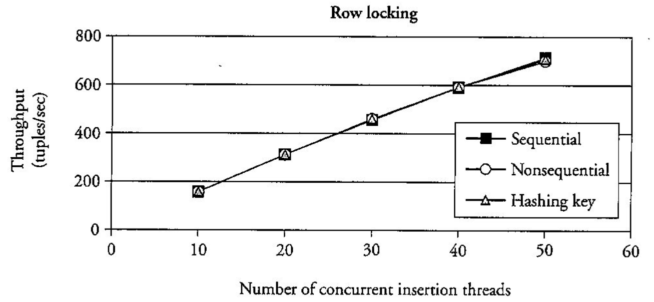
Experiment: Multiple Insertion Points and Page Locking



- Sequential: clustered B^+-tree index and key in insert order
- Non-sequential: clustered B^+-tree, key independent of insert order
- Hashing: composite index on random integer ($1..k$) and key attribute
- Page locking and sequential keys: insert contention!

SQL Server 7 on Windows 2000

Experiment: Multiple Insertion Points and Row Locking



- No insert contention with row locking.

SQL Server 7 on Windows 2000

Partitioning Example: DDL Statements and Catalog

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

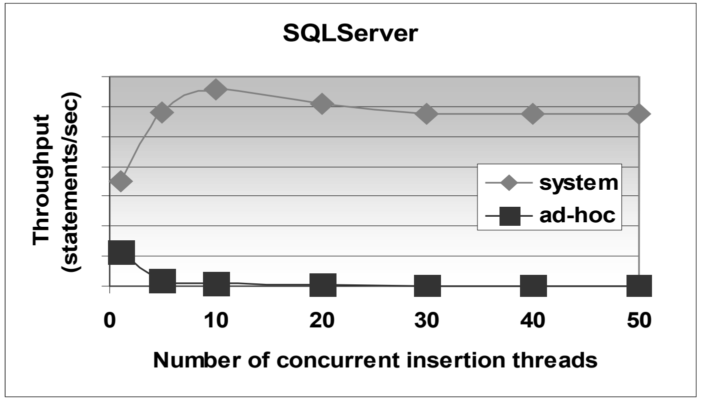
Partitioning Example: Free Lists

- Lock contention on free list:
  - free list: list of unused database buffer pages
  - a thread that needs a free page locks the free list
  - during the lock no other thread can get a free page
- Solution: Logical partitioning
  - create several free lists
  - each free list contains pointers to a portion of free pages
  - a thread that needs a free page randomly selects a list
  - with n free list the load per list is reduced by factor $1/n$

System Facilities: Latch on Counter

- Example: concurrent inserts with unique identifier
  - identifier is created by a counter
  - 2-phase locking: lock on counter is held until transaction ends
  - counter becomes hot spot
- Databases allow to hold a latch on the counter.
  - latch: exclusive lock that is held only during access
  - eliminates bottleneck but may introduce gaps in counter values
- Counter gaps with latches:
  - transaction T_1 increments counter to i
  - transaction T_2 increments counter to $i + 1$
  - if T_1 aborts now, then no data item has identifier i

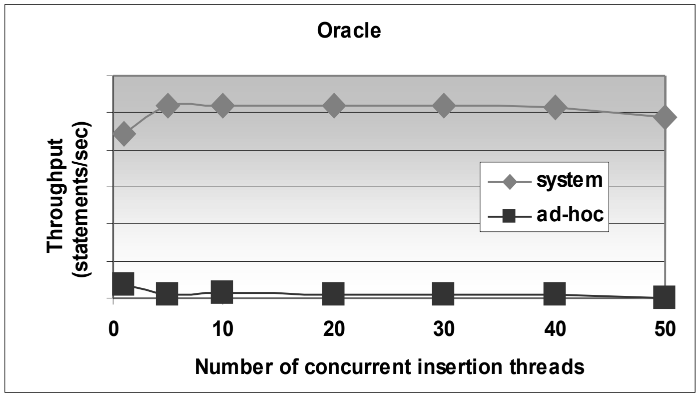
Experiment: Latch vs. Lock on Counter



- System (=latch): use system facility for generating counter values ("identity" in SQL Server)
- Ad hoc (=lock): increment a counter value in an ancillary table

SQL Server 7 on Windows 2000

Experiment: Latch vs. Lock on Counter



- System (=latch): use system facility for generating counter values ("sequence" in Oracle)
- Ad hoc (=lock): increment a counter value in an ancillary table

Oracle 8i EE on Windows 2000

Outline

Undesirable Phenomena of Concurrent Transactions

- Dirty read
  - transaction reads data written by concurrent uncommitted transaction
  - problem: read may return a value that was never in the database because the writing transaction aborted
- Non-repeatable read
  - different reads on the same item within a single transaction give different results (caused by other transactions)
  - e.g., concurrent transactions T_1: $x = R(A)$, $y = R(A)$, $z = y - x$ and T_2: $W(A = 2 * A)$, then z can be either zero or the initial value of A (should be zero!)
- Phantom read
  - repeating the same query later in the transaction gives a different set of result tuples
  - other transactions can insert new tuples during a scan
  - e.g., "Q: get accounts with $balance > 1000$" gives two tuples the first time, then a new account with $balance > 1000$ is inserted by an other transaction; the second time Q gives three tuples

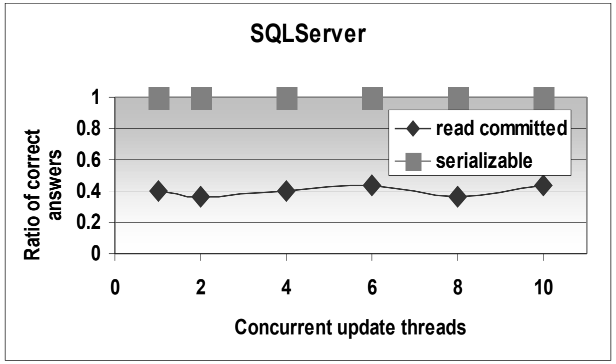
Isolation Guarantees (SQL Standard)

- Read uncommitted: dirty, non-repeatable, phantom
  - read locks released after read; write locks downgraded to read locks after write, downgraded locks released according to 2-phase locking
  - reads may access uncommitted data
  - writes do not overwrite uncommitted data
- Read committed: non-repeatable, phantom
  - read locks released after read, write locks according to (strict) 2-phase locking
  - reads can access only committed data
  - cursor stability: in addition, read is repeatable within single `SELECT`
- Repeatable read: phantom
  - (strict) 2-phase locking, but no range locks
  - phantom reads possible
- Serializable:
  - none of the undesired phenomenas can happen
  - enforced by (strict) 2-phase locking with range locks

Experiment: Read Commit vs. Serializable

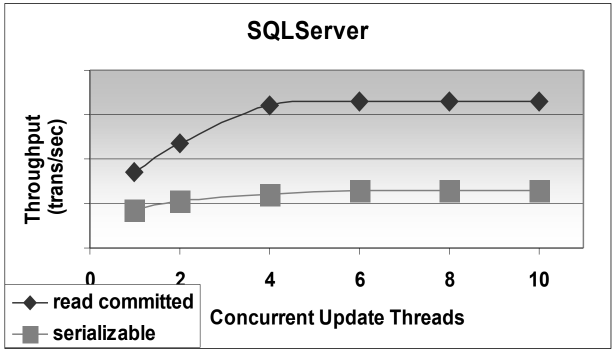
- Experimental setup:
  - T1: summation query: `SELECT SUM(balance) FROM Accounts`
  - T2: money transfers between accounts
  - row level locking
- Parameter: number of concurrent threads
- Measure:
  - percentage of correct answers (over multiple tries)
  - measure throughput

Experiment: Read Commit vs. Serializable



- Read committed allows sum of account balances after debit operation has taken place but before corresponding credit operation is performed – incorrect sum!

Experiment: Read Commit vs. Serializable



- Read committed: faster, but incorrect answers
- Serializable: always correct, but lower throughput

When To Weaken Isolation Guarantees?

- Query does not need exact answer (e.g., statistical queries)
  - example: count all accounts with balance> \$1000.
  - read committed is enough!
- Transactions with human interaction
  - example: flight reservation system
  - price for serializability too high!

Example: Flight Reservation System

- Reservation involves three steps:
  1. retrieve list of available seats
  2. let customer decide
  3. secure seat
- Single transaction:
  - seats are locked while customer decides
  - all other customers are blocked!
- Two transactions: (1) retrieve list, (2) secure seat
  - seat might already be taken when customer wants to secure it
  - more tolerable than blocking all other customers

Snapshot Isolation for Long Reads – The Problem

- Consider the following scenario in a bank:
  - read-only query Q: `SELECT SUM(deposit) FROM Accounts`
  - update transaction T: money transfer between customers A and B
- 2-Phase locking inefficient for long read-only queries:
  - read-only queries hold lock on all read items
  - in our example, T must wait for Q to finish (Q blocks T)
  - deadlocks might occur:
    $T.xL(A)$, $Q.sL(B)$, $Q.sL(A)$ - wait, $T.xL(B)$ - wait
- Read-committed may lead to incorrect results:
  - Before transactions: $A = 50, B = 30$
  - $Q : sL(A), R(A) = 50, uL(A)$
  - $T : xL(A), xL(B), W(A \leftarrow A + 20), W(B \leftarrow B - 20), uL(A), uL(B)$
  - $Q : sL(B), R(B) = 10, uL(B)$
  - sum computed by Q for $A + B$ is 60 (instead of 80)

Snapshot Isolation for Long Reads

- Snapshot isolation: correct read-only queries without locking
  - read-only query Q with snapshot isolation
  - remember old values of all data items that change after Q starts
  - Q sees the values of the data items when Q started
- Example: bank scenario with snapshot isolation
  - Before transactions: $A = 50, B = 30$
  - $Q : R(A) = 50$
  - $T : xL(A), xL(B), W(A \leftarrow A + 20), W(B \leftarrow B - 20), uL(A), uL(B)$
  - $Q : R(B) = 30$ (read old value)
  - sum computed by Q for $A + B$ is 80 as it should be

Concurrency in Oracle

- "Read committed" in Oracle means:
  - non-repeatable and phantom reads are possible at the transaction level, but not within a single SQL statement
  - update conflict: if row is already updated, wait for updating transaction to commit, then update new row version (or ignore row if deleted) – no rollback!
  - possibly inconsistent state: transaction sees updates of other transaction only on the rows that itself updates
- "Serializable" in Oracle means:
  - phenomena: none of the three undesired phenomena can happen
  - update conflict: if two transactions update the same item, the transaction that updates it later must abort – rollback!
  - not serializable: snapshot isolation does not guarantee full serializability (skew writes)
- Similar in PostgreSQL.

Skew Writes: Snapshot Isolation Not Serializable

- Example: $A = 3, B = 17$
  - $T1 : A \leftarrow B$
  - $T2 : B \leftarrow A$
- Serial execution:
  - order $T1, T2$: $A = B = 17$
  - order $T2, T1$: $A = B = 3$
- Snapshot isolation:
  - $T1 : R(B) = 17$
  - $T2 : R(A) = 3$
  - $T1 : W(A \leftarrow 17)$
  - $T2 : W(B \leftarrow 3)$
  - result: $A = 17, B = 3$ (different from serial execution)

Snapshot Isolation

- Advantages: (assuming "serializable" of Oracle)
  - readers do not block writers (as with locking)
  - writers do not block readers (as with locking)
  - writers block writers only if they update the same row
  - performance similar to read committed
  - no dirty, non-repeatable, or phantom reads
- Disadvantages:
  - system must write and hold old versions of modified data (only date modified between start and end of read-only transaction)
  - does not guarantee serializability for read/write transactions
- Implementation example: Oracle 9i
  - no overhead: leverages before-image in rollback segment
  - expiration time of before-images configurable, "snapshot too old" failure if this value is too small

Serializable Snapshot Isolation – Workaround and Solution

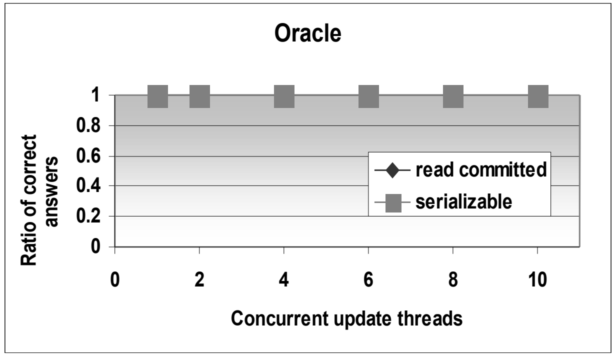
- Workarounds to get true serializability with snapshot isolation:
  - create additional data item that is updated by conflicting transactions (e.g., maintain sum of A and B in our skew write example)
  - use exclusive locks for dangerous reads (e.g., use exclusive lock for reading A and B in our skew write example)
- Problem: requires static analysis of all involved transactions
- Solution: serializable snapshot isolation[2]
  - conflicts are detected by the system
  - conflicting transactions are aborted
  - leads to more aborts, but keeps other advantages of snapshot isolation
- PostgreSQL (starting with version 9.1)
  - REPEATABLE READ is snapshot isolation
  - SERIALIZABLE is serializable snapshot isolation

[2] Michael J. Cahill, Uwe Röhm, Alan David Fekete: Serializable isolation for snapshot databases. SIGMOD Conference 2008: 729-738

Snapshot Isolation – Summary

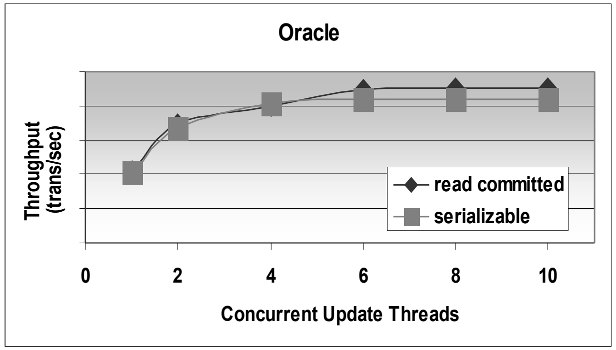
- Considerable performance advantages since reads are never blocked and do not block other transactions.
- Not fully serializable, although no dirty, non-repeatable, or phantom reads.
- Serializable snapshot isolation: fully serializable at the cost of more aborted transactions.

Experiment: Read Commit vs. Serializable



- Summation query with concurrent transfers between bank accounts.
- Oracle snapshot isolation: read-only summation query is not disturbed by concurrent transfer queries
- Summation (read-only) queries always give exact answer.

Experiment: Read Commit vs. Serializable



- Both "read commit" and "serializable" use snapshot isolation.
- "Serializable" rolls back transactions in case of write conflict.
- Summation queries always give exact answer.

Outline

1 Concurrency Tuning
- Introduction to Transactions
- Lock Tuning
- Weaken Isolation Guarantees
- Transaction Chopping

Chopping Long Transactions

- Shorter transactions
  - request less locks (thus they are less likely to be blocked or block an other transaction)
  - require other transactions to wait less for a lock
  - are better for logging
- Transaction chopping:
  - split long transactions into short ones
  - don't scarify correctness

Terminology

- Transaction: sequence of disc accesses (read/write)
- Piece of transaction: consecutive subsequence of database access.
  - example transaction $T : R(A), R(B), W(A)$
  - $R(A)$ and $R(A), R(B)$ are pieces of T
  - $R(A), W(A)$ is not a piece of T (not consecutive)
- Chopping: partitioning transaction it into pieces.
  - example transaction $T : R(A), R(B), W(A)$
  - $T_1 : R(A), R(B)$ and $T_2 : W(A)$ is a chopping of T

Split Long Transactions – Example 1

- Bank with accounts and branches:
  - each account is assigned to exactly one branch
  - branch balance is sum of accounts in that branch
  - customers can take out cash during day
- Transactions over night:
  - update transaction: reflect daily withdrawals in database
  - balance checks: customers ask for account balance (read-only)
- Update transaction T_{blob}
  - updates all account balances to reflect daily withdrawals
  - updates the respective branch balances
- Problem: balance checks are blocked by T_{blob} and take too long

Split Long Transactions – Example 1

- Solution: split update transactions T_{blob} into many small transactions
- Variant 1: each account update is one transaction which
  - updates one account
  - updates the respective branch balance
- Variant 2: each account update consists of two transactions
  - T_1 : update account
  - T_2 : update branch balance
- Note: isolation does not imply consistency
  - both variants maintain serializability (isolation)
  - variant 2: consistency (sum of accounts equal branch balance) compromised if only one of T_1 or T_2 commits.

Split Long Transactions – Example 2

- Bank scenario as in Example 1.
- Transactions:
  - update transaction: each transaction updates one account and the respective branch balance (variant 1 in Example 1)
  - balance checks: customers ask for account balance (read-only)
  - consistency (T'): compute account sum for each branch and compare to branch balance
- Splitting: T' can be split into transactions for each individual branch
- Serializability maintained:
  - consistency checks on different branches share no data item
  - updates leave database in consistent state for T'
- Note: update transaction can not be further split (variant 2)!
- Lessons learned:
  - sometimes transactions can be split without sacrificing serializability
  - adding new transaction to setting may invalidate all previous chopping

Formal Chopping Approach

- Assumptions: when can the chopping be applied?
- Execution rules: how must chopped transactions be executed?
- Chopping graph: which chopping is correct?

Assumptions for Transaction Chopping

1. Transactions: All transactions that run in an interval are known.
2. Rollbacks: It is known where in the transaction rollbacks are called.
3. Failure: In case of failure it is possible to determine which transactions completed and which did not.
4. Variables: The transaction code that modifies a program variable x must be reentrant, i.e., if the transaction aborts due to a concurrency conflict and then executes properly, x is left in a consistent state.

Execution Rules

1. Execution order: The execution of pieces obeys the order given by the transaction.
2. Lock conflict: If a piece is aborted due to a lock conflict, then it will be resubmitted until it commits.
3. Rollback: If a piece is aborted due to a rollback, then no other piece for that transaction will be executed.

The Transaction Chopping Problem

- Given: Set $A = \{T_1, T_2, \ldots, T_n\}$ of (possibly) concurrent transactions.
- Goal: Find a chopping B of the transactions in A such that any serializable execution of the transactions in B (following the execution rules) is equivalent so some serial execution of the transaction in A. Such a chopping is said to be correct.
- Note: The "serializable" execution of B may be concurrent, following a protocol for serializability.

Chopping Graph

- We represent a specific chopping of transactions as a graph.
- Chopping graph: undirected graph with two types of edges.
  - nodes: each piece in the chopping is a node
  - C-edges: edge between any two conflicting pieces
  - S-edges: edge between any two sibling pieces
- Conflicting pieces: two pieces p and p' conflict iff
  - p and p' are pieces of different original transactions
  - both p and p' access a data item x and at least one modifies it
- Sibling pieces: two pieces p and p' are siblings iff
  - p and p' are neighboring pieces of the same original transactions

Chopping Graph – Example

- Notation: chopping of possibly concurrent transactions.
  - original transactions are denoted as $T_1, T_2, \ldots$
  - chopping T_i results in pieces $T_{i1}, T_{i2}, \ldots$
- Example transactions: ($T_1 : R(x), R(y), W(y)$ is split into T_{11}, T_{12})
  - $T_{11} : R(x)$
  - $T_{12} : R(y), W(y)$
  - $T_2 : R(x), W(x)$
  - $T_3 : R(y), W(y)$
- Conflict edge between nodes
  - T_{11} and T_2 (conflict on x)
  - T_{12} and T_3 (conflict on y)
- Sibling edge between nodes
  - T_{11} and T_{22} (same original transaction T_1)

Rollback Safe

- Motivation: Transaction T is chopped into T_1 and T_2.
  - T_1 executes and commits
  - T_2 contains a rollback statement and rolls back
  - T_1 is already committed and will not roll back
  - in original transaction T rollback would also undo effect of piece T_1!
- A chopping of transaction T is rollback save if
  - T has no rollback statements or
  - all rollback statements are in the first piece of the chopping

Correct Chopping

Theorem (Correct Chopping)

A chopping is correct if it is rollback save and its chopping graph contains no SC-cycles.

- Chopping of previous example is correct (no SC-cycles, no rollbacks)
- If a chopping is not correct, then any further chopping of any of the transactions will not render it correct.
- If two pieces of transaction T are in an SC-cycle as a result of chopping T, then they will be in a cycle even if no other transactions (different from T) are chopped.

Private Chopping

- Private chopping: Given transactions $T_1, T_2, \ldots, T_n$. $T_{i1}, T_{i2}, \ldots, T_{ik}$ is a private chopping of T_i if
  - there is no SC-cycle in the graph with the nodes $\{T_1, \ldots, T_{i1}, \ldots, T_{ik}, \ldots, T_n\}$
  - T_i is rollback save
- Private chopping rule: The chopping that consists of $private(T_1), private(T_2), \ldots, private(T_n)$ is correct.
- Implication:
  - each transaction T_i can be chopped in isolation, resulting in $private(T_i)$
  - overall chopping is union of private choppings

Chopping Algorithm

1. Draw an S-edge between the R/W operations of a single transaction.
2. For each data item x produce a write list, i.e., a list of transactions that write this data item.
3. For each $R(x)$ or $W(x)$ in all transactions:
   (a) look up the conflicting transactions in the write list of x
   (b) draw a C-edge to the respective conflicting operations
4. Remove all S-edges that are involved in an SC-cycle.

Chopping Algorithm – Example

- Transactions: $(Rx = R(x), Wx = W(x))$
  - $T_1 : Rx, Wx, Ry, Wy$
  - $T_2 : Rx, Wx$
  - $T_3 : Ry, Rz, Wy$
- Write lists: $x{:}\,T_1, T_2;\ y{:}\,T_1, T_3;\ z{:}\,\emptyset$
- C-edges:
  - T_1: $Rx - T_2.Wx$, $Wx - T_2.Wx$, $Ry - T_3.Wy$, $Wy - T_3.Wy$
  - T_2: $Rx - T_1.Wx$ ($Wx - T_1.Wx$: see T_1)
  - T_3: $Ry - T_1.Wy$ ($Wy - T_1.Wy$: see T_1)
- Remove S-edges: T_1: $Rx - Wx$, $Ry - Wy$; T_2: $Rx - Wx$; T_3: $Ry - Rz, Rz - Wy$
- Final chopping:
  - $T_{11} : Rx, Wx$; $T_{12} : Ry, Wy$
  - $T_2 : Rx, Wx$
  - $T_3 : Ry, Rz, Wy$

Reordering Transactions

- Commutative operations:
  - changing the order does not change the semantics of the program
  - example: $R(y), R(z), W(y \leftarrow y + z)$ and $R(z), R(y), W(y \leftarrow y + z)$ do the same thing
- Transaction chopping:
  - changing the order of commutative operations may lead to better chopping
  - responsibility of the programmer to verify that operations are commutative!
- Example: consider $T_3 : Ry, Rz, Wy$ of the previous example
  - assume T_3 computes $y + z$ and stores the sum in y
  - then Ry and Rz are commutative and can be swapped
  - $T_3' : Rz, Ry, Wy$ can be chopped: $T_{31}' : Rz$, $T_{32}' : Ry, Wy$