Database Tuning

Concurrency Tuning

Nikolaus Augsten

University of Salzburg
Department of Computer Science
Database Group

Unit 9 – WS 2013/2014

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

Outline

1. Concurrency Tuning
   - Weaken Isolation Guarantees
   - Transaction Chopping

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

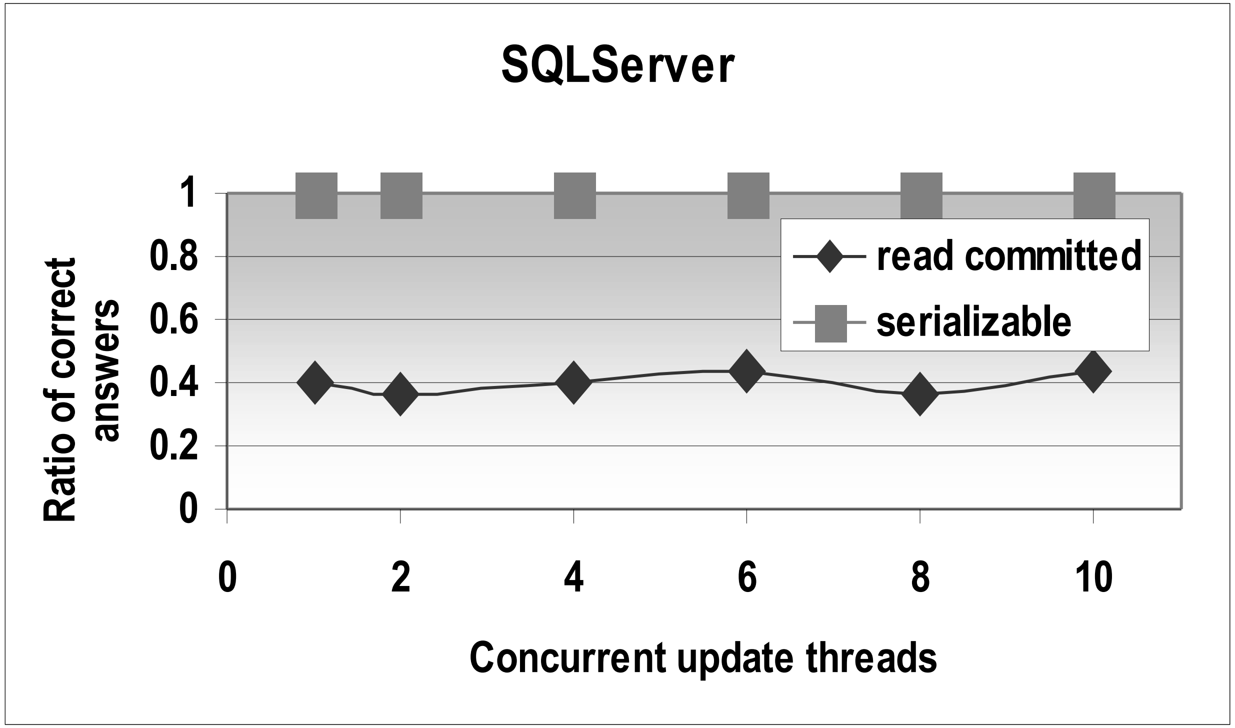
Isolation Guarantees (SQL Standard)

- 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 2-phase locking
  - reads can access only committed data
  - cursor stability: in addition, read is repeatable within single `SELECT`
- Repeatable read: phantom
  - 2-phase locking, but no range locks
  - phantom reads possible
- Serializable:
  - none of the undesired phenomenas can happen
  - enforced by 2-phase locking with range locks

Experiment: Read Commit vs. Serializable

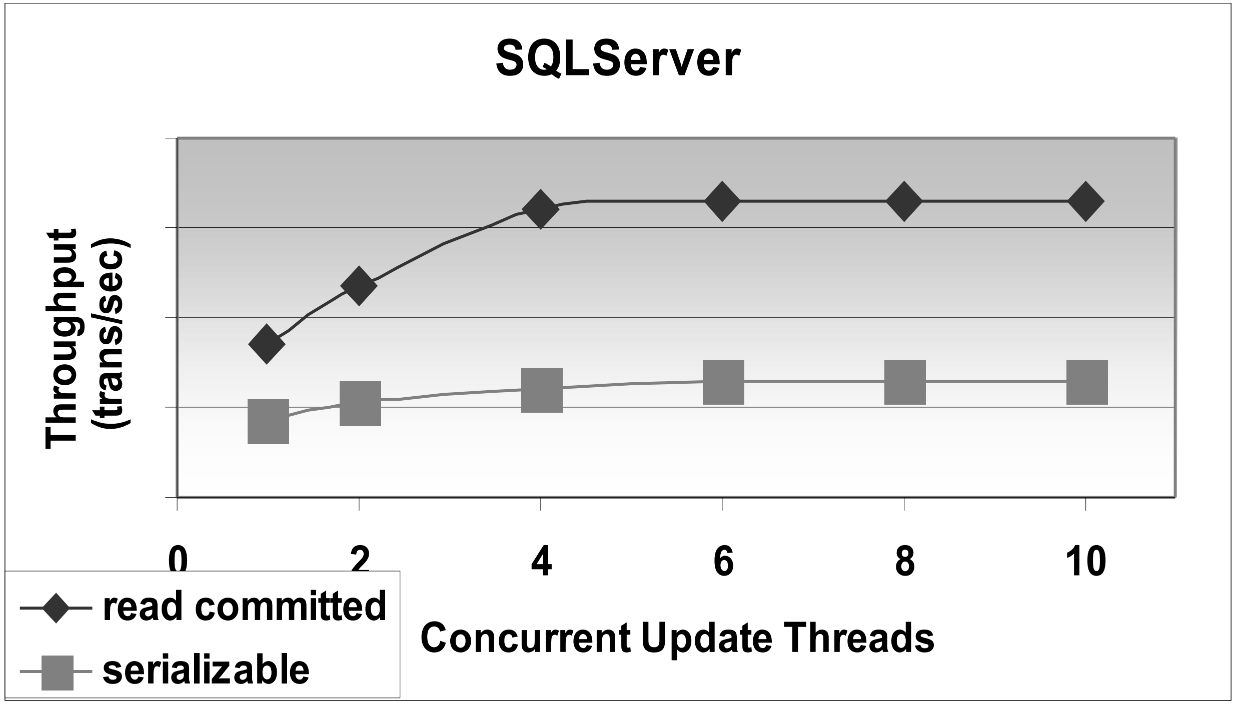
- Experimental setup:
  - T1: summation query: `SELECT SUM(balance) FROM Accounts`
  - T2: money transfers between accounts
  - row level locking
- Parameter: number of concurrent threads
- Measure:
  - percentage of correct answers (over multiple tries)
  - measure throughput

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)

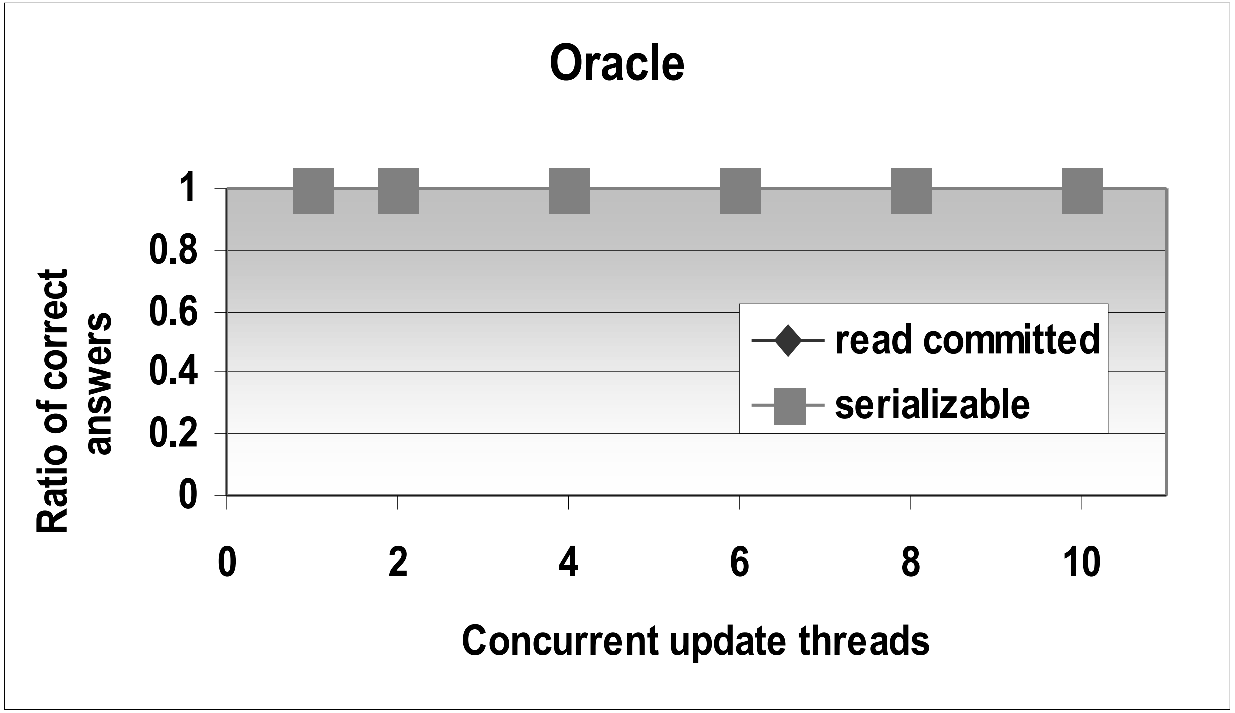
Snapshot Isolation

- 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

Snapshot Isolation – Summary

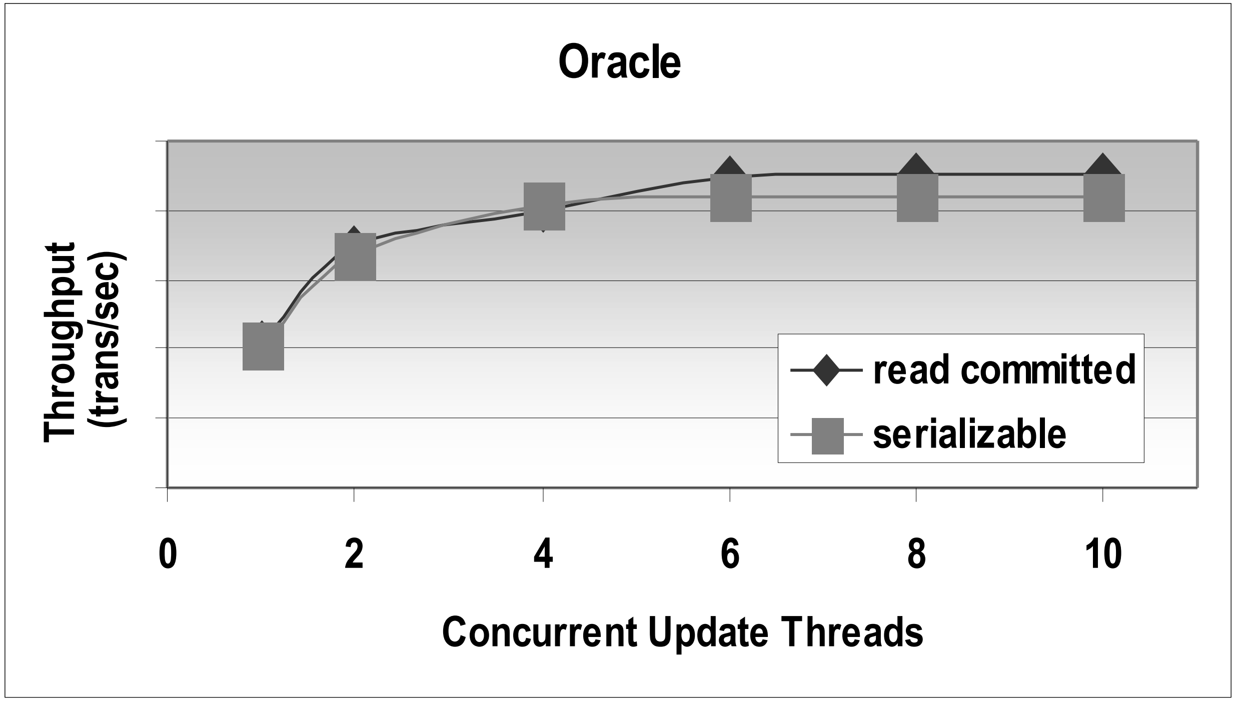
- Considerable performance advantages since reads are never blocked and do not block other transactions.
- Not fully serializable, although no dirty, non-repeatable, or phantom reads.

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
   - 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: $T_{31}' : Rz$, $T_{32}' : Ry, Wy$