Tuning Concurrency

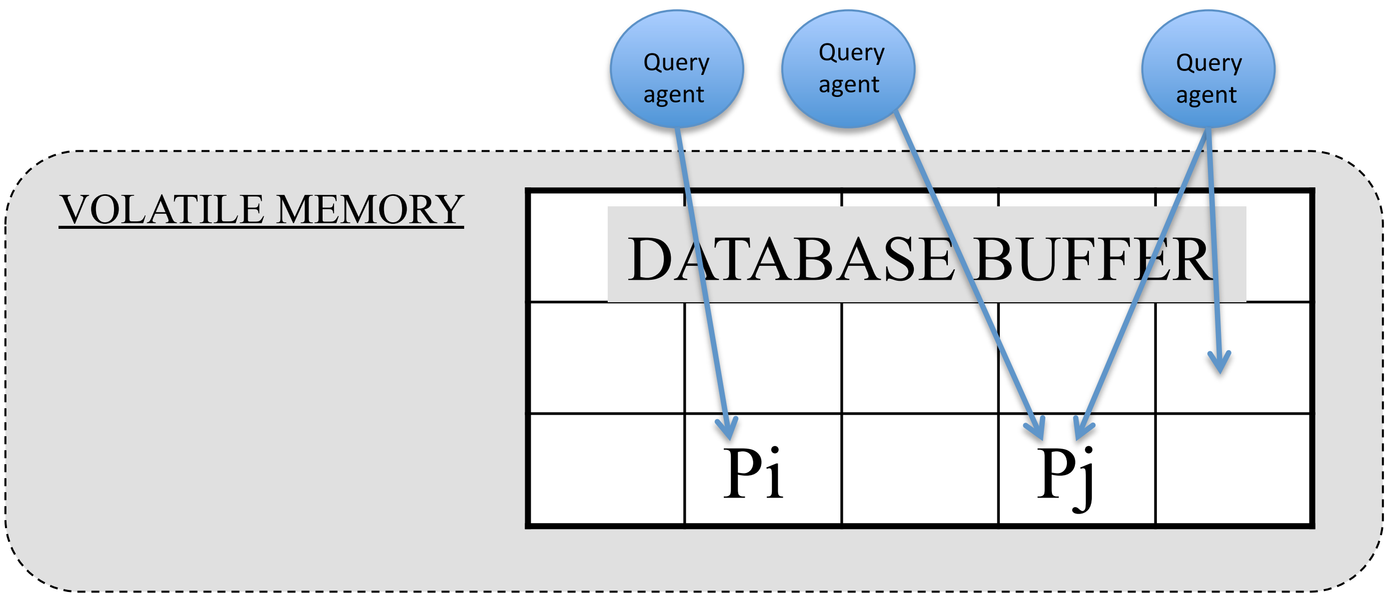
@ Dennis Shasha and Philippe Bonnet, 2013

Outline

- Correctness and Performance
- Concurrency control principles
  - Serialization graph
  - Read consistency
  - 2-phase locking
  - Lock granularity
    - The Phantom Problem
    - Next-key locking
  - Lock implementation
- Lock Tuning
  - Isolation levels
  - Transaction chopping

Context

How can the DBMS handle concurrent transactions efficiently?



Transactions Execution

- Each transaction is a sequence of operations
  - Read (select)
  - Write (update, insert, delete)
- Execution model
  - Batch: transactions are submitted and executed later on
  - Interactive: transactions are executed as they are submitted
    - The DBMS contains a scheduler that interleaves the execution of all transaction operations as they are submitted

Scheduling

- How can the schedules interleave operations so that isolation is preserved?
  - Each transaction should execute as if it was the only one in the system
    - This is the definition of a <u>correct</u> schedule
  - A schedule is correct if it is equivalent to a serial schedule
    - Two schedule are equivalent if all they operations commute
    - Most operations commute
      - Two operations on different data items
      - Two read operations

Scheduling

- Operations do not commute, if they conflict
  - Two transactions conflict when:
    - They contain operations applied on the same data
    - One of these operations is a write
  - Example conflicts:
    - W-W: lost update
    - W-R: inconsistent read
    - R-W-R: unrepeatable read
    - I-R/W: Phantom

Scheduling

- Locking is used to make conflicts explicit for the scheduler
  - Locking protocol to determine when transaction acquire/release locks
    - In case of conflict, a transaction is delayed
    - Direct impact on performance (throughput and latency)
  - **Trade-off between correctness and performance**

Serialization Graph

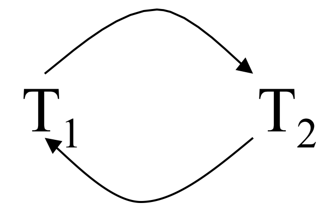
For a given schedule:

- Vertices are defined for all transactions
- Edges are defined for all conflicting transactions
  - There is an edge from Ti to Tj iff Ti contains an operation pi that conflicts with an operation pj, and Pi precedes pj in S
- **<u>Theorem</u>**: A schedule is conflict serializable if and only if its serialization graph has no cycles

Example#1

- Consider the nonserializable schedule S

  $r_1(x)\ w_2(x)\ r_2(y)\ w_1(y)$

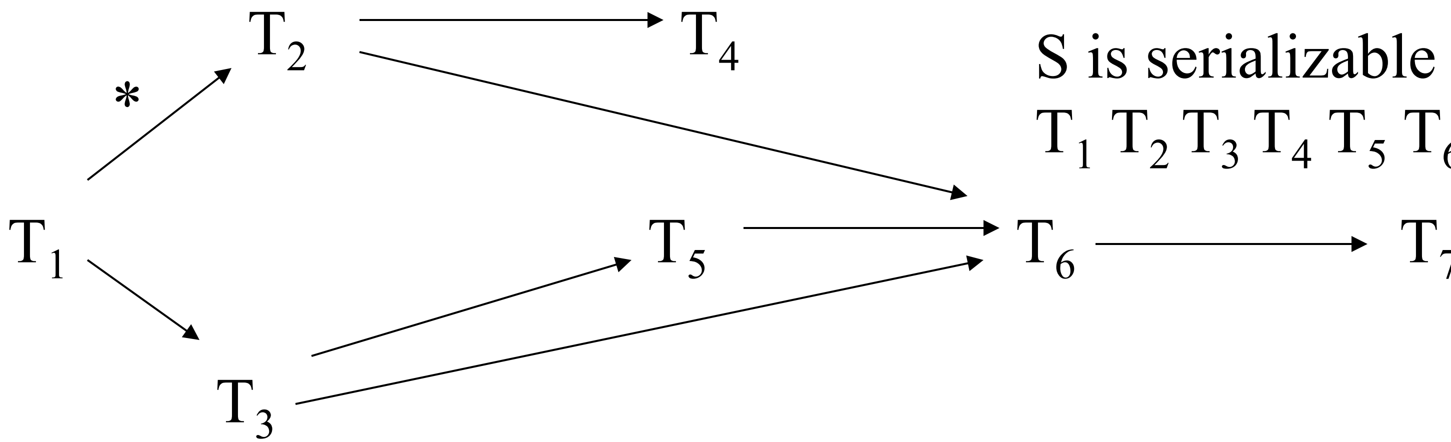


- Intuition
  - T1 and T2 conflict on x
    - T1 reads x before T2 writes it
      - Schedule S must be equivalent to a serial schedule where T1 precedes T2
  - T1 and T2 conflict on y
    - T2 writes y before T1 writes it
      - Schedule S must be equivalent to a serial schedule where T2 precedes T1
  - S is not equivalent to <u>any serializable schedule</u>
  - S does not guarantee isolation

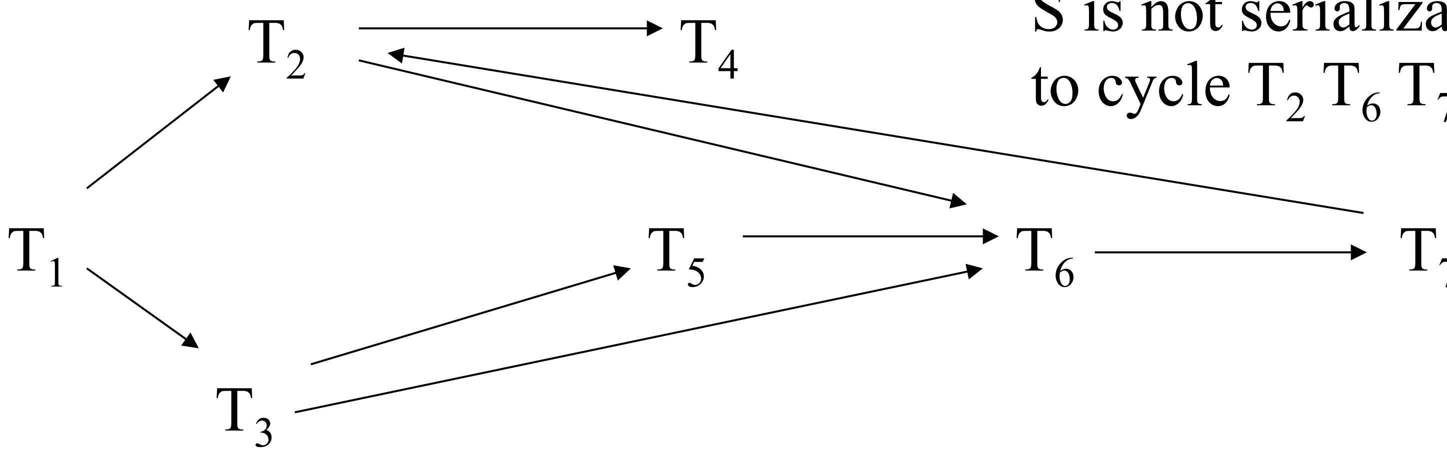
Example #2

Conflict (*)

S: … $p_{1,i}$, …, $p_{2,j}$, …

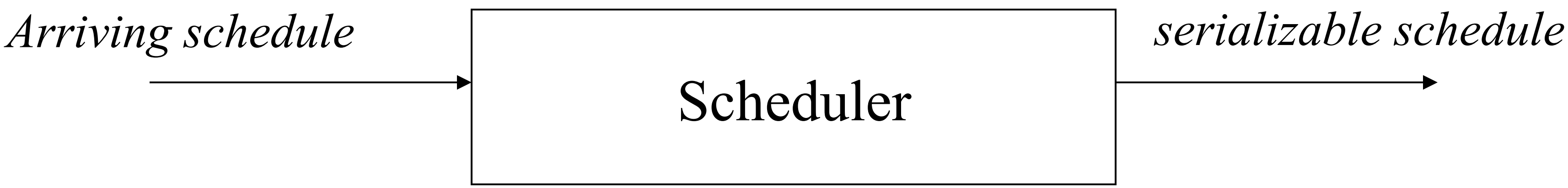


S is serializable in order $T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7$



S is not serializable due to cycle $T_2\ T_6\ T_7\ T_2$

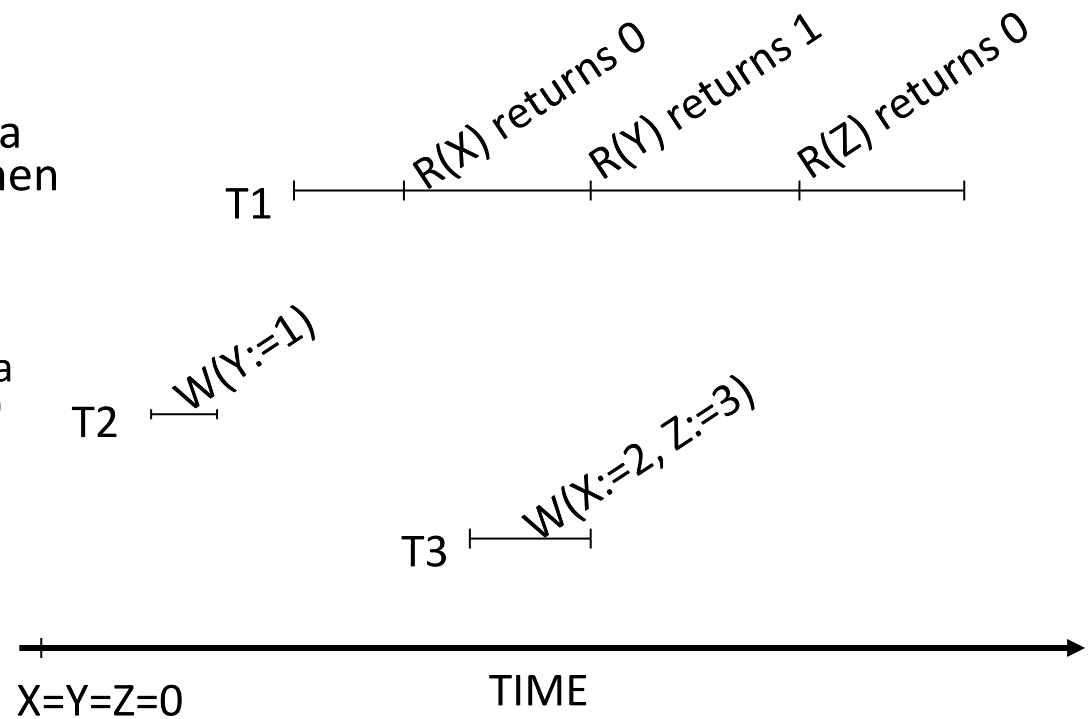
Concurrency Control



- Concurrency control cannot see entire schedule:
  - It sees one request at a time and must decide whether to allow it to be serviced
- Two (complementary) strategies:
  1. Avoid (some) conflicts – **Read consistency**
     - Restrict conflicts to W-W
     - Read and write executed on distinct copies of the data
  2. Make conflicts explicit - **Locking**
     - Delay operations that are conflicting with non committed operations

Read Consistency

- Also called snapshot isolation
- Each transaction executes against the version of the data items that was committed when the transaction started:
  - No locks for read
  - Locks for writes
  - Costs space (old copy of data must be kept – undo record)
- Almost serializable level:
  - T1: x:=y
  - T2: y:= x
  - Initially x=3 and y =17
  - Serial execution: x,y=17 or x,y=3
  - Snapshot isolation: x=17, y=3 if both transactions start at the same time.



Locking

- A transaction can read a database item if it holds a read (shared - S) lock on the item
- It can read *or* update the item if it holds a write (exclusive - X) lock
- If the transaction does not already hold the required lock, a lock request is automatically made as part of the (read or write) request

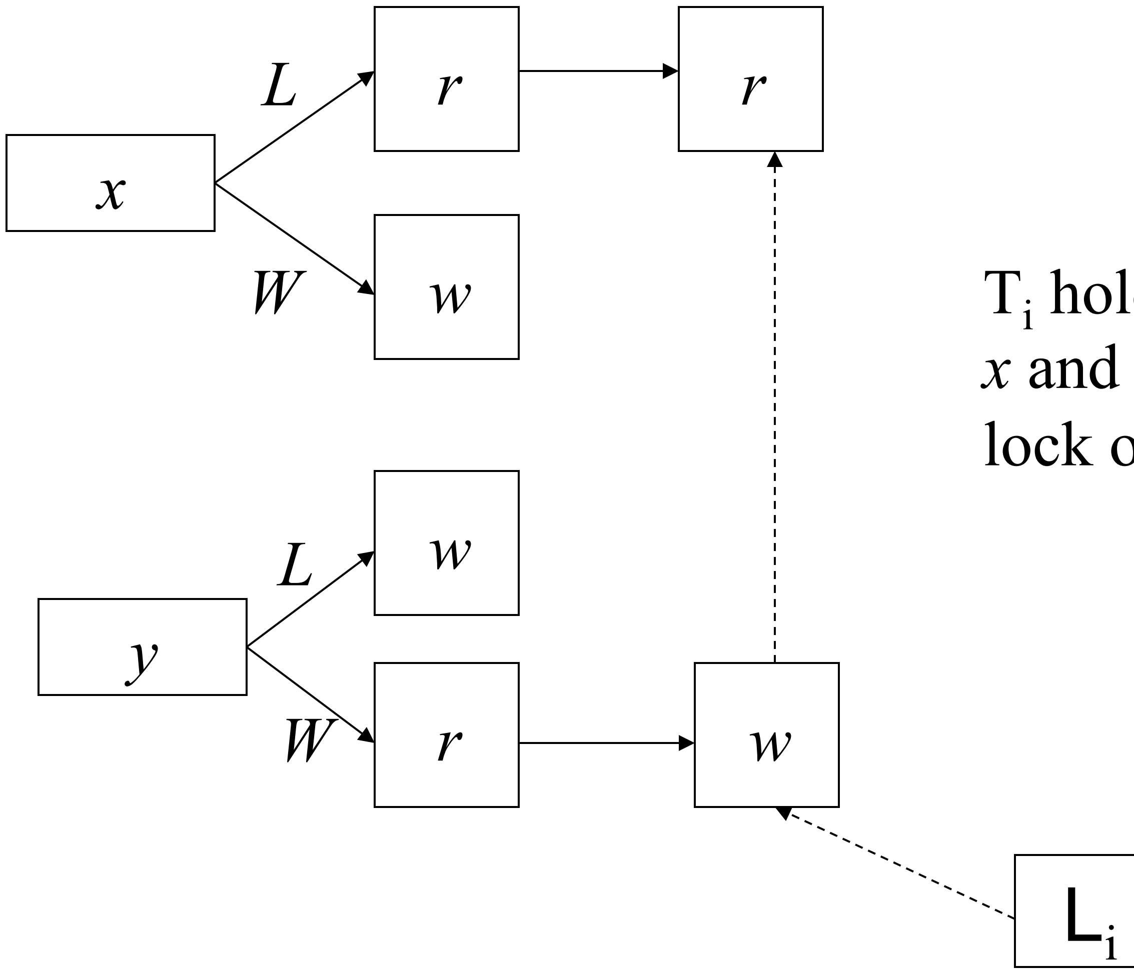
Locking

- **Request for read lock** on an item is granted if
  - **no transaction currently holds write** lock on the item
    - Cannot read an item written by an active transaction
- **Request for write lock** granted if
  - **no transaction currently holds any lock** on item
    - Cannot write an item read/written by an active transaction
- **Transaction is delayed** if request cannot be granted

Locking Implementation

- Associate a *lock set, L(x),* and a *wait set, W(x),* with each active database item, *x*
  - *L(x)* contains an entry for each granted lock on *x*
  - *W(x)* contains an entry for each pending request on *x*
  - When an entry is removed from *L(x)* (due to transaction termination), promote (non-conflicting) entries from *W(x)* using some scheduling policy (*e.g.,* FCFS)
- Associate a lock list, L_i , with each transaction, T_i.
  - L_i links T_i's elements in all lock and wait sets
  - Used to release locks on termination

Locking Implementation



T_i holds an *r* lock on *x* and waits for a *w* lock on *y*

Latches and Locks

- Locks are used for concurrency control
  - Requests for locks are queued
    - Priority queue
  - Lock data structure
    - Locking mode (S, lock granularity, transaction id.
    - Lock table
- Latches are used for mutual exclusion
  - Requests for latch succeeds or fails
    - Active wait (spinning) on latches on multiple CPU.
  - Single location in memory
    - Test and set for latch manipulation

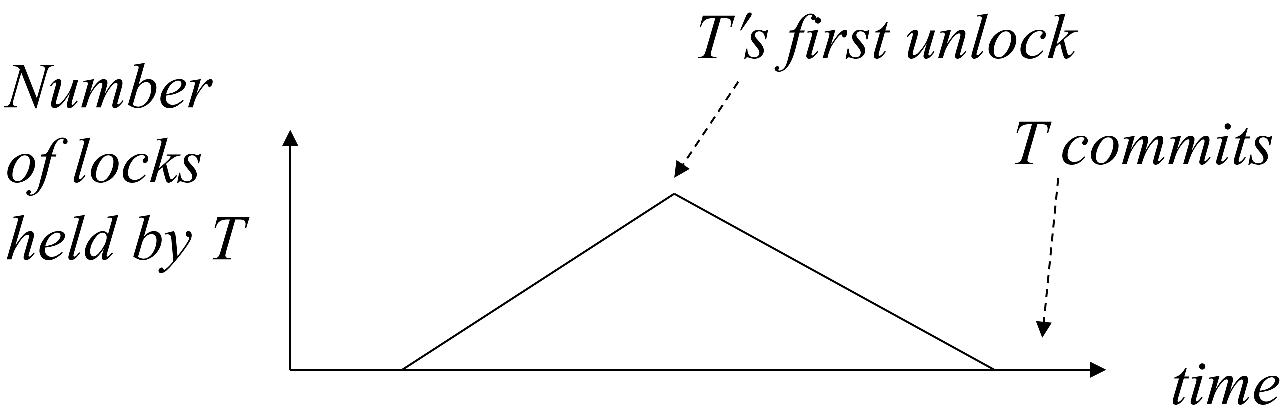
Locking Protocol

- Locking does not guarantee serializability. It depends on the locking protocol!
  - Example protocol:
    - Release lock on item when finished accessing the item
    - It can lead to non-serializable schedules

$$\begin{array}{llll} T_1: & l(x)\ r(x)\ u(x) & & l(y)\ r(y)\ u(y) \\ T_2: & & l(x)\ l(y)\ w(x)\ w(y)\ \underbrace{u(x)\ u(y)}_{commit} & \end{array}$$

Two-Phase Locking

- Transaction does not release a lock until it has all the locks it will ever require.
- Transaction has a locking phase followed by an unlocking phase



Deadlock

- **Problem**: Conflicts that cause transactions to wait can cause deadlocks

  Example:

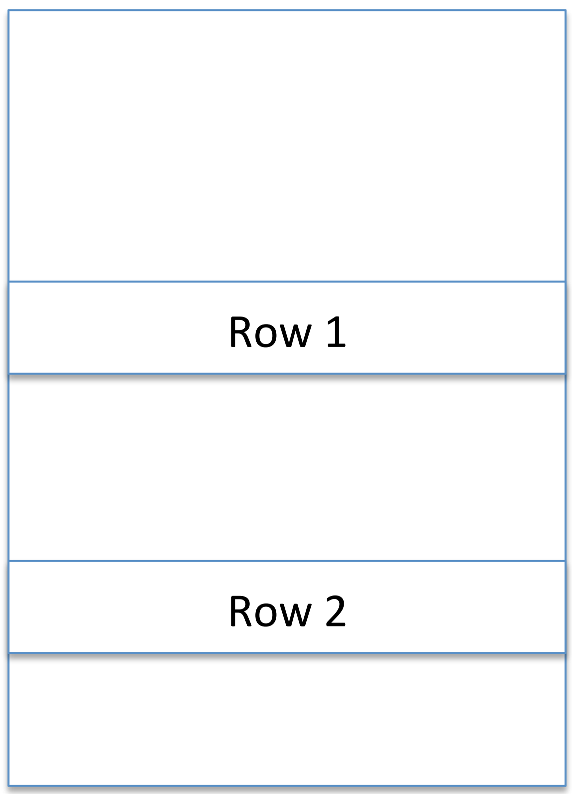
  T1: w(x) r(y)

  T2: w(y) r(x)

- **Wait-for graph:** For a schedule S
  - Vertices for each transaction
  - Edges when a transaction wait for another
  - Deadlock in case of cycle in a wait-for graph

- **Solution**:
  1. Detect deadlock: Use wait-for graph to detect cycle when a request is delayed, pick one of the transaction in the cycle and abort it
  2. Assume a deadlock when a transaction waits longer than some time-out period

Lock Granularity



Table

- Table lock
  - T1.Read(row1) **<u>conflicts</u>** with T2.write(row2)
  - The scope of conflicts is artificially large
- Row lock
  - T1.read(row1) **<u>does not conflict</u>** with T2.write(row2)
  - T1.read(row1) conflicts with T2.write(row1)
  - The scope of conflict is narrow

Lock Granularity

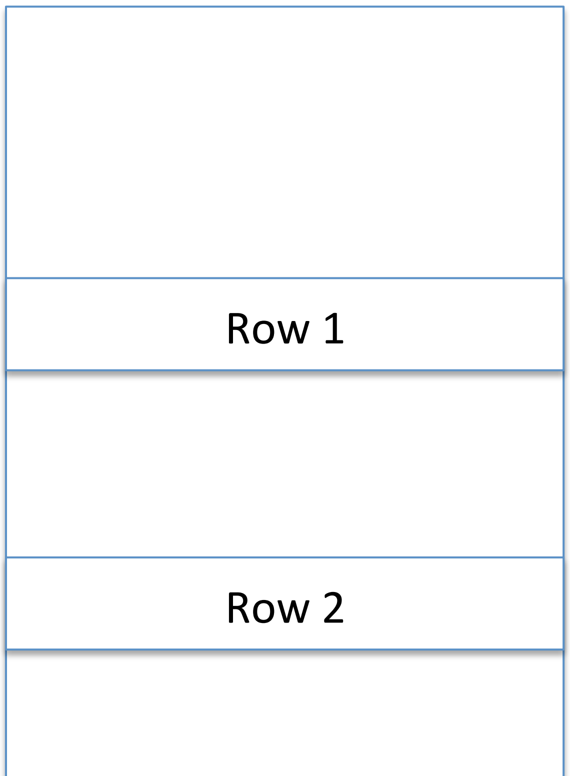
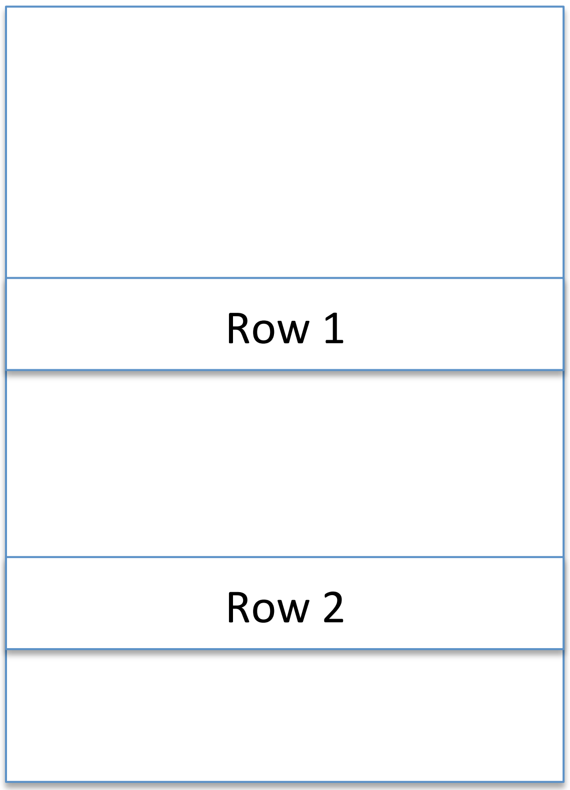


Table T

- Case:
  - transaction T1 is granted a table lock in mode shared on Table T
  - T2 requests a row lock in mode exclusive on Row1
  - Should T2's lock request be granted?

Lock Granularity



Table

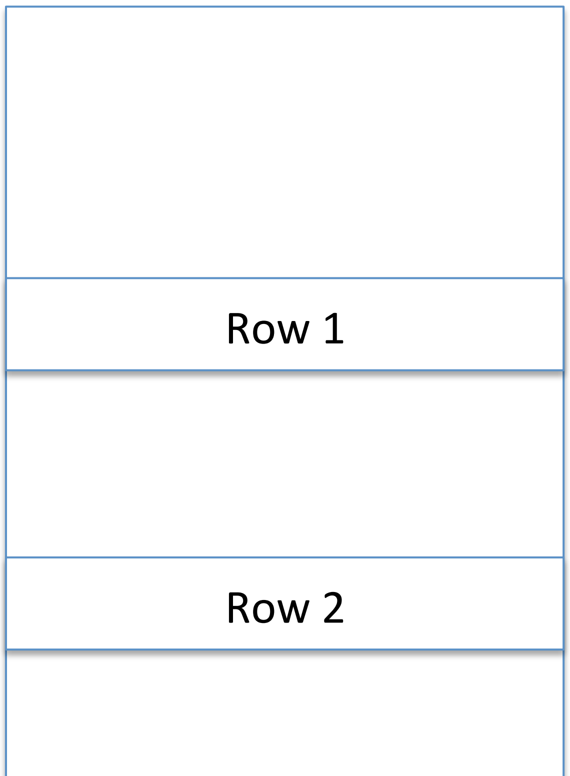
- A row lock request is preceded by an intention lock request at the table level:
  - S (resp. X) lock on Row tied to IS (resp. IX) lock on Table
- Lock compatibility table

Lock held

Lock requested

	S	IS	X	IX
S	X	X		
IS	X	X		x
X				
IX		x		x

Lock Escalation



Table

- The DBMS might choose to convert many row locks into a table lock
- DB2 allows lock escalation; Oracle does not
- Problem: lock escalation might cause the DBMS to cause a deadlock, e.g.,
  - T1 holds a row lock on row1
  - T2 holds a row lock on a million rows in the Table include row2
  - T1 requests a lock on row2
  - The DBMS decides to escalate T2 row locks to a table lock

Phantom Problem

Table R

	E#	Name	age
[row1]	01	Smith	35
[row2]	02	Jones	28

- T1:

  insert (03, Povl, 32) into R

- T2:

  Select max(age) from R
  where 30 < age < 40

<u>Snapshot isolation not in effect</u>

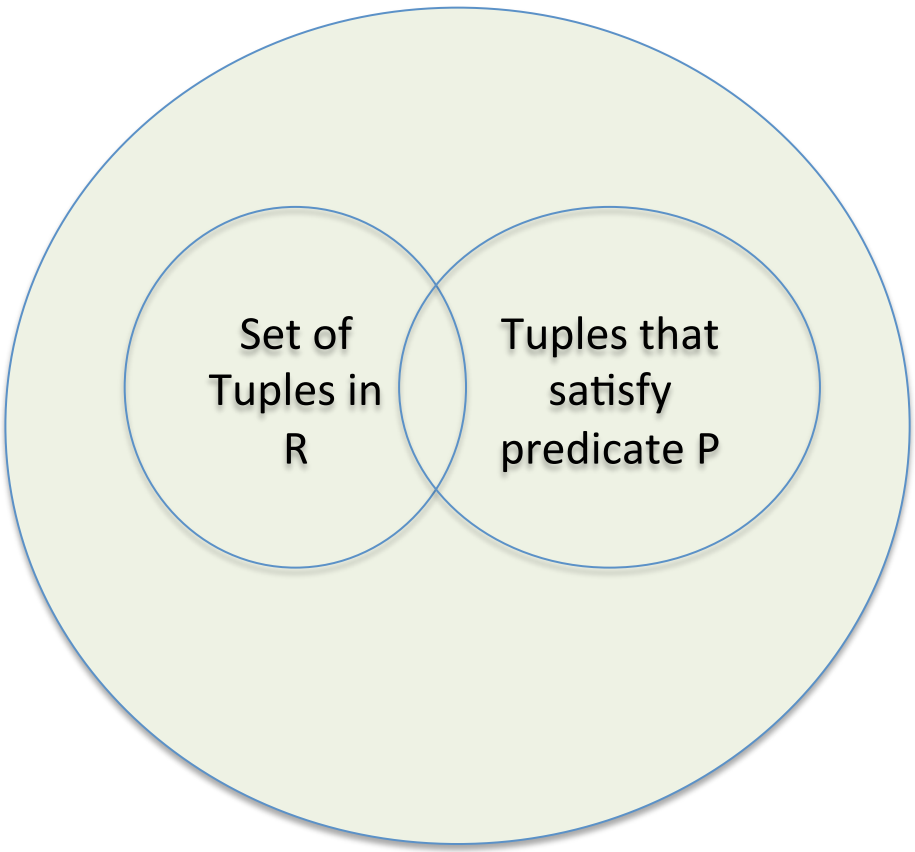
Time

T1: insert(03,Povl, 32), commit

T2: l(row1,s)l(row2,s) read all rows, compute avg, u(row1),u(row2), commit

Row locks do not prevent concurrent insertions as they only protect existing rows.

Solution to Phantom Problem



Set of all tuples that can be inserted in R

- Solution#1:
  - Table locking (mode X)
  - No insertion is allowed in the table
  - Problem: too coarse if predicate is used in transactions
- Solution #2:
  - Predicate locking – avoid inserting tuples that satisfy a given predicate
    - E.g., 30 < age < 40
  - Problem: very complex to implement
- Solution #3:
  - Next key locking (NS)
  - See index tuning

Locking in SQL Server

syslockinfo

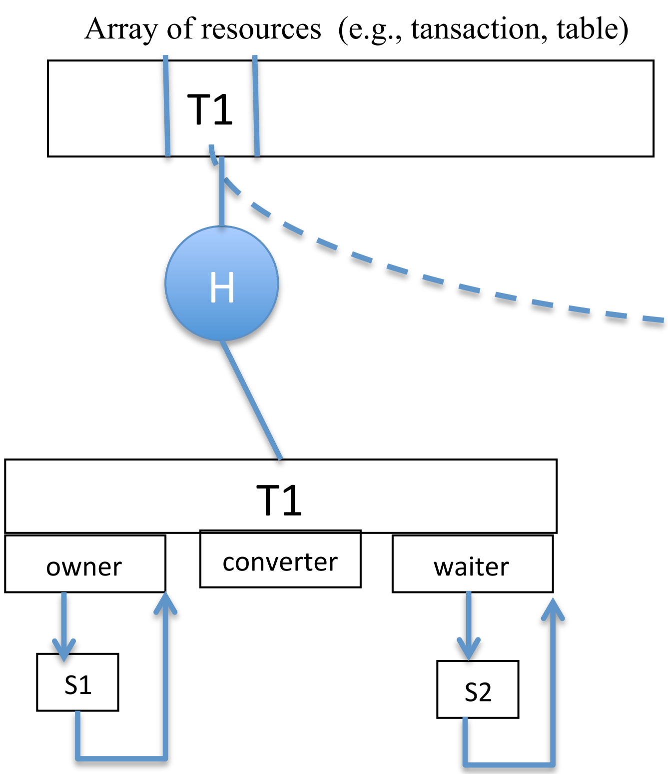
spid	dbid	objid	lock granularity	lock mode	lock owner	lock waiter
10	1	117	RID	X	LO1	LW1, LW4
10	1	117	PAG	IX	LO1	
10	1	117	TAB	IX	LO1	LW3
10	1	118	RID	S	LO2, LO3	LW2

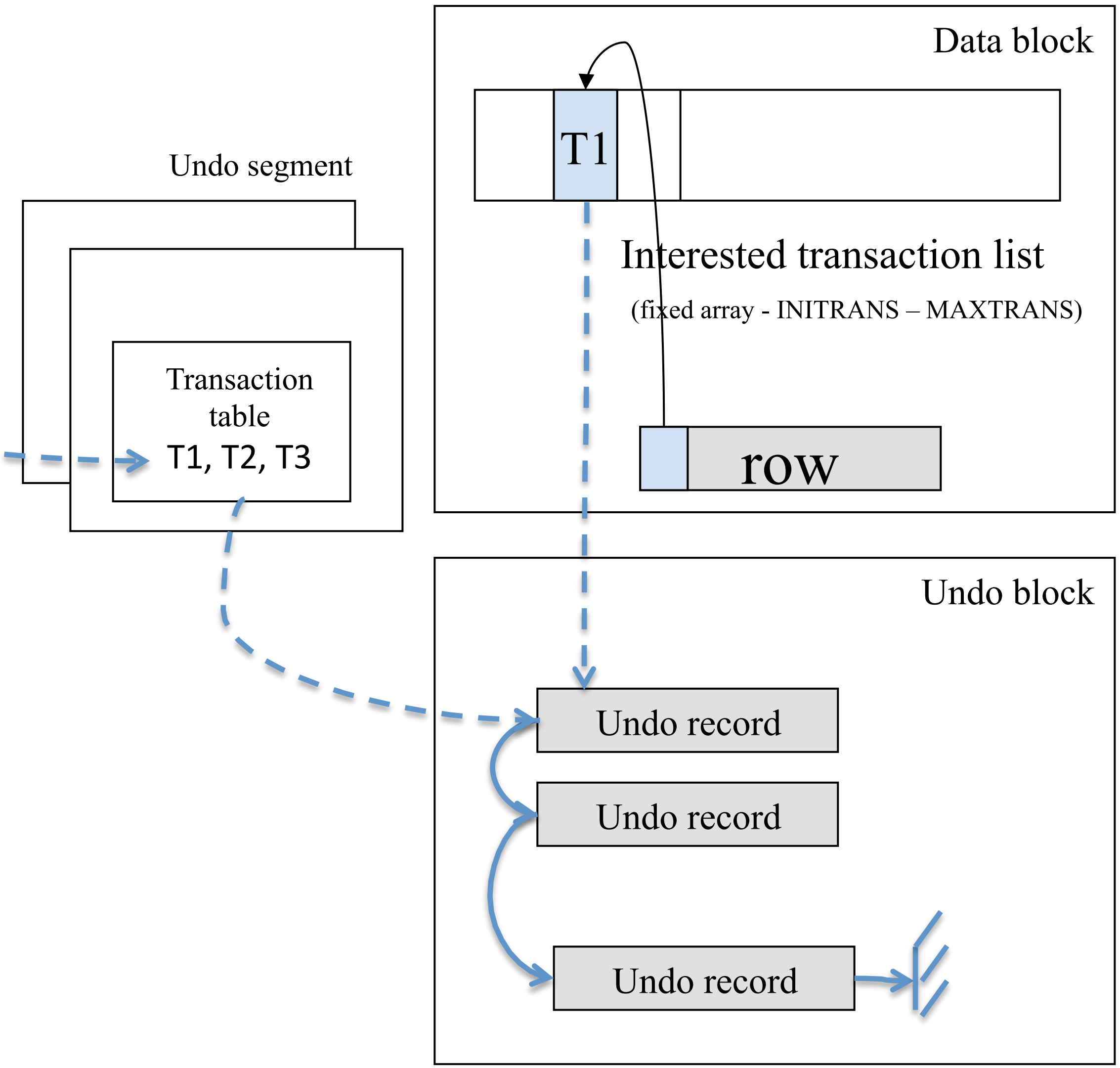
Lock – 32 bytes

Lock owner block – 32 bytes
Lock waiter block – 32 bytes

Locking in Oracle

Enqueue resource structure
(fixed array – default 4 entries per transaction)





<u>Deadlock detection</u>:
Enqueue wait (time out ~ 3sec)

Source: Oracle Core by Jonathan Lewis

Lock Compatibility Table in DB2

	State of Held Resource										
State Being Requested	**None**	**IN**	**IS**	**NS**	**S**	**IX**	**SIX**	**U**	**X**	**Z**	**NW**
None	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
IN (Intent None)	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes
IS (Intent Share)	yes	yes	yes	yes	yes	yes	yes	yes	no	no	no
NS (Scan Share)	yes	yes	yes	yes	yes	no	no	yes	no	no	yes
S (Share)	yes	yes	yes	yes	yes	no	no	yes	no	no	no
IX (Intent Exclusive)	yes	yes	yes	no	no	yes	no	no	no	no	no
SIX (Share with Intent Exclusive)	yes	yes	yes	no	no	no	no	no	no	no	no
U (Update)	yes	yes	yes	yes	yes	no	no	no	no	no	no
X (Exclusive)	yes	yes	no	no	no	no	no	no	no	no	no
Z (Super Exclusive)	yes	no	no	no	no	no	no	no	no	no	no
NW (Next Key Weak Exclusive)	yes	yes	no	yes	no	no	no	no	no	no	no

None/IN: no lock

SIX: combination of S and IX

U: Similar to S

Z: lock held when table is created/dropped, indexes are created, table is reorganized

NW: next key locking

Source: DB2 10.1 LUW lock compatiblity , lock mode

Concurrency Control Goals

- Performance goals
  - Reduce blocking
    - One transaction waits for another to release its locks
  - Avoid deadlocks
    - Transactions are waiting for each other to release their locks

- Correctness goals
  - Serializability: each transaction appears to execute in isolation
  - The programmer ensures that serial execution is correct.

Trade-off between correctness and concurrency

Ideal Transaction

- Acquires few locks and favors shared locks over exclusive locks
  - Reduce the number of conflicts -- conflicts are due to exclusive locks
- Acquires locks with *fine granularity*
  - Reduce the scope of each conflict
- Holds locks for a short time
  - Reduce waiting

Lock Tuning

- Transaction Chopping
  - Rewriting applications to obtain best locking performance
- Isolation Levels
  - Relaxing correctness to improve performance
- Counters
  - Using system features to circumvent bottlenecks

Example: Simple Purchases

- Purchase item I for price P
  1. If cash < P then roll back transaction (constraint)
  2. Inventory(I) := inventory(I)+P
  3. Cash := Cash – P
- Two purchase transaction P1 and P2
  - P1 has item I for price 50
  - P2 has item I for price 75
  - Cash is 100

Example: Simple Purchases

- If 1-2-3 as one transaction then one of P1, P2 rolls back.
- If 1, 2, 3 as three distinct transactions:
  - P1 checks that cash > 50. It is.
  - P2 checks that cash > 75. It is.
  - P1 completes. Cash = 50.
  - P2 completes. Cash = - 25.

Example: Simple Purchases

- Orthodox solution
  - Make whole program a single transaction
    - Cash becomes a bottleneck!
- Chopping solution
  - Find a way to rearrange and then chop up the transactions without violating serializable isolation level.

Example: Simple Purchases

- Chopping solution:
  1. If Cash < P then roll back.
     Cash := Cash – P.
  2. Inventory(I) := inventory(I) + P
- Chopping execution:
  – P11: 100 > 50. Cash := 50.
  – P21: 75 > 50. Rollback.
  – P12: inventory := inventory + 50.

Transaction Chopping

- Execution rules:
  - When pieces execute, they follow the partial order defined by the transactions.
  - If a piece is aborted because of a conflict, it will be resubmitted until it commits
  - If a piece is aborted because of an abort, no other pieces for that transaction will execute.

Transaction Chopping

- Let T1, T2, …, Tn be a set of transactions. A chopping partitions each Ti into pieces ci1, ci2, …, cik.
- A chopping of T is rollback-safe if (a)T does not contain any abort commands or (b) if the abort commands are in the first piece.

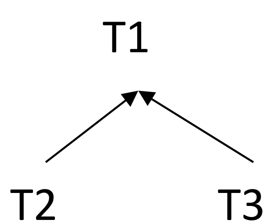
Correct Chopping

- Chopping graph (variation of the serialization graph):
  - Nodes are pieces
  - Edges:
    - C-edges: C stands for conflict. There is a C-edge between two pieces from different transactions if they contain operations that access the same data item and one operation is a write.
    - S-edges: S stands for siblings. There is an S-edge between two pieces, iff they come from the same transaction.
- A chopping graph contains an S-C cycle if it contains a cycle that includes at least one S-edge and one C-edge.

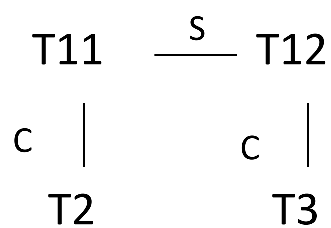
Correct Chopping

- A chopping is correct if it is rollback safe and its chopping graph contains no SC-cycle.

T1: r(x) w(x) r(y) w(y)
T2: r(x) w(x)
T3: r(y) w(y)

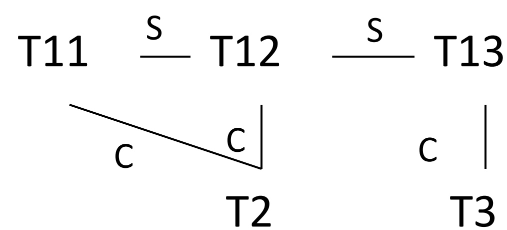


T11: r(x) w(x)
T12: r(y) w(y)



CORRECT

T11: r(x)
T12: w(x)
T13: r(y) w(y)



NOT CORRECT

Chopping Example

T1: RW(A) RW (B)

T2: RW(D) RW(B)

T3: RW(E) RW(C)

T4: R(F)

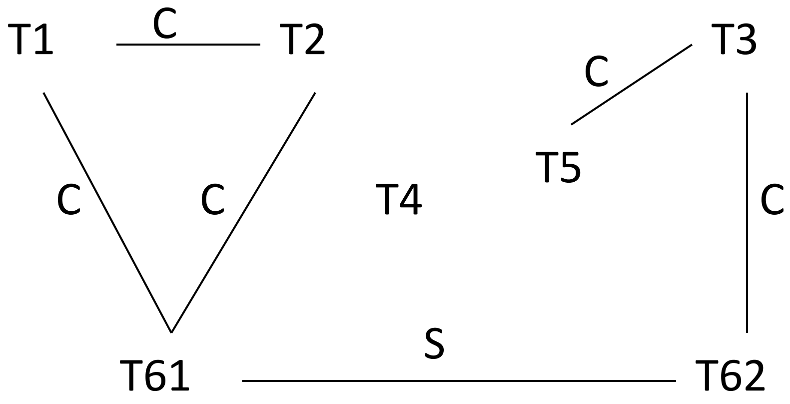
T5: R(E)

T6: R(A) R(F) R(D) R(B) R(E) R(G) R(C)

Chopping Example

T61: R(A) R(F) R(D) R(B)

T62: R(E) R(G) R(C)



Finest Chopping

- A private chopping of transaction Ti, denoted private(Ti) is a set of pieces {ci1, ci2, …, cik} such that:
  - {ci1, ci2, …, cik} is a rollback safe chopping
  - There is no SC-cycle in the graph whose nodes are {T1, …, Ti-1, ci1, ci2, …, cik, Ti+1, … Tn}
- The chopping consisting of {private(T1), private(T2), …, private(T2)} is rollback-safe and has no SC-cycles.

Finest Chopping

- In: T, {T1, .. Tn-1}
- Initialization
  - If there are abort commands
    - then p_1 := all writes of T (and all non swappable reads)that may occur before or concurrently with any abort command in T
    - else p_1 := first database access
  - P := {x | x is a database operation not in p_1}
  - P := P {p_1}

Finest Chopping

- Merging pieces
  - Construct the connected components of the graph induced by C edges alone on all transactions {T1, ..., Tn-1} and on the pieces in P.
  - Update P based on the following rule:
    - If p_j and p_k are in the same connected component and $j < k$, then
      - add the accesses from p_k to p_j
      - delete p_k from P

Sacrificing Isolation for Performance

A transaction that holds locks during a screen interaction is an invitation to bottlenecks

– Airline Reservation
  1. Retrieve list of seats available
  2. Talk with customer regarding availability
  3. Secure seat

- Single transaction is intolerable, because each customer would hold lock on seats available.
- Keep user interaction outside a transactional context

Problem: ask for a seat but then find it's unavailable. More tolerable.

Isolation Levels

- Read Uncomitted (No lost update)
  - Exclusive locks for write operations are held for the duration of the transactions
  - Lock for writes until commit time. No locks for reads
- Read Committed (No inconsistent retrieval)
  - Lock for writes until commit time.
  - Shared locks are released as soon as the read operation terminates.
- Repeatable Read (no unrepeatable reads)
  - Strict two phase locking: lock for writes and reads until commit time.
- Serializable (no phantoms)
  - Table locking or index locking to avoid phantoms

Logical Bottleneck: Sequential Key generation

- Consider an application in which one needs a sequential number to act as a key in a table, e.g. invoice numbers for bills.
- Ad hoc approach: a separate table holding the last invoice number. Fetch and update that number on each insert transaction.
- Counter approach: use facility such as Sequence (Oracle)/Identity(SQL Server).