Lesson 12: Transactions & Concurrency Control

Contents

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block
      - data is read back after being written to ensure that the write was successful

Measures of Query Cost (Cont.)

- For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks

    $b * t_T + S * t_S$
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation

Selection Operation

- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition.
- **A1** (*linear search*). Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate = b_r block transfers + 1 seek
    - b_r denotes number of blocks containing records from relation r
  - If selection is on a key attribute, can stop on finding record
    - cost = $(b_r/2)$ block transfers + 1 seek
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices
- **A2** *(binary search).* Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - Assume that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - cost of locating the first tuple by a binary search on the blocks
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - If there are multiple records satisfying selection
      - *Add transfer cost of the* number of blocks containing records that satisfy selection condition

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A3** (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition
  - *Cost* = $(h_i + 1) * (t_T + t_S)$
- **A4** (*primary index on nonkey, equality)* Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - *Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$
- **A5** (*equality on search-key of secondary index).*
  - Retrieve a single record if the search-key is a candidate key
    - *Cost* = $(h_i + 1) * (t_T + t_S)$
  - Retrieve multiple records if search-key is not a candidate key
    - each of *n* matching records may be on a different block
    - Cost = $(h_i + n) * (t_T + t_S)$
      - Can be very expensive!

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan or binary search,
  - or by using indices in the following ways:
- **A6** (*primary index, comparison).* (Relation is sorted on A)
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple > *v;* do not use index
- **A7** (*secondary index, comparison*).
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry > *v*
  - In either case, retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information
  - Number of records of *customer*: 10,000 *depositor*: 5000
  - Number of blocks of *customer*: 400 *depositor*: 100

Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

```
for each tuple $t_r$ in $r$ do begin
 for each tuple $t_s$ in $s$ do begin
     test pair $(t_r,t_s)$ to see if they satisfy the join condition $\theta$
     if they do, add $t_r \bullet t_s$ to the result.
 end
end
```

- *r* is called the **outer relation** and *s* the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

$$(n_r * b_s + b_r) * t_T + (n_r + b_r) * t_S$$

- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- Assuming worst case memory availability cost estimate is
  - with *depositor* as outer relation:
    - 5000 $*$ 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - with *customer* as the outer relation
    - 10000 $*$ 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*depositor)* fits entirely in memory, the cost estimate will be 500 block transfers.
- Block nested-loops algorithm (next slide) is preferable.

Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block $B_r$ of $r$ do begin
    for each block $B_s$ of $s$ do begin
        for each tuple $t_r$ in $B_r$ do begin
            for each tuple $t_s$ in $B_s$ do begin
                Check if $(t_r,t_s)$ satisfy the join condition

                if they do, add $t_r \bullet t_s$ to the result.
            end
        end
    end
end
```

Block Nested-Loop Join (Cont.)

- Worst case estimate: $b_r * b_s + b_r$ block transfers + 2 * b_r seeks
  - Each block in the inner relation *s* is read once for each *block* in the outer relation (instead of once for each tuple in the outer relation
- Best case: $b_r + b_s$ block transfers + 2 seeks.
- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use *M* — 2 disk blocks as blocking unit for outer relations, where *M* = memory size in blocks; use remaining two blocks to buffer inner relation and output
    - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers + $2 \lceil b_r / (M-2) \rceil$ seeks
  - If equi-join attribute forms a key or inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
  - Use index on inner relation if available (next slide)

Indexed Nested-Loop Join

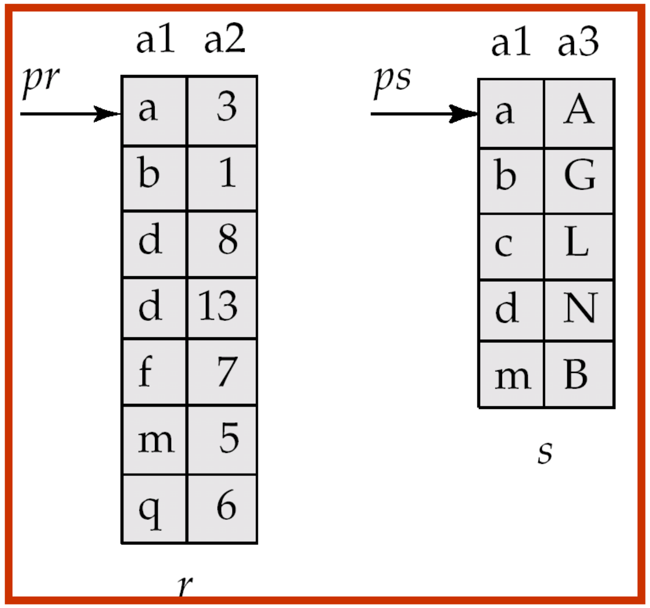
- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple t_r in the outer relation *r,* use the index to look up tuples in *s* that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost of the join: $(b_r + n_r * c)\ (t_T + t_S)$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition.
- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation.

Example of Nested-Loop Join Costs

- Compute *depositor* ⋈ *customer,* with *depositor* as the outer relation.
- Let *customer* have a primary B$^+$-tree index on the join attribute *customer-name,* which contains 20 entries in each index node.
- Since *customer* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *depositor* has 5000 tuples
- Cost of block nested loops join
  - 400*100 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks
    - assuming worst case memory
    - may be significantly less with more memory
- Cost of indexed nested loops join
  - 100 + 5000 * 5 = 25,100 block transfers and seeks.
  - CPU cost likely to be less than that for block nested loops join

Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
   1. Join step is similar to the merge stage of the sort-merge algorithm.
   2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
   3. Detailed algorithm in book



Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory
- Thus the cost of merge join is:

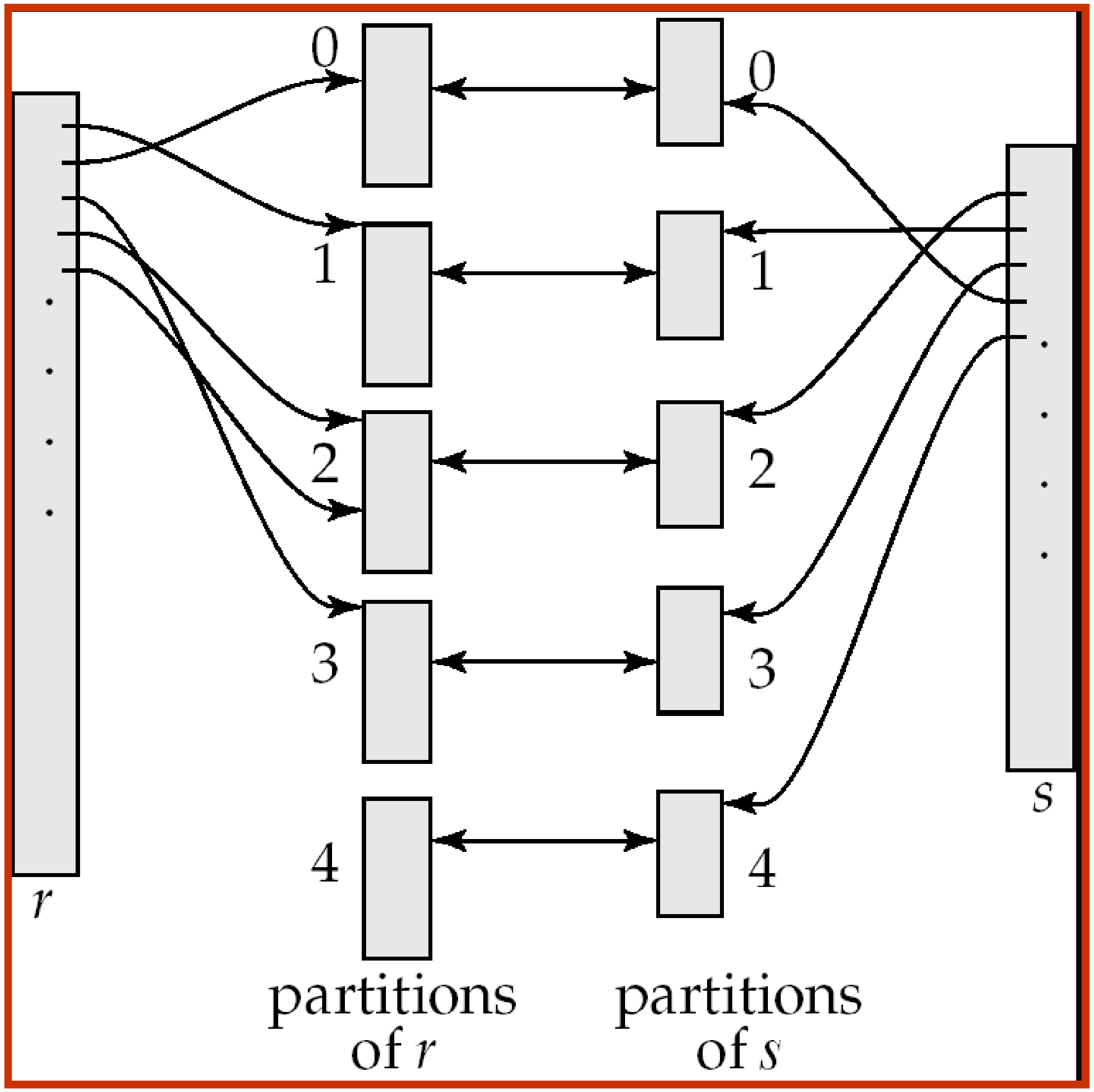
  $$b_r + b_s \text{ block transfers } + \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil \text{ seeks}$$

  - + the cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B$^+$-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B$^+$-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan more efficient than random lookup

Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
  - r_0, r_1, . . ., r_n denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - r_0,, r_1. . ., r_n denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.
- *Note:* In book, r_i is denoted as $H_{ri,}$ s_i is denoted as H_{si} and *n* is denoted as $n_{h.}$

Hash-Join (Cont.)



Hash-Join (Cont.)

- r tuples in r_i need only to be compared with s tuples in s_i Need not be compared with s tuples in any other partition, since:
  - an r tuple and an s tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value i, the r tuple has to be in r_i and the s tuple in s_i.

Hash-Join Algorithm

The hash-join of *r* and *s* is computed as follows.

1. Partition the relation *s* using hashing function *h*. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition *r* similarly.
3. For each *i:*
   (a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one *h.*
   (b) Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

Relation *s* is called the **build input** and

r is called the **probe input.**

Hash-Join algorithm (Cont.)

- The value *n* and the hash function *h* is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "fudge factor", typically around 1.2
  - The probe relation partitions s_i need not fit in memory
- **Recursive partitioning** required if number of partitions *n* is greater than number of pages *M* of memory.
  - instead of partitioning *n* ways, use *M* – 1 partitions for s
  - Further partition the *M* – 1 partitions using a different hash function
  - Use same partitioning method on *r*
  - Rarely required: e.g., recursive partitioning not needed for relations of 1GB or less with memory size of 2MB, with block size of 4KB.

Transaction

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items
  - A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes
- A transaction must see a consistent database
- During transaction execution the database may be temporarily inconsistent
  - A sequence of many actions which are considered to be one atomic unit of work
- When the transaction completes successfully (is committed), the database must be consistent
  - After a transaction commits, the changes it has made to the database persist, even if there are system failures
- Multiple transactions can execute in parallel
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

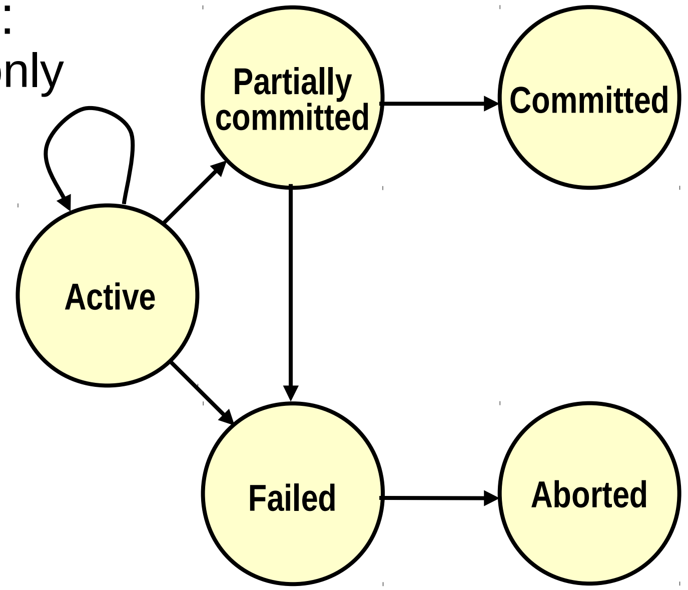
- To preserve the integrity of data the database system transaction mechanism must ensure:
  - ***Atomicity***. Either all operations of the transaction are properly reflected in the database or none are
  - ***Consistency***. Execution of a transaction in isolation preserves the consistency of the database
  - ***Isolation***. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions
    - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished
  - ***Durability***. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
  - **Atomicity requirement** – if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
  - **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.
  - **Isolation requirement** – if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be)
    - Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
    - However, executing multiple transactions concurrently has significant benefits in DBMS throughput
  - **Durability requirement** – once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.

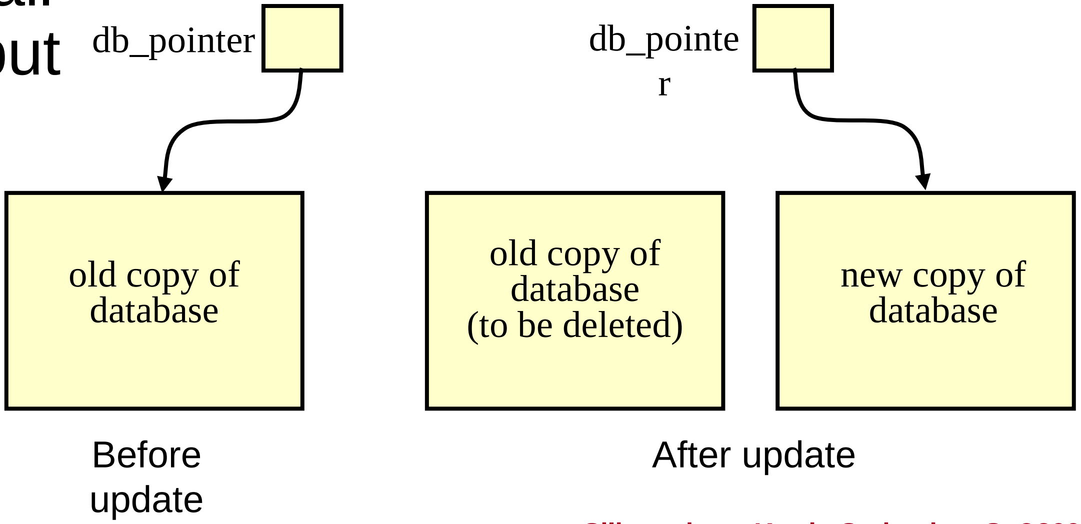
Transaction States

- **Active**
  - the initial state; the transaction stays in this state while it is executing
- **Partially committed**
  - after the final statement has been executed
- **Failed**
  - after the discovery that normal execution can no longer proceed
- **Aborted**
  - after the transaction has been rolled back and the database restored to its state prior to the start of the transaction
  - Two options after it has been aborted:
    - Restart the transaction; can be done only
    - if no internal logical error occurred
    - Kill the transaction
- **Committed**
  - after successful completion



Implementation of Atomicity and Durability

- The **recovery-management** component of a database system implements the support for atomicity and durability.
- The ***shadow-database*** scheme:
  - assume that only one transaction is active at a time.
  - a pointer called db_pointer always points to the current consistent copy of the database
  - all updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk
  - in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted
- Assumes disks do not fail
- Useful for text editors, but
  - extremely inefficient for large databases (why?)
  - Does not handle concurrent transactions
- Better schemes later



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation; that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
  - Will study later in this lesson

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement (will be omitted if it is obvious)
- A transaction that fails to successfully complete its execution will have an abort instructions as the last statement (will be omitted if it is obvious)

Correct Schedule Examples

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedules S_1 and S_2
- Schedule S_3 is not serial, but it is *equivalent* to Schedule S_1

Schedule S_1

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$tmp := A*0.1$
	$A := A - tmp$
	write(A)
	read(B)
	$B := B + tmp$
	write(B)

$T_1 \prec T_2$

Schedule S_2

T_1	T_2
	read(A)
	$tmp := A*0.1$
	$A := A - tmp$
	write(A)
	read(B)
	$B := B + tmp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

$T_2 \prec T_1$

Schedule S_3

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$tmp := A*0.1$
	$A := A - tmp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + tmp$
	write(B)

- All schedules preserve ($A + B$)

Bad Schedule

- The following concurrent schedule does not preserve the value of (*A* + *B*) and violates the consistency requirement

Schedule S_4

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	tmp := A*0.1
	$A := A - tmp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + tmp$
	write(B)

Serializability

- **Basic Assumption** – Each transaction preserves database consistency
- Thus serial execution of a set of transactions preserves database consistency
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- We ignore operations other than **read** and **write** operations (OS-level instructions), and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
    - Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.

1. I_i = **read**(Q),	I_j = **read**(Q)	I_i and I_j don’t conflict.
2. I_i = **read**(Q),	I_j = **write**(Q)	They conflict.
3. I_i = **write**(Q),	I_j = **read**(Q)	They conflict
4. I_i = **write**(Q),	I_j = **write**(Q)	They conflict

- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Serializability

- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.
- We say that a schedule *S* is **serializable** if it is conflict equivalent to a serial schedule
- Schedule S_3 can be transformed into S_6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions
  - Therefore Schedule S_3 is serializable

Schedule S_3

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule S_6

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

- Schedule S_5 is **not** serializable:
  - We are unable to swap instructions in the schedule to obtain either the serial schedule $<T_3, T_4>$, or the serial schedule $<T_4, T_3>$.

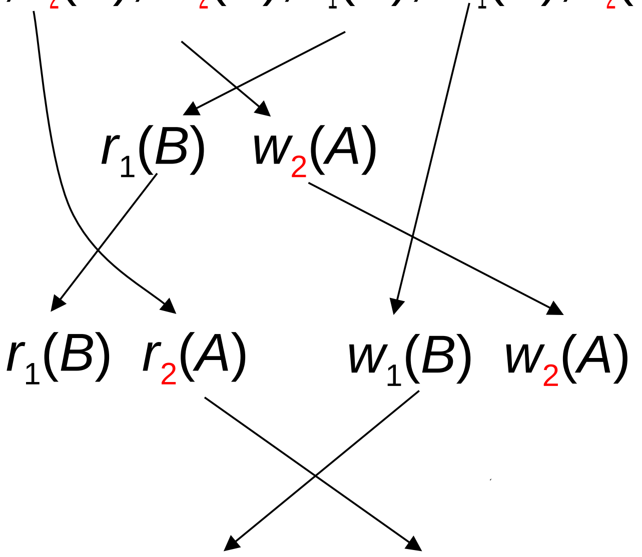
Schedule S_5

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Serializability Example

- Swapping non-conflicting actions
- Example:
  - r_1, w_1 – transaction 1 actions, r_2, w_2 – transaction 2 actions

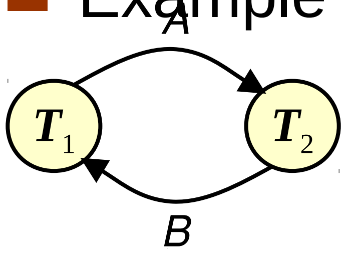
$$S = r_1(A),\ w_1(A),\ r_2(A),\ w_2(A),\ r_1(B),\ w_1(B),\ r_2(B),\ w_2(B)$$



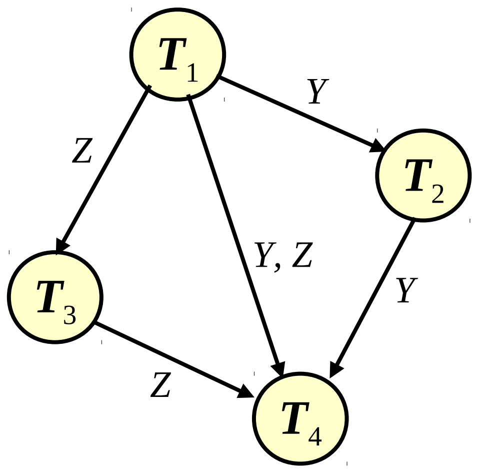
$$S' = \underbrace{r_1(A),\ w_1(A),\ r_1(B),\ w_1(B);}_{T_1}\ \ \underbrace{r_2(A),\ w_2(A),\ r_2(B),\ w_2(B)}_{T_2}$$

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** – a directed graph
  - The **vertices** are the **transactions** (names).
  - An arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
  - We may label the arc by the item that was accessed.
- Example



T_1	T_2	T_3	T_4	T_5
	$r(X)$			
$r(Y)$				
$r(Z)$				
				$r(V)$
				$r(W)$
	$r(Y)$			
	$w(Y)$			
		$w(Z)$		
$r(U)$				
			$r(Y)$	
			$w(Y)$	
			$r(Z)$	
			$w(Z)$	
$r(U)$				
$w(U)$				

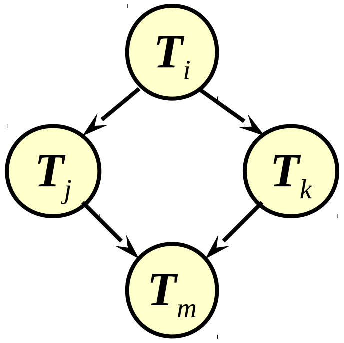


Test for Serializability

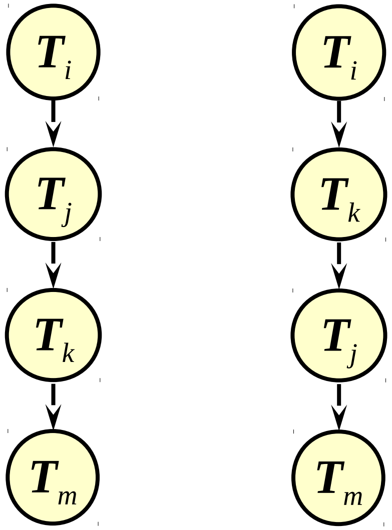
- A schedule is serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - Better algorithms take order $n + e$ where e is the number of edges
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - A linear ordering of nodes in which each node precedes all nodes to which it has outbound edges. There are one or more topological sorts.
  - For example, a serializability order for Schedule from the previous slide would be

    $$T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$$

    - Are there others?



$$\begin{tikzcd} & T_i \arrow[dl] \arrow[dr] & \\ T_j \arrow[dr] & & T_k \arrow[dl] \\ & T_m & \end{tikzcd}$$



$$\begin{tikzcd} T_i \arrow[d] & T_i \arrow[d] \\ T_j \arrow[d] & T_k \arrow[d] \\ T_k \arrow[d] & T_j \arrow[d] \\ T_m & T_m \end{tikzcd}$$

Recoverable Schedules

- Need to address the effect of transaction failures on concurrently running transactions
- **Recoverable schedule**
  - if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i ***must*** appear before the commit operation of T_j.

Schedule S_{11}

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- The schedule S_{11} is not recoverable if T_9 commits immediately after the read
- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state.
  - DBMS must ensure that schedules are recoverable

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure can lead to a series of transaction rollbacks
  - Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

  If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- This can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **Cascadeless schedules** – cascading rollbacks do not occur
  - For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency and low throughput
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are serializable, and are recoverable and cascadeless.
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur
- Tests for serializability help us understand why a concurrency control protocol is correct

Concurrency Control Mechanisms and Protocols

Lock-Based Concurrency Control Protocols

- A lock is a mechanism to control concurrent access to a data item. Data items can be locked in two modes:
  1. *exclusive* (*X*) *mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared* (*S*) *mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
  - Lock requests are made to **concurrency-control manager**. Transaction can proceed only after request is granted
  - **Lock-compatibility matrix**

	S	*X*
S	true	false
X	false	false

  - A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
  - Any number of transactions can hold shared locks on an item,
    - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted

Lock-Based Protocols (Cont.)

- Example of a transaction with locking:

T_2:

```
lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display(A+B);
```

- Locking as above is not sufficient to guarantee serializability
  - if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
  - Locking protocols restrict the set of possible schedules
- Locking may be dangerous
  - Danger of **deadlocks**
    - Cannot be completely solved – transactions have to be killed and rolled back
  - Danger of **starvation**
    - A transaction is repeatedly rolled back due to deadlocks
    - Concurrency control manager can be designed to prevent starvation
  - Compare these problems with critical sections in OS

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules
- 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
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock)

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**.
  - Here a transaction must hold all its *exclusive locks* till it commits or aborts.
- **Rigorous two-phase locking** is even stricter:
  - *All* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the locking instructions.

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls (locking is a part of these operations)
- The operation **read**(D) is processed as:

```
if T_i has a lock on D then
     read(D)
else begin
     if necessary wait until no other transaction has a lock-X on D;
     grant T_i a lock-S on D;
     read(D)
end
```

- **write**(D) is processed as:

```
if T_i has a lock-X on D then
     write(D)
else begin
     if necessary wait until no other transaction has any lock on D;
     if T_i has a lock-S on D then
          upgrade lock on D to lock-X
else
     grant T_i a lock-X on D;
write(D)
end;
```

- All locks are released after commit or abort

Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages
  - or a message asking the transaction to roll back, in case a deadlock is detected
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table

- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- Remind the ordering principle of shared resources in general approach to critical sections
- The ***tree-protocol*** is a simple kind of graph protocol

Tree Protocol

1. Only exclusive locks are considered.
2. The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i

Tree Protocol (Cont.)

- The tree protocol ensures serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - shorter waiting times, and increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
- Drawbacks
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **fine granularity** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy

- The levels, starting from the coarsest (top) level are
  - *database*
  - *area*
  - *file*
  - *record*

Deadlock Handling

- Consider the following two transactions:

T_1:	write(X)	T_2:	write(Y)
	write(Y)		write(X)

- Schedule with deadlock

T_1	T_2
lock-X on X write(X)	
	lock-X on Y write (Y) wait for **lock-X** on X
wait for **lock-X** on Y	

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies are:
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- **wait-die** scheme – non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme – preemptive
  - older transaction *wounds* (= forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme
- Both in *wait-die* and in *wound-wait* schemes:
  - a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- Timeout-Based Schemes:
  - a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back
    - thus deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge T_i T_j is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.
- For further detail see lessons on deadlocks in the OS part of the course

Deadlock Recovery

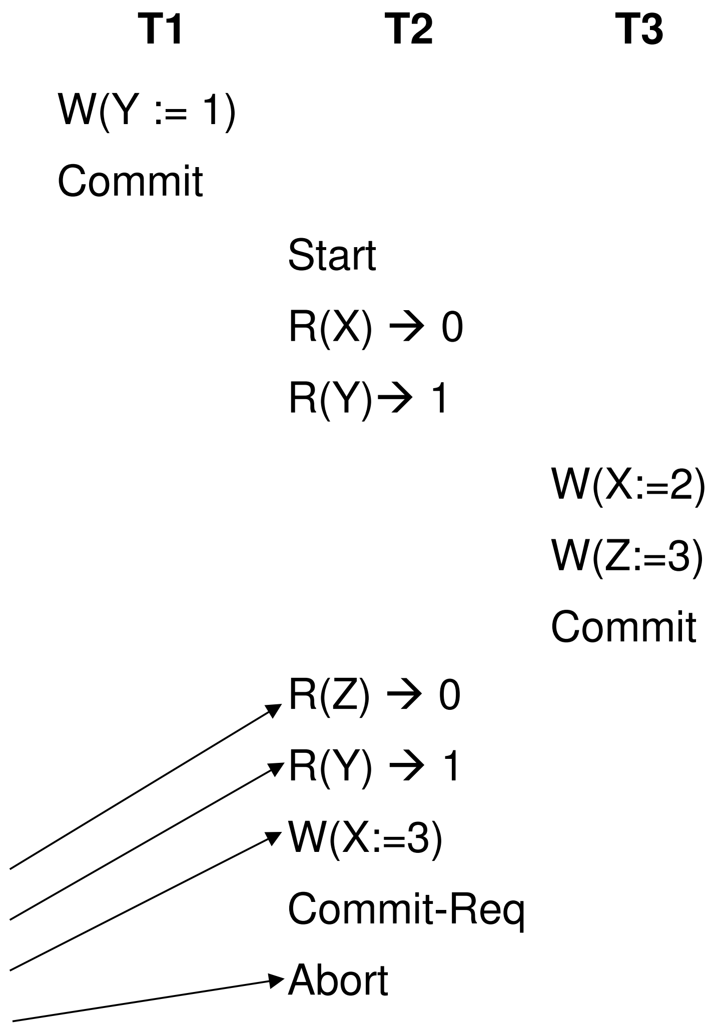
- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback – determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- Solution 1: Give logical "snapshot" of database state to read only transactions, read-write transactions use normal locking
  - Multiversion 2-phase locking
  - Works well, but how does system know a transaction is read only?
- Solution 2: Give snapshot of database state to every transaction, updates alone use 2-phase locking to guard against concurrent updates
  - Problem: variety of anomalies such as lost update can result
  - Partial solution: snapshot isolation level (next slide)
    - Proposed by Berenson et al, SIGMOD 1995
    - Variants implemented in many database systems
      - E.g. Oracle, PostgreSQL, SQL Server 2005

Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.



T1	T2	T3
W(Y := 1)		
Commit		
	Start	
	R(X) → 0	
	R(Y)→ 1	
		W(X:=2)
		W(Z:=3)
		Commit
	R(Z) → 0	
	R(Y) → 1	
	W(X:=3)	
	Commit-Req	
	Abort	

Concurrent updates not visible

Own updates are visible

Not first-committer of X

Serialization error, T2 is rolled back

Benefits of SI

- Reading is *never* blocked,
  - and also doesn’t block other txns activities
- Performance similar to Read Committed
- Avoids the usual anomalies
  - No dirty read
  - No lost update
  - No non-repeatable read
  - Predicate based selects are repeatable (no phantoms)
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated

Snapshot Isolation

- E.g. of problem with SI
  - T1: x:=y
  - T2: y:= x
  - Initially x = 3 and y = 17
    - Serial execution: x = ??, y = ??
    - if both transactions start at the same time, with snapshot isolation: x = ?? , y = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1

Snapshot Isolation Anomalies

- SI breaks serializability when txns modify *different* items, each based on a previous state of the item the other modified
  - Not very comming in practice
    - Eg. the TPC-C benchmark runs correctly under SI
    - when txns conflict due to modifying different data, there is usually also a shared item they both modify too (like a total quantity) so SI will abort one of them
  - But does occur
    - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details

SI In Oracle and PostgreSQL

- **Warning**: SI used when isolation level is set to serializable, by Oracle and PostgreSQL
  - PostgreSQL's implementation of SI described in Section 26.4.1.3
  - Oracle implements "first updater wins" rule (variant of "first committer wins")
    - concurrent writer check is done at time of write, not at commit time
    - Allows transactions to be rolled back earlier
  - Neither supports true serializable execution
- Can sidestep for specific queries by using **select .. for update** in Oracle and PostgreSQL
  - Locks the data which is read, preventing concurrent updates
  - E.g.
    1. **select max**(orderno) **from** orders <u>**for update**</u>
    2. read value into local variable maxorder
    3. insert into orders (maxorder+1, …)

End of Lesson 12

Questions?