Transaction Models and Concurrency Control

5DV052 — Advanced Data Models and Systems
Umeå University
Department of Computing Science

Stephen J. Hegner
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

Spring 2011

The Issue of Concurrency in the DBMS Context

- It is often the case that a database system will be accessed by many users simultaneously.
- If this access is read-only, then there are no serious integrity problems; only ones of performance.
- If the access includes writing the database, then serious problems will arise if the interaction is not regulated.
- It is therefore necessary to characterize correct behavior in the context of concurrent transactions.

The ACID Characterization

- The properties which a set of concurrent transactions should exhibit is often expressed via the acronym *ACID*:

Atomicity: For each transaction, either the complete result of its execution is recorded in the database, or else nothing about its results is recorded.

Consistency: The execution of any transaction in isolation preserves the integrity of the database.

Isolation: The execution of one running transaction must not affect the execution of another concurrently running transaction.

Durability: The results of the transactions are permanent in the database.

- These slides will focus primarily upon *isolation*.
- A subsequent set of slides will focus upon *atomicity* and *durability*.
- *Consistency* is a property of a single transaction and will not be the focus here.

Example Transactions

Example (simplified bank transactions) : Two transactions T_1 and T_2.

- R_i and W_i are local variables for transaction i with $i \in \{1, 2\}$.
- There are the following operations:

  $\mathrm{R\_Bal}_i\langle a\rangle$ means that transaction T_i reads the balance of account a into a local variable R_i: $R_i \longleftarrow \mathrm{Bal}\langle a\rangle$.

  $\mathrm{W\_Bal}_i\langle a\rangle$ means that transaction T_i writes the balance of account a from variable W_i to the database: $\mathrm{Bal}\langle a\rangle \longleftarrow W_i$.

  $\mathrm{Cpd\_Bal}_i\langle X\rangle$ is a local operation that adds $X\%$ interest to R_i and places the result in W_i: $W_i \longleftarrow R_i \times (1 + X/100)$.

  $\mathrm{Wthd}_i\langle X\rangle$ means that X Euros is subtracted from the local value R_i and placed in W_i: $W_i \longleftarrow R_i - X$.

T_1 Compound 10% on account 15:

$$\mathrm{R\_Bal}_1\langle 15\rangle;\ \mathrm{Cpd\_Bal}_1\langle 10\rangle;\ \mathrm{W\_Bal}_1\langle 15\rangle.$$

T_2 Withdraw 2000 from account 15:

$$\mathrm{R\_Bal}_2\langle 15\rangle;\ \mathrm{Wthd}_2\langle 2000\rangle;\ \mathrm{W\_Bal}_2\langle 15\rangle.$$

Order of Execution

- Shown below are two possibilities for schedules for these transactions.

T_1	T_2	Bal⟨15⟩	T_1	T_2	Bal⟨15⟩
R_Bal$_1$⟨15⟩		10000		R_Bal$_2$⟨15⟩	10000
Cpd_Bal$_1$⟨10⟩		10000		Wthd$_2$⟨2000⟩	10000
W_Bal$_1$⟨15⟩		11000		W_Bal$_2$⟨15⟩	8000
	R_Bal$_2$⟨15⟩	11000	R_Bal$_1$⟨15⟩		8000
	Wthd$_2$⟨2000⟩	11000	Cpd_Bal$_1$⟨10⟩		8000
	W_Bal$_2$⟨15⟩	9000	W_Bal$_1$⟨15⟩		8800

- Both schedules are *serial* and both are correct ...
    - ... even though the results differ.
- The order of *serial* execution does not affect correctness.
- The system cannot and should not decide which order is better.

Lost Updates

- If the steps of the transactions are interleaved in certain ways, updates may be lost. Shown below are two possibilities for schedules for these transactions.

T_1	T_2	$Bal\langle 15\rangle$
$R\_Bal_1\langle 15\rangle$		10000
$Cpd\_Bal_1\langle 10\rangle$		10000
	$R\_Bal_2\langle 15\rangle$	10000
	$Wthd_2\langle 2000\rangle$	10000
	$W\_Bal_2\langle 15\rangle$	8000
$W\_Bal_1\langle 15\rangle$		11000

T_1	T_2	$Bal\langle 15\rangle$
	$R\_Bal_2\langle 15\rangle$	10000
	$Wthd_2\langle 2000\rangle$	10000
$R\_Bal_1\langle 15\rangle$		10000
$Cpd\_Bal_1\langle 10\rangle$		10000
$W\_Bal_1\langle 15\rangle$		11000
	$W\_Bal_2\langle 15\rangle$	8000

- In the schedule on the left, the result of T_2 is lost.
- In the schedule on the right, the result of T_1 is lost.

Basic Steps and Transactions

- To study the issues surrounding concurrency systematically, some formal notions are necessary.

Basic steps: A *basic step* for a transaction T is either a read $\mathrm{r}\langle x\rangle$ or a write $\mathrm{w}\langle x\rangle$ of a data object x.

  - The actual values of x which are read and written are not important to the model.
  - The internal steps (*e.g.*, $\mathrm{R\_Bal}_i\langle x\rangle$, $\mathrm{R\_Bal}_i\langle x\rangle$, $\mathrm{Wthd}_i\langle n\rangle$, $\mathrm{Cpd\_Bal}_i\langle n\rangle$) are not represented.
  - Only the fact that T read or wrote that object is important.
  - For T_i, these are usually written $\mathrm{r}_i\langle x\rangle$ and $\mathrm{w}_i\langle x\rangle$, respectively.

Transaction: A *transaction* $T = \langle t_1, t_2, \ldots, t_n\rangle$ is a finite sequence of steps, with each t_i a basic step for T.

Example: $T_1 = \mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle \mathrm{w}_1\langle z\rangle$ is a transaction.

- $\mathrm{Steps}\langle T\rangle$ denotes the set of basic steps of T.

Example: $\mathrm{Steps}\langle T_1\rangle = \{\mathrm{r}_1\langle x\rangle, \mathrm{r}_1\langle y\rangle, \mathrm{w}_1\langle y\rangle, \mathrm{w}_1\langle z\rangle\}$.

Schedules

- A *schedule* for a set of transactions is a specification of the order in which the basic steps will be executed.
- Formally, let $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ be a set of transactions, with

$$T_i = \langle t_{i1}, t_{i2}, \ldots, t_{in_i} \rangle$$

for $1 \leq i \leq m$.

The steps of a schedule: Define $\mathsf{Steps}\langle \mathbf{T} \rangle = \bigcup_{i=1}^{m} \mathsf{Steps}\langle T_i \rangle$.

Schedule: A *schedule* S for $\mathbf{T}$ is any total ordering $\leq_s$ of the set $\mathsf{Steps}\langle \mathbf{T} \rangle$ with the property that $t_{ij} \leq_s t_{ik}$ whenever $j \leq_s k$.

  - In other words, the order of elements within each T_i is preserved.

Serial Schedules

Serial schedules: A schedule S for the set $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ of transactions is *serial* if there is a total ordering $\leq$ of $\mathbf{T}$ with the property that if $T_i < T_j$, then all elements of T_i occur before any element of T_j in the ordering $\leq_S$.

Examples: Let

$$T_1 = \mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle x\rangle \mathrm{w}_1\langle y\rangle$$
$$T_2 = \mathrm{r}_2\langle z\rangle \mathrm{w}_2\langle z\rangle \mathrm{w}_2\langle y\rangle$$
$$T_3 = \mathrm{r}_3\langle z\rangle \mathrm{w}_3\langle z\rangle \mathrm{r}_3\langle x\rangle \mathrm{w}_3\langle x\rangle$$

- Then

$$\mathrm{r}_2\langle z\rangle \mathrm{w}_2\langle z\rangle \mathrm{w}_2\langle y\rangle \; \mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle x\rangle \mathrm{w}_1\langle y\rangle \; \mathrm{r}_3\langle z\rangle \mathrm{w}_3\langle z\rangle \mathrm{r}_3\langle x\rangle \mathrm{w}_3\langle x\rangle$$

is the schedule corresponding to $T_2 < T_1 < T_3$, while

$$\mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \; \mathrm{r}_3\langle z\rangle \mathrm{w}_3\langle z\rangle \; \mathrm{r}_2\langle z\rangle \; \mathrm{w}_1\langle x\rangle \mathrm{w}_1\langle y\rangle \; \mathrm{w}_2\langle z\rangle \mathrm{w}_2\langle y\rangle \; \mathrm{r}_3\langle x\rangle \mathrm{w}_3\langle x\rangle$$

is not a serial schedule.

Serializability

- A serial schedule exhibits a correct semantics of concurrency, as there is no undesirable intertwining of actions of different transactions.
- Allowing only serial schedules is too restrictive.
  - It prohibits any form or concurrency whatever.
  - Performance would be compromised greatly in many situations.
- The solution is to allow *serializable* schedules – ones which are equivalent to serial schedules.
  - Parallelism is allowed.
  - The correctness of transactions is not compromised.

Question: How is *serializability* defined?

- It turns out that there are (at least) three reasonable definitions.

Three Notions of Serializability

View serializability: In view serializability, it is ensured that the reads and subsequent writes of each data object occur in the same order as in some serial schedule.

- This is the most important theoretical notion of serializability.
- It is the "correct" theoretical notion of serializability.
- Testing a schedule for view serializability is NP-complete.

Final-state serializability: In final-state serializability, it is ensured that the final result (*i.e.*, the final values of the data objects) is the same as in some serial schedule.

- This form of serializability is strictly weaker than view serializability and not widely used.
- It will not be considered further in this course.

Conflict serializability: In conflict serializability, specific forms of conflict are ruled out.

- Conflict serializability is strictly stronger than view serializability.
- It is of interest because there exist efficient algorithms for testing conflict serializability.

The Three Conditions Surrounding View Equivalence

- Let $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ be a set of transactions, and let S be a schedule for $\mathbf{T}$.
- Let $\mathrm{r}_i\langle x\rangle \in \mathrm{Steps}\langle T_i\rangle$ and $\mathrm{w}_j\langle x\rangle \in \mathrm{Steps}\langle T_j\rangle$.

Read from: $\mathrm{r}_i\langle x\rangle$ *reads from* $\mathrm{w}_j\langle x\rangle$ in S if $\mathrm{w}_j\langle x\rangle \leq_S \mathrm{r}_i\langle x\rangle$ and there is no $k \neq j$ for which $\mathrm{w}_j\langle x\rangle \leq_S \mathrm{w}_k\langle x\rangle \leq_S \mathrm{r}_i\langle x\rangle$.

Initial read: $\mathrm{r}_i\langle x\rangle$ is an *initial read* in S if there is no k for which $\mathrm{w}_k\langle x\rangle \leq_S \mathrm{r}_i\langle x\rangle$.

Final write: $\mathrm{w}_j\langle x\rangle$ is a *final write* in S if there is no $k \neq j$ for which $\mathrm{w}_j\langle x\rangle \leq_S \mathrm{w}_k\langle x\rangle$.

Example: In

$\mathrm{r}_1\langle x\rangle\mathrm{r}_1\langle y\rangle$	$\mathrm{r}_3\langle z\rangle\mathrm{w}_3\langle z\rangle$	$\mathrm{r}_2\langle z\rangle$	$\mathrm{w}_1\langle x\rangle\mathrm{w}_1\langle y\rangle$	$\mathrm{w}_2\langle z\rangle\mathrm{w}_2\langle y\rangle$	$\mathrm{r}_3\langle x\rangle\mathrm{w}_3\langle x\rangle$

- $\mathrm{r}_2\langle z\rangle$ reads from $\mathrm{w}_3\langle z\rangle$.
- $\mathrm{r}_3\langle x\rangle$ reads from $\mathrm{w}_1\langle x\rangle$.
- $\mathrm{r}_1\langle x\rangle$, $\mathrm{r}_1\langle y\rangle$, and $\mathrm{r}_3\langle z\rangle$ are initial reads.
- $\mathrm{w}_2\langle z\rangle$, $\mathrm{w}_2\langle y\rangle$, and $\mathrm{w}_3\langle x\rangle$ are final writes.

View Equivalence and View Serializability

- Let $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ be a set of transactions, and let S and S' be schedules for $\mathbf{T}$.

View equivalence: S and S' are *view equivalent*, written $S \approx_V S'$, if:

(ve-i) Every read action of the form $\mathsf{r}_i\langle x\rangle$ is, in each schedule, either an initial read or else reads from the same write action $\mathsf{w}_j\langle x\rangle$.

(ve-ii) The two schedules have the same final-write steps.

View Serializability: S is said to be *view serializable* if there is a serial schedule S' such that $S \approx_V S'$.

Examples: The following schedules are view equivalent and view serializable.

$\mathsf{r}_1\langle x\rangle\mathsf{r}_1\langle y\rangle\ \mathsf{r}_3\langle z\rangle\mathsf{w}_3\langle z\rangle\ \mathsf{r}_2\langle z\rangle\ \mathsf{w}_1\langle x\rangle\mathsf{w}_1\langle y\rangle\ \mathsf{w}_2\langle z\rangle\mathsf{w}_2\langle y\rangle\ \mathsf{r}_3\langle x\rangle\mathsf{w}_3\langle x\rangle$

$\mathsf{r}_1\langle x\rangle\mathsf{r}_1\langle y\rangle\ \mathsf{r}_3\langle z\rangle\ \mathsf{w}_1\langle x\rangle\mathsf{w}_1\langle y\rangle\ \mathsf{w}_3\langle z\rangle\ \mathsf{r}_2\langle z\rangle\ \mathsf{w}_2\langle z\rangle\mathsf{w}_2\langle y\rangle\ \mathsf{r}_3\langle x\rangle\mathsf{w}_3\langle x\rangle$

$\mathsf{r}_1\langle x\rangle\mathsf{r}_1\langle y\rangle\ \mathsf{w}_1\langle x\rangle\mathsf{w}_1\langle y\rangle\ \mathsf{r}_3\langle z\rangle\ \mathsf{w}_3\langle z\rangle\mathsf{r}_3\langle x\rangle\mathsf{w}_3\langle x\rangle\ \mathsf{r}_2\langle z\rangle\ \mathsf{w}_2\langle z\rangle\mathsf{w}_2\langle y\rangle$

- The following schedule is not view serializable.

$\mathsf{r}_1\langle x\rangle\mathsf{r}_1\langle y\rangle\ \mathsf{r}_3\langle z\rangle\mathsf{w}_3\langle z\rangle\mathsf{r}_3\langle x\rangle\ \mathsf{r}_2\langle z\rangle\ \mathsf{w}_1\langle x\rangle\mathsf{w}_1\langle y\rangle\ \mathsf{w}_2\langle z\rangle\mathsf{w}_2\langle y\rangle\ \mathsf{w}_3\langle x\rangle$

Motivation for Conditions of View Equivalence

- The motivation for condition (ve-i) is clear.

Question: Is (ve-ii) (equivalence of final writes) really necessary?

Example to motivate (ve-ii): Let

$$T_1 = \mathrm{w}_1\langle x\rangle \mathrm{w}_1\langle y\rangle \qquad T_2 = \mathrm{w}_2\langle x\rangle \mathrm{w}_2\langle y\rangle$$

- Note that:
  - In any serial schedule, the first transaction has no effect.
  - Since there are no reads, no schedule can violate (ve-i).

Example: The following schedule is not equivalent to a serial schedule, yet satisfies (ve-i):

$$\mathrm{w}_1\langle x\rangle \mathrm{w}_2\langle x\rangle \mathrm{w}_2\langle y\rangle \mathrm{w}_1\langle y\rangle$$

- In general, both (ve-i) and (ve-ii) are necessary to obtain a satisfactory notion of serializability.

Blind Writes

Examples Each of the two schedules contains write operations which do not first read the associated data object.

$$T_1 = \mathrm{w}_1\langle x\rangle \mathrm{w}_1\langle y\rangle \qquad T_2 = \mathrm{w}_2\langle x\rangle \mathrm{w}_2\langle y\rangle$$

Blind write: Let $T = \langle t_1, t_2, \ldots, t_n\rangle$ be a transaction. The operation $t_j = \mathrm{w}\langle x\rangle$ is called a *blind write* (of x) if for no $i < j$ is it the case that $t_i = \mathrm{r}\langle x\rangle$.

- Without blind writes, condition (ve-ii) would not be necessary.

Theorem: In general, the problem of deciding whether two schedules are view equivalent is NP-complete. □

But..: There is a polynomial-time algorithm to decide view serializability for the special case that none of the transactions involves blind writes.

Conflict Serializability

- Let $\mathbf{T} = \{ T_1, T_2, \ldots, T_m\}$ be a set of transactions.

Conflicting steps: The pair $\{p, q\} \subseteq \mathrm{Steps}\langle\mathbf{T}\rangle$ is said to be *conflicting* for $\mathbf{T}$ if the following three conditions hold:

- They are from distinct transactions.
- They operate on the same data object.
- At least one is a write.

Conflict equivalence: Two schedules S and S' for $\mathbf{T}$ are *conflict equivalent*, denoted $S \approx_C S'$, if for any pair $\{p, q\}$ which is conflicting for $\mathbf{T}$:

$$(p \leq_S q) \Leftrightarrow (p \leq_{S'} q)$$

Conflict serializability: The schedule S for $\mathbf{T}$ is *conflict serializable* if there is a serial schedule S' for $\mathbf{T}$ with $S \approx_C S'$.

Theorem: Every conflict-serializable schedule is also view-serializable. $\square$

An Algorithm to Decide Conflict Serializability

- Let $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ be a set of transactions, and let S be a schedule for $\mathbf{T}$.

Conflict graph: The (directed) conflict graph of S is defined as follows:

Vertices: The vertices are just the elements of $\mathbf{T}$.

Edges: There is a directed edge from T_i to T_j iff there are $p \in \mathrm{Steps}\langle T_i \rangle$, $q \in \mathrm{Steps}\langle T_j \rangle$ with $p \leq_S q$ and $\{p, q\}$ conflicting for $\mathbf{T}$.

Observation: If the conflict graph of S is acyclic, then any ordering of $\mathbf{T}$ for which $T_i \leq T_j$ identifies a serial execution of $\mathbf{T}$ which is conflict equivalent to S. $\square$

Theorem: S is conflict serializable iff its conflict graph is acyclic. $\square$

Corollary: If its conflict graph is acyclic, then S is view serializable. $\square$

Remark: The conflict graph is also called the *precedence graph*.

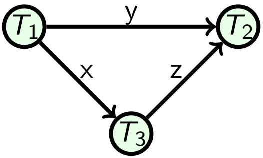
Examples of Conflict Graphs

Examples: The conflict graph for all of these schedules

$r_1\langle x\rangle r_1\langle y\rangle \; r_3\langle z\rangle w_3\langle z\rangle \; r_2\langle z\rangle \; w_1\langle x\rangle w_1\langle y\rangle \; w_2\langle z\rangle w_2\langle y\rangle \; r_3\langle x\rangle w_3\langle x\rangle$

$r_1\langle x\rangle r_1\langle y\rangle \; r_3\langle z\rangle \; w_1\langle x\rangle w_1\langle y\rangle \; w_3\langle z\rangle \; r_2\langle z\rangle \; w_2\langle z\rangle w_2\langle y\rangle \; r_3\langle x\rangle w_3\langle x\rangle$

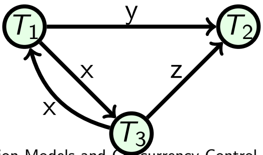
$r_1\langle x\rangle r_1\langle y\rangle \; w_1\langle x\rangle w_1\langle y\rangle \; r_3\langle z\rangle \; w_3\langle z\rangle r_3\langle x\rangle w_3\langle x\rangle \; r_2\langle z\rangle \; w_2\langle z\rangle w_2\langle y\rangle$



The edges are labelled with the data names which induce them.

Example: The conflict graph for

$r_1\langle x\rangle r_1\langle y\rangle \; r_3\langle z\rangle w_3\langle z\rangle r_3\langle x\rangle \; r_2\langle z\rangle \; w_1\langle x\rangle w_1\langle y\rangle \; w_2\langle z\rangle w_2\langle y\rangle \; w_3\langle x\rangle$



The Relationship Between View and Conflict Equivalence

Recall: Every conflict-serializable schedule is also view-serializable. □

Restricted model: In the *restricted model* of transactions, there are no blind writes.

Theorem: In the restricted model, a schedule is view serializable iff it is conflict serializable. □

Corollary: It is blind writes which force the decision problem for view serializability to be NP-complete. □

Example: For $i \in \{1, 2, 3\}$, let $T_i = \mathrm{w}_i\langle x\rangle \mathrm{w}_i\langle y\rangle$.
Then

$$\mathrm{w}_1\langle x\rangle \; \mathrm{w}_2\langle x\rangle \mathrm{w}_2\langle y\rangle \; \mathrm{w}_1\langle y\rangle \; \mathrm{w}_3\langle x\rangle \mathrm{w}_3\langle y\rangle$$

is view serializable and with $T_1 < T_2 < T_3$, but it is not conflict serializable since $T_1 \xrightarrow{x} T_2 \xrightarrow{y} T_1$ occurs in the conflict graph.

Realizing Serializable Schedules

- It is not reasonable to generate candidate schedules and then test for serializability.
- Rather, what is needed is a systematic way of guaranteeing that constructed schedules are serializable.
- There are several approaches in practice:

  Locking: Locks are used to prevent more than one transaction from writing the same data object concurrently, and also to prevent reads of objects which are being written.

  Pure optimism: Nothing is locked; conflicts are detected when transactions commit, and conflicts are resolved by aborting one or more transactions.

  Multiversioning: Each write operation generates a new version of the data object which is written. The versions are consolidated when the transactions finish.

- Many “real” approaches combine aspects of all three.

Locks

- In a lock-based approach, for a transaction to access a data object, it must request and be granted a *lock* on that object.
- There are two basic forms of lock:

Write lock: A write lock permits a transaction both to read and to write a data object.

- Only one transaction may hold a write lock on a data object at any given point in time.
- Also called an *exclusive lock* or *X-lock*.

Read lock: A read lock permits a transaction to read a data object, but not to write it.

- Several transactions may hold read locks on a data object concurrently.
- Also called a *shared lock* or *S-lock*.

Lock Requests and Releases:

- The following three basic lock operations are defined for a data object x by transaction T_i.

$\mathrm{rlk}_i\langle x\rangle$: Request a read lock on x. This request may be granted provided there are no current write locks on x.

$\mathrm{wlk}_i\langle x\rangle$: Request a write lock on x. This request may be granted provided there are no locks on x.

$\mathrm{unlk}_i\langle x\rangle$: Dissolve the lock on x held by T_i.

- There are also two operations which upgrade and downgrade locks:

$\mathrm{upgr}_i\langle x\rangle$: Convert a read lock by T_i on x to a write lock. This request may only be granted in the case that no other transaction holds a read lock on x.

$\mathrm{dngr}_i\langle x\rangle$: Convert a write lock by T_i on x to a read lock.

- For various reasons, upgrades and downgrades are sometimes excluded from a modelling situation.

Transactions with Locks

- Informally, a *transaction with locks* is a transaction with lock commands interspersed.

Transaction with locks A *transaction with locks* is a sequence T_i of elements of the form $\mathrm{r}_i\langle x\rangle$, $\mathrm{w}_i\langle x\rangle$, $\mathrm{rlk}_i\langle x\rangle$, $\mathrm{wlk}_i\langle x\rangle$, $\mathrm{unlk}_i\langle x\rangle$, $\mathrm{upgr}_i\langle x\rangle$, and $\mathrm{dngr}_i\langle x\rangle$, where x may be any data object and need not be the same for each element in the sequence, such that:

- If the operations of the form $\mathrm{rlk}_i\langle x\rangle$, $\mathrm{wlk}_i\langle x\rangle$, $\mathrm{unlk}_i\langle x\rangle$, $\mathrm{upgr}_i\langle x\rangle$, and $\mathrm{dngr}_i\langle x\rangle$ are removed, the result is an ordinary transaction.
- The sequence must obey the *locking protocol* given on the next slide.

Locking Requirements

- A transaction with locks T_i must obey the following locking rules:
  - Before a data object x is read by T_i, a lock (read or write) must be requested and granted.
  - Before a data object x is written by T_i, a write lock must be requested and granted.
  - All reads on x must be performed before the corresponding lock on x is released.
  - All writes on x must be performed before the corresponding lock on x is released or downgraded.
  - All locks must be released (via unlock) before the transaction finishes (commits).
  - It is usually (but not always) assumed that transactions do not request redundant locks.
    - This makes analyses simpler.

Locking protocol: A scheduler operates according to a *locking protocol* just in case these conventions are followed.

Examples of Transactions with Locks

- Consider the transaction $T_1 = \mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle \mathrm{w}_1\langle z\rangle$.
- The following are schedules with locks for T_1.

$\mathrm{wlk}_1\langle x\rangle \mathrm{wlk}_1\langle y\rangle \mathrm{wlk}_1\langle z\rangle \mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle \mathrm{w}_1\langle z\rangle \mathrm{unlk}_1\langle x\rangle \mathrm{unlk}_1\langle y\rangle \mathrm{unlk}_1\langle z\rangle$

$\mathrm{rlk}_1\langle x\rangle \mathrm{wlk}_1\langle y\rangle \mathrm{wlk}_1\langle z\rangle \mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle \mathrm{w}_1\langle z\rangle \mathrm{unlk}_1\langle x\rangle \mathrm{unlk}_1\langle y\rangle \mathrm{unlk}_1\langle z\rangle$

$\mathrm{rlk}_1\langle x\rangle \mathrm{r}_1\langle x\rangle \mathrm{wlk}_1\langle y\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle \mathrm{wlk}_1\langle z\rangle \mathrm{w}_1\langle z\rangle \mathrm{unlk}_1\langle x\rangle \mathrm{unlk}_1\langle y\rangle \mathrm{unlk}_1\langle z\rangle$

$\mathrm{rlk}_1\langle x\rangle \mathrm{r}_1\langle x\rangle \mathrm{unlk}_1\langle x\rangle \mathrm{wlk}_1\langle y\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle \mathrm{unlk}_1\langle y\rangle \mathrm{wlk}_1\langle z\rangle \mathrm{w}_1\langle z\rangle \mathrm{unlk}_1\langle z\rangle$

$\mathrm{rlk}_1\langle x\rangle \mathrm{r}_1\langle x\rangle \mathrm{unlk}_1\langle x\rangle \mathrm{rlk}_1\langle y\rangle \mathrm{r}_1\langle y\rangle \mathrm{upgr}_1\langle y\rangle \mathrm{w}_1\langle y\rangle \mathrm{unlk}_1\langle y\rangle \mathrm{wlk}_1\langle z\rangle \mathrm{w}_1\langle z\rangle \mathrm{unlk}_1\langle z\rangle$

Schedules with Locks

- Let $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ be a set of transactions, and let S be a schedule for $\mathbf{T}$.
- A *schedule with locks* S' is a schedule S which has been augmented with lock operations.
- More precisely, it is a sequence of operations of the form $\mathrm{r}_i\langle x\rangle$, $\mathrm{w}_i\langle x\rangle$, $\mathrm{rlk}_i\langle x\rangle$, $\mathrm{wlk}_i\langle x\rangle$, $\mathrm{unlk}_i\langle x\rangle$, $\mathrm{upgr}_i\langle x\rangle$, and $\mathrm{dngr}_i\langle x\rangle$ which satisfies:
  - If the lock, unlock, upgrade, and downgrade operations are removed, the result is a schedule.
  - The rules given on the previous slide which define when these locking operations may be applied are followed.
  - The locking protocol is followed.
    - Informally, this means that objects must be locked appropriately before they are accessed.
    - This idea is expanded on the next slide.

Locking schedule: In this case, S' is said to be a locking schedule for S.

Example of a Schedule with Locks

- Here is a nonserializable schedule considered earlier.

$$r_1\langle x\rangle r_1\langle y\rangle\ r_3\langle z\rangle w_3\langle z\rangle r_3\langle x\rangle\ r_2\langle z\rangle\ w_1\langle x\rangle w_1\langle y\rangle\ w_2\langle z\rangle w_2\langle y\rangle\ w_3\langle x\rangle$$

- Here is one valid schedule of locks for it:

$$\mathrm{rlk}_1\langle x\rangle \mathrm{wlk}_1\langle y\rangle\ r_1\langle x\rangle r_1\langle y\rangle\ \mathrm{rlk}_3\langle z\rangle\ r_3\langle z\rangle\ \mathrm{upgr}_3\langle z\rangle \mathrm{rlk}_3\langle x\rangle$$

$$w_3\langle z\rangle r_3\langle x\rangle\ \mathrm{unlk}_3\langle z\rangle \mathrm{unlk}_3\langle x\rangle \mathrm{wlk}_2\langle z\rangle\ r_2\langle z\rangle\ \mathrm{upgr}_1\langle x\rangle\ w_1\langle x\rangle w_1\langle y\rangle$$

$$\mathrm{unlk}_1\langle y\rangle \mathrm{wlk}_2\langle y\rangle\ w_2\langle z\rangle w_2\langle y\rangle\ \mathrm{unlk}_1\langle x\rangle \mathrm{wlk}_3\langle x\rangle\ w_3\langle x\rangle$$

$$\mathrm{unlk}_3\langle x\rangle \mathrm{unlk}_2\langle x\rangle \mathrm{unlk}_2\langle y\rangle$$

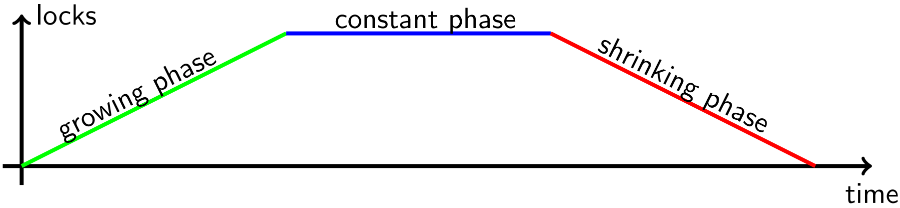
- Note that it is necessary for T_3 to lock x, release it, and then lock it again.
- This is for illustration only; it is not a reasonable schedule.

The Two-Phase Locking Protocol

- The *two-phase locking protocol* is defined for each transaction T_i individually.
- Let T_i be a transaction with locks.

Condition for two-phase locking (2PL): T_i satisfies the *two-phase locking protocol (2PL)* if no lock or upgrade operation comes after an unlock or downgrade operation in the ordering.

- All lock and upgrade operations precede all unlock and downgrade operations.



Definition: A schedule with locks is defined to be 2PL if each of its transactions with locks has that property.

Examples of 2PL

- Here is a nonserializable schedule considered earlier.

$$\mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle\ \mathrm{r}_3\langle z\rangle \mathrm{w}_3\langle z\rangle \mathrm{r}_3\langle x\rangle\ \mathrm{r}_2\langle z\rangle\ \mathrm{w}_1\langle x\rangle \mathrm{w}_1\langle y\rangle\ \mathrm{w}_2\langle z\rangle \mathrm{w}_2\langle y\rangle\ \mathrm{w}_3\langle x\rangle$$

- Here is one valid schedule of locks for it:

$$\mathrm{rlk}_1\langle x\rangle \mathrm{wlk}_1\langle y\rangle\ \mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle\ \mathrm{rlk}_3\langle z\rangle\ \mathrm{r}_3\langle z\rangle\ \mathrm{upgr}_3\langle z\rangle \mathrm{rlk}_3\langle x\rangle$$

$$\mathrm{w}_3\langle z\rangle \mathrm{r}_3\langle x\rangle\ \mathrm{unlk}_3\langle z\rangle \mathrm{unlk}_3\langle x\rangle \mathrm{wlk}_2\langle z\rangle\ \mathrm{r}_2\langle z\rangle\ \mathrm{upgr}_1\langle x\rangle\ \mathrm{w}_1\langle x\rangle \mathrm{w}_1\langle y\rangle$$

$$\mathrm{unlk}_1\langle y\rangle \mathrm{wlk}_2\langle y\rangle\ \mathrm{w}_2\langle z\rangle \mathrm{w}_2\langle y\rangle\ \mathrm{unlk}_1\langle x\rangle \mathrm{wlk}_3\langle x\rangle\ \mathrm{w}_3\langle x\rangle$$

$$\mathrm{unlk}_3\langle x\rangle \mathrm{unlk}_2\langle x\rangle \mathrm{unlk}_2\langle y\rangle$$

- In this schedule with locks, T_1 and T_2 are 2PL, but T_3 is not.
- Hence, the schedule is not 2PL.

2PL Schedules with Locks

- Let $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ be a set of transactions, let S be a schedule for $\mathbf{T}$, and let S' be a locking schedule for S.

Theorem: If S' is 2PL, then S is conflict serializable. $\square$

- Call a schedule with locks S'' *view serializable* (resp. *conflict serializable*) iff the underlying schedule without locks has that property.

Theorem: If S'' is 2PL, then it is conflict serializable. $\square$

Remark: There exist schedules with locks which are conflict serializable but not 2PL.

Example: Let

$$T_1 = \mathrm{w}_1\langle x\rangle \mathrm{w}_1\langle y\rangle$$
$$T_2 = \mathrm{r}_2\langle x\rangle \mathrm{r}_2\langle z\rangle$$
$$T_3 = \mathrm{r}_3\langle y\rangle$$

and let $S = \mathrm{w}_1\langle x\rangle \mathrm{r}_2\langle x\rangle \mathrm{r}_3\langle y\rangle \mathrm{r}_2\langle z\rangle \mathrm{w}_1\langle y\rangle$.

- S is conflict serializable with $T_3 < T_1 < T_2$.
- There is no 2PL locking schedule for S.

Assessment of 2PL

Question: To what extent is 2PL useful in real systems?

- The answer is not a simple one.
- There are at least three issues which must be considered.

  Recoverability: If a transaction does not finish normally, that is, if it *aborts*, it must be handled in such as way that preserves the integrity of the remaining transactions.

  Management of deadlock: Transactions can *deadlock* in their requests for resources. If they occur, these deadlocks must be resolved.

  Implications of locking: Locking entails significant costs, and can reduce parallelism immensely.

- Each of these issues will be considered in turn.

Termination of Transactions

- The *atomicity* requirement of ACID demands that a transaction either run to completion or else have no effect on the database.

Commit: When a transaction *commits*, its results are irrevocably entered into the database, and the transaction ceases to exist.

Abort: When a transaction *aborts*, it is terminated without entering any updates into the database.

- The model of transactions which has been considered so far does not take the possibility of abort into account.

Problem: What if a transaction aborts after executing at least one write operation?

  - A second transaction may have read from that write.
  - The effects of that second transaction must be reversed.

    Question: What if that second transaction has already committed?
  - This process can lead to *cascading aborts* of many transactions.

- A more detailed analysis of this phenomenon is required.

The Commit Operation

- When a transaction has completed its operations successfully, it *commits*.
  - The results of its operations are made a permanent part of the database.
  - The transaction ceases to exist and so cannot be aborted any more.
- The commit operation is, by definition, the last thing that a successful transaction does.
- It is useful to express the commit operation explicitly.
- Write cmt_i to indicate that transaction T_i commits.

Example: $T_1 = \mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle \mathrm{w}_1\langle z\rangle$ with explicit commit is written
$T_1 = \mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle \mathrm{w}_1\langle z\rangle \mathrm{cmt}_1$.

- Call such a representation a *transaction with explicit commit*.

Schedules with Explicit Commits

- It is often useful to write *schedules with explicit commits* for its transactions.

Examples: In this example, the respective commit operations occur immediately after the end of each transaction.

$$\mathsf{r}_1\langle x\rangle \mathsf{r}_1\langle y\rangle \; \mathsf{r}_3\langle z\rangle \mathsf{w}_3\langle z\rangle \; \mathsf{cmt}_3 \; \mathsf{r}_2\langle z\rangle \; \mathsf{w}_1\langle x\rangle \mathsf{w}_1\langle y\rangle \; \mathsf{cmt}_1 \; \mathsf{w}_2\langle z\rangle \mathsf{w}_2\langle y\rangle \; \mathsf{cmt}_2$$

- However, this is not required.
- Each of the following is also admissible.

$$\mathsf{r}_1\langle x\rangle \mathsf{r}_1\langle y\rangle \; \mathsf{r}_3\langle z\rangle \mathsf{w}_3\langle z\rangle \; \mathsf{r}_2\langle z\rangle \; \mathsf{w}_1\langle x\rangle \mathsf{w}_1\langle y\rangle \; \mathsf{w}_2\langle z\rangle \mathsf{w}_2\langle y\rangle \; \mathsf{cmt}_1 \mathsf{cmt}_2 \mathsf{cmt}_3$$

$$\mathsf{r}_1\langle x\rangle \mathsf{r}_1\langle y\rangle \; \mathsf{r}_3\langle z\rangle \mathsf{w}_3\langle z\rangle \; \mathsf{r}_2\langle z\rangle \; \mathsf{w}_1\langle x\rangle \mathsf{w}_1\langle y\rangle \; \mathsf{cmt}_3 \; \mathsf{w}_2\langle z\rangle \; \mathsf{cmt}_1 \; \mathsf{w}_2\langle y\rangle \; \mathsf{cmt}_2$$

Nonrecoverable Schedules

Example: Consider the following two simple transactions:

$$T_1 = \mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle$$
$$T_2 = \mathrm{r}_2\langle x\rangle \mathrm{w}_2\langle x\rangle$$

- and the following schedule:

$$\mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle\ \mathrm{r}_2\langle x\rangle \mathrm{w}_2\langle x\rangle\ \mathrm{cmt}_2\ \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle\ \mathrm{abort}_1$$

  in which T_1 aborts before completion.
- Note that T_2 read the value of x from T_1.
- Since T_1 aborts, this value is invalid.
- Thus, T_2 must be aborted as well.
- But it has committed.
- This is an example of a *nonrecoverable schedule*.

Cascading Nonrecoverability

Example: Consider the following three simple transactions:

$$T_1 = \mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle$$
$$T_2 = \mathrm{r}_2\langle x\rangle \mathrm{w}_2\langle x\rangle \mathrm{r}_2\langle z\rangle \mathrm{w}_2\langle z\rangle$$
$$T_3 = \mathrm{r}_3\langle z\rangle \mathrm{w}_3\langle z\rangle$$

- and the following schedule:

$\mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle$	$\mathrm{r}_2\langle x\rangle \mathrm{w}_2\langle x\rangle \mathrm{r}_2\langle z\rangle \mathrm{w}_2\langle z\rangle$	cmt_2	$\mathrm{r}_3\langle z\rangle \mathrm{r}_3\langle z\rangle$	cmt_3	$\mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle$	abort_1

- T_2 reads x from T_1 and then commits.
- T_3 reads z from T_2 and then commits.
- Both T_2 and T_3 must be aborted even though they have committed.
- This illustrates *cascading nonrecoverability*.
- It may clearly be extended to any finite number of transactions.

Recoverable Schedules

- A schedule is *recoverable* if T_j reads from T_i implies that T_i commits before T_j.

Example: Consider again the following two simple transactions:

$$T_1 = \mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle$$
$$T_2 = \mathrm{r}_2\langle x\rangle \mathrm{w}_2\langle x\rangle$$

- The first schedule below is recoverable, while the other two are not.

$$\mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle\ \mathrm{r}_2\langle x\rangle \mathrm{w}_2\langle x\rangle\ \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle\ \mathrm{cmt}_1\ \mathrm{cmt}_2$$
$$\mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle\ \mathrm{r}_2\langle x\rangle \mathrm{w}_2\langle x\rangle\ \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle\ \mathrm{cmt}_2\ \mathrm{cmt}_1$$
$$\mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle\ \mathrm{r}_2\langle x\rangle \mathrm{w}_2\langle x\rangle\ \mathrm{cmt}_2\ \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle\ \mathrm{cmt}_1$$

2PL and Recoverability

- It is easy to see that 2PL does not guarantee recoverability.

Example: The following schedule is 2PL but not recoverable.

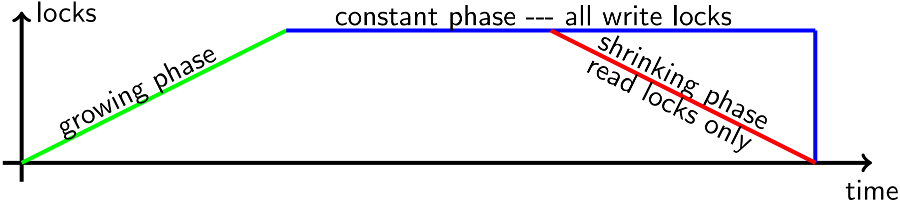
$$\mathsf{wlk}_1\langle x\rangle\mathsf{wlk}_1\langle y\rangle\ \mathsf{r}_1\langle x\rangle\mathsf{w}_1\langle x\rangle\ \mathsf{unlk}_1\langle x\rangle\mathsf{wlk}_2\langle x\rangle$$
$$\mathsf{r}_2\langle x\rangle\mathsf{w}_2\langle x\rangle\ \mathsf{unlk}_2\langle x\rangle\ \mathsf{cmt}_2\ \mathsf{r}_1\langle y\rangle\mathsf{w}_1\langle y\rangle\ \mathsf{unlk}_1\langle y\rangle\ \mathsf{cmt}_1$$

- To guarantee recoverability, transactions must not release locks too early.

Strict 2PL

- One way to ensure recoverability is to require that each transaction retain all of its write locks until it commits.
- Let T_i be a transaction with locks and explicit commit.

Condition for strict two-phase locking (S2PL): T_i satisfies the *strict two-phase locking protocol (S2PL)* if it satisfies 2PL and all write locks are held until the transaction commits.



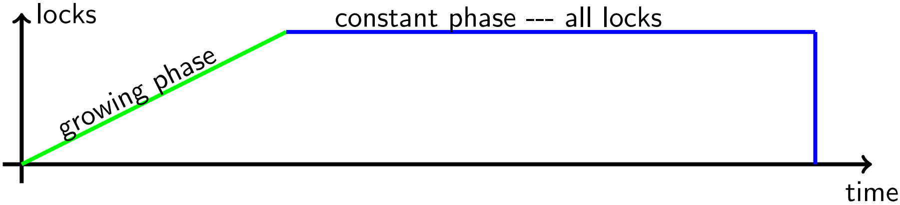
Definition: A schedule with locks and explicit commits is defined to be *strict 2PL (S2PL)* if each of its transactions has that property.

Theorem: Every S2PL schedule is recoverable. □

Superstrict 2PL

- Let T_i be a transaction with locks and explicit commit.

Condition for superstrict two-phase locking (SS2PL): T_i satisfies the *superstrict two-phase locking protocol (SS2PL)* if it satisfies 2PL and all locks (read and write) are held until the transaction commits.



Definition: A schedule with locks and explicit commits is defined to be *superstrict 2PL (SS2PL)* if each of its transactions has that property.

Theorem: Every SS2PL schedule is recoverable. □

- SS2PL is also called *rigorous 2PL*.

2PL in Real Systems

- Textbooks on database systems often state that SS2PL is widely used in practice.
- The degree to which this is true will be discussed later in these lectures.
- What can be stated is the following:
  - To the extent that 2PL is used in real systems, it is of the form SS2PL.
- Nonerecoverable schedules are almost never acceptable in real systems.

Question: Why SS2PL and not S2PL?

  - I do not have a good answer to that question.
  - Possibly complexity of implementation is an issue.

Remark: In early literature, SS2PL was sometimes called S2PL.

- This terminology is no longer used.

The Problem of Deadlock

Motivating example: Consider the following two transactions:

$$T_1 = \mathrm{r}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle x\rangle$$
$$T_2 = \mathrm{r}_2\langle y\rangle \mathrm{r}_2\langle x\rangle \mathrm{w}_2\langle y\rangle$$

- Suppose that scheduling of execution begins as follows:

$$\mathrm{wlk}_1\langle x\rangle\ \mathrm{r}_1\langle x\rangle\ \mathrm{wlk}_2\langle y\rangle\ \mathrm{r}_2\langle y\rangle$$

- To continue, either T_1 must acquire at least a read lock on y, or else T_2 must acquire at least a read lock on x.
- Neither is possible without forcing the other transaction to release a lock, which it still needs.
- A *deadlock* has occurred.
- This can happen even if T_1 and T_2 begin with read locks.

Detection of Deadlock

- Let $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ be a set of transactions.
- A *lock set* for $\mathbf{T}$ is any subset of
$$\{\mathrm{wlk}_i\langle x\rangle \mid 1 \leq i \leq m \text{ and } x \text{ is a data object}\}.$$
  - For simplicity, only write locks are considered.
- A *lock situation* for $\mathbf{T}$ is a pair (L, R) in which L and R are lock sets.
  - L is the set of locks which are currently held.
  - R is the set of locks which must be obtained in order to continue.

Wait-for graph: The (directed) *wait-for graph* for (L, R) has:

Vertices: $\mathbf{T}$.

Edges: $T_i \xrightarrow{x} T_j$ iff $\mathrm{wlk}_i\langle x\rangle \in L$ and $\mathrm{wlk}_j\langle x\rangle \in R$.

Theorem: (L, R) represents a deadlock situation iff the wait-for graph has a (directed) cycle. $\square$

Example of the Wait-For Graph

- Return to the motivating example:

$$T_1 = \mathsf{r}_1\langle x\rangle \mathsf{r}_1\langle y\rangle \mathsf{w}_1\langle x\rangle$$
$$T_2 = \mathsf{r}_2\langle y\rangle \mathsf{r}_2\langle x\rangle \mathsf{w}_2\langle y\rangle$$

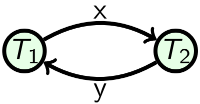
- The scheduling of execution begins as follows:

$$\mathsf{wlk}_1\langle x\rangle\ \mathsf{r}_1\langle x\rangle\ \mathsf{wlk}_2\langle y\rangle\ \mathsf{r}_2\langle y\rangle$$

- The lock sets are:

$$L = \{\mathsf{wlk}_1\langle x\rangle, \mathsf{wlk}_2\langle y\rangle\}$$
$$R = \{\mathsf{wlk}_1\langle y\rangle, \mathsf{wlk}_2\langle x\rangle\}$$

- The wait-for graph is:



Resolution of Deadlock

- To resolve deadlock, an edge (or edges) must be removed from the wait-for graph to render it acyclic.
- There are two main approaches to managing deadlock:

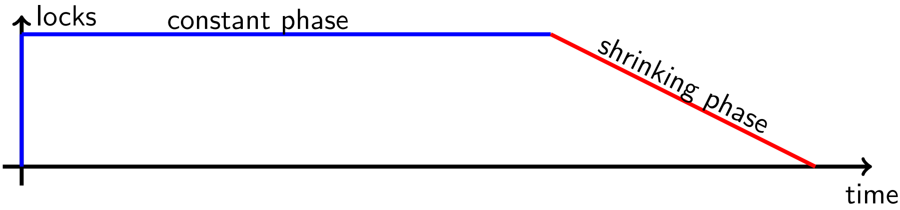
  Pessimistic resolution: Do not allow a transaction to begin until it is guaranteed that it can acquire all of the locks that it needs.

  Optimistic resolution: Allow transactions to proceed unhindered.

  - When a deadlock is detected, abort one or more transactions in order to render the wait-for graph acyclic.

Pessimistic Resolution of Deadlock

- Pessimistic resolution may be guaranteed via *conservative 2PL*, in which all locks are acquired before the transaction is allowed to proceed.



- Pessimistic resolution is seldom employed in the DBMS context.
- Typically, conflicts due to lock contention far outnumber conflicts due to deadlock.
- Also, when a transaction begins, it is not always known which resources it will need.

  $\Longrightarrow$ Conservative 2PL may result in many unnecessary locks.

Bottom line: The performance penalty imposed by conservative 2PL outweighs the advantages gained.

Optimistic Resolution of Deadlock

- Optimistic resolution of deadlock proceeds by choosing a *victim* transaction to abort when a deadlock is detected.

Livelock: Livelock (also called *starvation*) occurs when a given transaction is chosen to be the victim over and over, and so never is able to complete.

- Livelock may be avoided by timestamping each transaction with the time of its initial begin.
- When a transaction is restarted after an abort, it is restarted with its timestamp.
- In this way, transactions which have been aborted repeatedly receive increasing priority and will eventually complete.

Caution: *Optimistic resolution of deadlock* and *optimistic concurrency control* are two entirely different things which address two completely different issues.

Granularity of Locks

Question: What size of objects should be locked? (*lock granularity*)

- At first thought, it might seem best to lock the smallest possible objects.
- Smaller lock objects (finer granularity) have the advantage of allowing increased parallelism due to lesser contention for data objects.
- However, finer granularity of locks implies greater overhead from lock management.

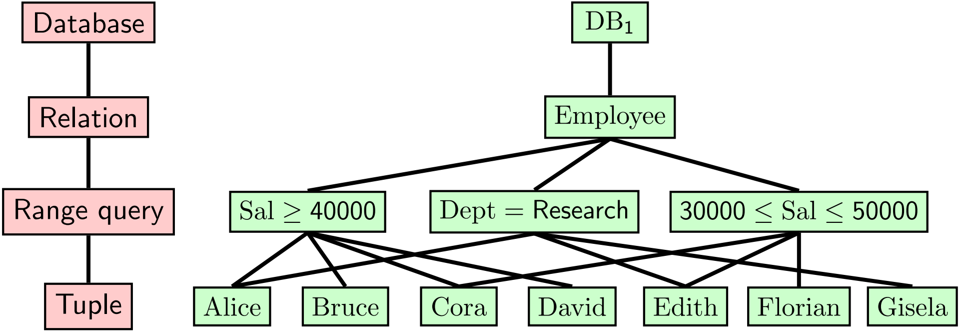
Observation: Different transactions may require different lock granularities.

- Transaction A processes a whole relation or a large part of a relation and so works best with coarse-grained locks.
- Transaction B processes only a few tuples at a time and so will interfere less with other transactions if its locks are fine grained.
- Transaction C needs to read lock an entire relation but then updates only a small part of it.

$\Longrightarrow$ It is thus advantageous to allow read and write locks of differing granularity.

Multigranularity Locking

- The classes of objects to be locked are arranged in a hierarchy.
- A simple example is shown in pink to the left below.
- An example of object instances is shown to the right in green.
- This hierarchy need not be a tree, but its graph must be acyclic.
- When an object is (read/write) locked, all objects below it are also (read/write) locked.
- Note also there is a hierarchy within the range queries (not shown).



Intention Locks

- Locking all objects below a given object provides correct semantics of multigranular locking.
- Additional efficiency may be obtained by propagating a certain type of lock to those objects above the object to be locked.
- When an object is locked, all objects above it are assigned an *intention lock* of the same type.

Example: Consider the following two transactions:

  - T_1 updates information on employee Alice.
  - T_2 updates information on employees in the $\mathrm{Research}$ department.

- Before T_1 is allowed to write lock the Alice tuple, it must obtain an intention write lock on all employees in the $\mathrm{Research}$ department, as well as the $\mathrm{Employees}$ relation and the whole database.
- This intention lock ensures that no other transaction will be able to lock those objects which subsume the Alice tuple.

Intention Locks — Formalization

Intention-to-read locks: Before a transaction T_i may obtain a read lock $\text{rlk}_i\langle x \rangle$ on object x, it must obtain an intention-to-read lock $\text{irlk}_i\langle x' \rangle$ on every data object x' above and including x in the hierarchy.

- Also called a *intention-shared lock* or *IS-lock*.

Intention-to-write locks: Before a transaction T_i may obtain a write lock $\text{wlk}_i\langle x \rangle$ on object x, it must obtain an intention-to-write lock $\text{iwlk}_i\langle x' \rangle$ on every data object x' above and including x in the hierarchy.

- Also called an *intention-exclusive lock* or *IX-lock*.

Compatibility matrix: Shows which types of locks are compatible.

Type of Lock Held		Type of Lock Requested			
		$\text{rlk}_i\langle x \rangle$	$\text{wlk}_i\langle x \rangle$	$\text{irlk}_i\langle x \rangle$	$\text{iwlk}_i\langle x \rangle$
	$\text{rlk}_j\langle x \rangle$	yes	no	yes	no
	$\text{wlk}_j\langle x \rangle$	no	no	no	no
	$\text{irlk}_j\langle x \rangle$	yes	no	yes	yes
	$\text{iwlk}_j\langle x \rangle$	no	no	yes	yes

Read with Intention to Write

- It is often the case that a transaction will require a write lock on some object x as well as a read lock on some object x' which is above x in the hierarchy.

  Example: Execute a large range query (read only), and then update (write) just a few tuples.

RIW-locks: A *read-with-intention-to-write lock* $\mathrm{riwlk}_i\langle x\rangle$ is equivalent to $\mathrm{rlk}_i\langle x\rangle$ and $\mathrm{iwlk}_i\langle x\rangle$ together.

- Also called a *shared and intention-exclusive lock* or *SIX-lock*.

Compatibility matrix: Shows which types of locks are compatible.

Type of Lock Held	Type of Lock Requested				
	$\mathrm{rlk}_i\langle x\rangle$	$\mathrm{wlk}_i\langle x\rangle$	$\mathrm{irlk}_i\langle x\rangle$	$\mathrm{iwlk}_i\langle x\rangle$	$\mathrm{riwlk}_i\langle x\rangle$
$\mathrm{rlk}_j\langle x\rangle$	yes	no	yes	no	no
$\mathrm{wlk}_j\langle x\rangle$	no	no	no	no	no
$\mathrm{irlk}_j\langle x\rangle$	yes	no	yes	yes	yes
$\mathrm{iwlk}_j\langle x\rangle$	no	no	yes	yes	no
$\mathrm{riwlk}_j\langle x\rangle$	no	no	yes	no	no

Concurrency Control in Real Systems

- SS2PL provides the degree of transaction correctness and recoverability needed for true isolation.
- However, it comes with a price.

Example: Consider an update which first requires a range query on an attribute which is not indexed.

  - Give all employees with $29000 \leq \text{Salary} \leq 30000$ a 10% raise.

- To execute that query, the entire Employee relation must be read locked in order to identify those employee which meet the range condition.
- Those parts which are to be updated must then be write locked, and this can cause a huge delay if the read lock is shared.
- Additionally, according to SS2PL, this read lock cannot be released until the entire transaction has completed.
- Locking the entire relation, even for reading, can have a serious impact on performance.
- Weaker notions of isolation are therefore often used in practice.

Standard Degrees of Isolation

- The SQL standard specifies four degrees of isolation.
- Each is described in terms of certain anomalies.
- They are summarized in the table below.

		Anomaly allowed		
Degree of Isolation	Level of isolation	Dirty read	Nonrepeatable read	Phantom
1	Read uncommitted (RU)	Yes	Yes	Yes
2	Read committed (RC)	No	Yes	Yes
3	Repeatable read (RR)	No	No	Yes
4	Serializable (SER)	No	No	No

- Each of these modes requires write locks, but not necessarily following 2PL.
- The differences lie in the degrees of read locks required.
- Because of their importance in practice, each will be discussed briefly.

Degree 1 Isolation — Read Uncommitted

- Under the *read-uncommitted (RU)* isolation level, data which are not committed may be read by a transaction.
- In the context of locking, no locks are required for reading.
  - Thus, the usual locking protocol is not followed.
- This mode allows *dirty reads*.
- An example in which T_2 reads dirty data is shown below.

$$\mathsf{wlk}_1\langle x\rangle\, \mathsf{r}_1\langle x\rangle \mathsf{w}_1\langle x\rangle\, \mathsf{wlk}_2\langle y\rangle\, \mathsf{r}_2\langle x\rangle \mathsf{r}_2\langle y\rangle \mathsf{w}_2\langle y\rangle\, \mathsf{r}_1\langle z\rangle\, \mathsf{abort}_1$$

- Since T_1 aborts, the value which it wrote for x is invalid.
- However, T_2 uses it anyway.
- Degree 1 isolation is useful in computing summary results, where small errors are not an issue.

Degree 2 Isolation — Read Committed

- Under the *read-committed (RC)* isolation level, only committed data may be read.
  - There are no other guarantees, however.
- In the setting of a locking protocol, this isolation level is often implemented by requiring that a transaction acquire a read lock before reading a given data item.
  - But the lock may be released as soon as the item has been read.
  - Another transaction may alter that data before the original reader commits.
- This mode allows *nonrepeatable reads*.
- An example is shown on the next slide.

Example of Nonrepeatable Read

- An example in which T_1 performs a nonrepeatable read is shown below.

$$\mathrm{rlk}_1\langle x\rangle\ \mathrm{r}_1\langle x\rangle\ \mathrm{unlk}_1\langle x\rangle\mathrm{wlk}_2\langle x\rangle\mathrm{wlk}_2\langle y\rangle\ \mathrm{r}_2\langle x\rangle\mathrm{r}_2\langle y\rangle\mathrm{w}_2\langle x\rangle\mathrm{w}_2\langle y\rangle$$
$$\mathrm{unlk}_2\langle x\rangle\mathrm{unlk}_2\langle y\rangle\mathrm{rlk}_1\langle y\rangle\mathrm{wlk}_1\langle z\rangle\ \mathrm{r}_1\langle y\rangle\mathrm{w}_1\langle z\rangle\ \mathrm{unlk}_1\langle y\rangle\mathrm{unlk}_1\langle z\rangle$$

Concrete interpretation: x and y hold account balances.

- T_1 computes $z \leftarrow x + y$.
- T_2 transfers 100€ from x to y.

- Note that this is not possible with 2PL.
- Note that the read uncommitted error of the previous slide is not possible with read committed.

Degree 3 Isolation — Repeatable Read

- Under the *repeatable read (RR)* isolation level, the value read from a single data item must be the same over multiple reads by the transaction.
- However, the set of tuples in a range query may change.
- In the context of locks, this issue is whether read locks are allowed on nonexistent tuples.
  - With RR, read locks are not necessary on nonexistent tuples.
  - With serializabe isolation (SER), read locks are necessary for all *possible* tuples in the range of the query.
  - Such nonexistent tuples are called *phantoms*.
- An example is presented on the next slide.

Example of Phantom with Repeatable Read

- An example in which T_1 performs a repeatable read may be expressed informally as follows.
  - T_1 computes the sum of the salaries of all employees.
  - T_2 inserts a new employee with a positive salary.
  - T_1 computes again the sum of the salaries of all employees.
- The second read by T_1 may return a different value than the first.
- The inserted tuple is called a *phantom*.
- This is not possible with 2PL, since in 2PL all tuples in the range of the query must be locked.
- Note, however, that the example of nonrepeatable read given previously is not possible with repeatable read isolation.
  - Reads of existing data items are by definition repeatable under RR isolation.

Full Isolation — Serializability?

Obvious fact: The serializable isolation SER mode of the SQL standard is view serializability. Right?

- Wrong! Nothing is ever easy with the SQL standard.
- The SQL standard defines serializability as the absence of dirty reads, nonrepeatable reads, and phantoms...
  - ... and there are anomalies which pass those three tests yet violate the SER isolation level.

Question: But surely the major DBMS vendors implement the standard SQL serializable isolation level as SS2PL?

Answer: Of the five major DBMSs Oracle, IBM DB2, Microsoft SQL Server, PostgreSQL and MySQL/InnoDB...

  - ...only SQL Server even provides true serializable mode.

- The others provide something called *snapshot isolation* for serializable isolation.

Multiversion Concurrency Control

- Historically, systems which work with just a single version of the database have have been widely used in DBMSs.
    - Temporary data which may be maintained, of course.
    - This model is called *single-version concurrency control (SVCC)*.
- Nowadays, however, a much more common approach is *multiversion concurrency control (MVCC)*.
- In MVCC, there may be several versions of each primitive data object (often tuple).
    - An update to data object x does not overwrite the current value of x; rather, it creates a new version.
    - Each version is tagged with a *transaction identifier*.
    - The system also keeps a *commit list* of the identifiers of all committed transactions.
    - The current version of the database (called the *stable version*) is constructed from the latest values of each object which are associated with a committed transaction.
    - Versions which are no longer needed are eventually removed.

Motivation for Studying Lock-Based Concurrency Control

Question: Given that MVCC has become the dominant form of concurrency control in DBMSs, why study SVCC lock-based approaches?

- SVCC provides a firm theoretical foundation for understanding what a concurrency-control mechanism should do.
- It thus forms something of a reference model.
- At least in part, MVCC may be viewed as a mechanism for implementing SVCC.
- This will become clearer when recovery techniques are studied.
- All that having been said, given its importance, most DBMS textbooks unfortunately do not provide anything close to adequate coverage of MVCC.

The Models of MVCC Used in this Presentation

- It is relatively straightforward to implement classical lock-based concurrency control, and in particular 2PL, S2PL, and SS2PL, within MVCC.
- For reasons of limited time, the details will not be given in these lectures.
- Rather, a higher-level *version-based* model will be used, which builds upon the idea that each transaction sees a distinct version of the database.
- The details how this version-based model is mapped to the lower-level, data-item-based model will not be given in these lectures.
- The focus here is upon concepts, not low-level details.

The Version-Based Model of MVCC

- The basic idea behind the version-based model of MVCC is that there are many *versions* of the database.
- One of these versions is, of course, the stable database, reflecting just the committed updates.
- (Obviously), the whole database is not replicated in each copy.
- Rather, the copies are implemented as descriptions of which versions of data objects apply to it.
- Uncommitted updates by a transaction T_i are usually (always?) reflected only in a private version of the database which is associated with T_i.
- Reads by T_i are made from a version which is determined by the isolation level.

Two Fundamental Modes for MVCC

- There are two fundamental modes for the version-based model of MVCC, which correspond to distinct levels of isolation.

Snapshot mode: In *snapshot mode*, a "snapshot" of the database is taken for transaction T_i when it begins execution.

- Throughout the life of the transaction, it reads from and writes to this snapshot.
- This mode is used to define a new and very important isolation mode called *snapshot isolation*.

Read-Committed dynamic mode (RC dynamic mode): In this mode, reads of data objects which the transaction has not yet written are always made from the latest committed version of that object.

- However, once a transaction has written data item x, that value becomes part of its private version and it no longer see values of x which are committed afterwards by other transactions.
- This mode is often used to implement the classical RC mode of isolation within MVCC.

Read-Committed Isolation in MVCC

- Suppose that transaction T_i is operating with the RC isolation level.
- It will then use RC dynamic mode of MVCC.
- All reads are made from the (unique) current committed version of the database.
  - Note that this version can (and usually will) change during the lifetime of T_i.
- Writes are always made to a private version of the database which is associated with T_i.
- If the transaction is to be aborted, this private version is simply discarded.
  - Since it is invisible to the other transactions, it is not necessary to "undo" anything which T_i has written to this private copy.
- Before T_i can commit, its private copy must be integrated into the database.

Managing Update Conflicts in MVCC

- Before moving on to snapshot isolation, it is important to sketch how update conflicts and transaction commits are managed in MVCC.

Update conflict: Say that T_i and T_j are in *update conflict* if their private versions contain at least one update on a common data object.

- There are many ways to resolve such conflicts.
- One common one is...

First Committer Wins: Let T_i be a transaction.

- No locks are required.
- No action is taken until a T_i is ready to commit.
- When it is, the stable version of the database is checked to see whether any committed updates have been made to data items which T_i has updated in its private version.
- If there is a conflict, T_i must be aborted.
- Otherwise, its updates are committed to the stable database.

Managing Update Conflicts in MVCC via Locks

- Update conflicts in MVCC may also be managed using write locks.
- A transaction must write lock all data objects which it intends to write, but there are no read locks.
- If T_i and T_j each hold a write lock on the same data object x, they may proceed to update their local copies.
- However, only one will be allowed to commit.
  - The choice of which is to commit may be made in many ways.
- A particular case is the following.

First updater wins: Let T_i and T_j be transactions.

- In this protocol, a transactions still write locks the data objects which it intends to update.
- If T_j already holds a lock on some data object x which T_i also wishes to write, then T_i must wait until T_j commits or aborts.
- If T_j commits, then T_i must abort.
- If T_j aborts, then T_i may obtain a write lock on x and continue.

Snapshot Isolation

Snapshot Isolation (SI): Let T_i be a transaction.

- The private version of the database for T_i is a "snapshot" of the database at the time at which T_i begins.
- This version cannot (normally) be accessed by other transactions.
- Thus, it appears to T_i that it is executing without any concurrent operations from other transactions.

- When a transaction is ready to commit, the updates to its local version must be integrated into the main database.
- This is typically realized via the *first-committer-wins* protocol.
    - Thus, no locks at all are involved.

Anomalies of Snapshot Isolation

- With SI, dirty writes and nonrepeatable reads cannot occur.
- Thus, the isolation level is at least as great as those provided by RU (read uncommitted) and RC (read committed).
- Whether SI is as strong as repeatable read depends upon technical details of the model.
  - See [Berenson et al. 1995] for a detailed discussion.
- However, with the anomaly model presented here, it is strictly stronger than RR (repeatable read).
- There are nevertheless other anomalies which cannot occur in true serializable mode.
- The two principal ones are known as *write skew* and *SI read-only anomaly*. updates.

Write Skew

Example: Let x and y represent the balances of two distinct accounts.

Integrity constraint: $x + y \geq 500$€.

Initial state: $x = 300$€, $y = 300$€.

T_1 : Withdraw 100€ from x.

T_2 : Withdraw 100€ from y.

- Suppose that the two transactions run concurrently, so that they see the same initial state "snapshot".
- Each runs without knowledge of what the other does.
- The final state will be $x = 200$€, $y = 200$€, which violates the constraint.
- This schedule clearly does not involve dirty reads, nonrepeatable reads, or phantoms, so it is passes the test for the isolation levels RU, RC, and RR.
- Write skew cannot occur with true serializability.
- Thus, SI is strictly weaker than SER isolation.

SI Read-Only Anomaly

- This anomaly is interesting in that two transactions produce a final result which is consistent with a serializable schedule.
  - However, a read-only transaction sees a state which is not possible in any serial schedule.
- Let

$$T_1 = \mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle \qquad T_2 = \mathrm{r}_2\langle x\rangle \mathrm{r}_2\langle y\rangle \mathrm{w}_2\langle y\rangle \qquad T_3 = \mathrm{r}_3\langle x\rangle \mathrm{r}_3\langle y\rangle$$

- If the schedule is

$$\mathrm{r}_2\langle x\rangle \mathrm{r}_2\langle y\rangle\ \mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle\ \mathrm{cmt}_1\ \mathrm{r}_3\langle x\rangle \mathrm{r}_3\langle y\rangle\ \mathrm{cmt}_3\ \mathrm{w}_2\langle y\rangle\ \mathrm{cmt}_2$$

  then the result which T_3 sees need not be the result of a serializable schedule.
- Note that this schedule becomes serializable if T_3 is removed.
- This is most easily seen via a concrete interpretation.

SI Read-Only Anomaly 2

- Let x and y represent the balances of bank accounts.
- If a transaction forces $x + y < 0$, then a 10% interest charge is imposed.
- Let the initial balances be $(x, y) = (0€, 0€)$.
- Let T_1 read and then add 20€ to the balance of x.
- Let T_2 read both balance and then deduct 10€ from the balance of y.
    - Note that if the snapshot for T_2 is taken before T_1 commits, then 1€ in interest is also deducted.
- The final result of

$$\mathrm{r}_2\langle x\rangle \mathrm{r}_2\langle y\rangle\ \mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle\ \mathrm{cmt}_1\ \mathrm{r}_3\langle x\rangle \mathrm{r}_3\langle y\rangle\ \mathrm{cmt}_3\ \mathrm{w}_2\langle y\rangle\ \mathrm{cmt}_2$$

  is $(x, y) = (20€, -11€)$, while T_3 sees $(x, y) = (20€, 0€)$.
- The values seen by T_3 cannot be the result of reading during any serializable execution of these transactions which computes $(20€, -11€)$ as its result, since the $-11€$ implies an interest charge.
    - The only possibilities are $(x, y) \in \{(0€, 0€), (0€, -11€), (20€, -11€)\}$.

Serializable Isolation in MVCC

- The weakness of SI, relative to serializable isolation, is that SI cannot identify certain read conflicts between two transactions.

Serializable SI: Very recently, a method for augmenting SI so that it always produces serializable isolation has been developed.

  - This method is called *serializable SI* or *SerSI*.

- It is clear that any such method must include a means of determining which data items a transaction T_i reads from in computing its updates.
- The full development is complex and will not be presented here.

Performance: In benchmark tests, it appears that in many common transaction mixes, SerSI does not incur a significant performance penalty over ordinary SI.

- It seems likely that SerSI will be incorporated into real DBMSs in the near future.

SVCC vs. MVCC in Current Systems

- While SVCC used to be the norm, virtually all current-generation DBMSs use MVCC.

Why?

  - MVCC offers superior support for concurrency control.
  - But it used to be too expensive to implement effectively.
  - Memory (both primary and secondary) has become much less expensive and available in much larger sizes.
  - MVCC requires lots of memory to store the versions.

- The sole holdout seems to be IBM DB2, which is still primarily based upon SVCC.
  - However, even that system now offers MVCC with a particular configuration for concurrency control.

Isolation Levels in Current Systems

- Most systems with MVCC offer RC and SI as options.
  - This is understandable since these two have natural implementations with MVCC.
- Most systems have RC as the default isolation level.
  - This is despite the fact that the SQL standard specifies SER as the default isolation level.
- Of the major systems (Oracle, DB2, SQL Server, PostgreSQL, MySQL/InnoDB), only SQL Server and DB2 offer SER isolation level.
- In the other systems, the isolation level which is identified as SER is really SI!

Note: In DB2, the SER isolation level is called *Repeatable Read*.

- The isolation level which is called RR in these slides is called *Read Stability* in DB2.
- The locking granularity for DB2 SER isolation is the table.
- In other words, if any part of a relation is involved in a transaction, the entire relation is locked, regardless of available indices.

Isolation Levels in Current Systems 2

- In PostgreSQL, RU is the same as RC, and RR is the same as SI.
  - This is consistent with the standard, since RU and RR isolation levels forbid certain anomalies but do not require that others be possible.
- As will be discussed shortly, RU and RR are not natural modes in MVCC.
- This PostgreSQL convention is likely used in many other systems as well.
- Many of the systems offer other non-standard modes as well.
  - In particular, locking of physical entities such as pages and files is sometimes supported.
- It is very easy to develop applications which are not portable because they use choices of isolation level which are not used by other systems.
- The best choices for portability are RC and SI.

Read-Uncommitted Isolation in MVCC

Interesting question: Does RU isolation make sense in MVCC?

- It could be implemented in a manner similar to that used for RC, except that for a data item x which T_i reads but has not yet written, the version of the database for T_i would contain the latest available version x, regardless of whether or not it has been committed.

Question: Why would this be easier to implement than RC?

Answer: In the general MVCC context, it would not be.

- Since RC provides "superior" data to RU, there is no apparent advantage to RU over RC in MVCC.

Consequence: At least in some real DBMSs (*e.g.*, PostgreSQL), RC and RU isolation levels are identical and behave as RC.

However: It might be possible to implement RU to advantage in the context of the DB cache.

  - This possibility will be discussed in the context of recovery.

Repeatable-Read Isolation in MVCC

- Repeatable read also seems a bit problematic in MVCC.
- Consider how repeatable read is implement in SVCC:
    - The transaction read locks the part of the database which corresponds to the retrieved data for the given range query Q on the DB instance when the transaction is awarded the lock.
    - These data are the correct answer to Q as long as new data which satisfy Q are not added to the database.
- In MVCC, there would have to be a version which is invariant on the result of Q on the initial database, but which may vary on other parts.
- What is the advantage of such an instance?
- It seems that this classical isolation mode does not make a lot of sense for MVCC.
- In PostgreSQL, RR and SER isolation levels are identical (implemented as SI).

Optimistic and Pessimistic Concurrency Control

- In the discussion of the resolution of deadlock for lock-based SVCC, notions of *optimistic* and *pessimistic* methods for the resolution of deadlocks were presented.
- These concepts make sense in a more general context, including but not limited to MVCC, possibly without locks and without deadlocks.

Optimisic concurrency control: refers to an approach in which transactions are allowed to proceed, with conflicts resolved at commit time.

Pessimistic concurrency control: refers to an approach in which potential conflicts are detected and resolved early on.

Example: Within MVCC, First Committer Wins is an example of optimistic concurrency control.

Example: First Updater Wins is an example which has both optimistic and pessimistic aspects.

Issues with Concurrent Isolation Modes

- In a system with multiple concurrency models, each transaction is allowed to choose its own concurrency model.
- The issue of isolation level is one of interacting transaction, and not just a single transaction.
- Thus, even if transaction T_1 chooses true serializable isolation, if transaction T_2 chooses read uncommitted, it can compromise the results of T_1.
- This underlines the necessity of having a policy for transactions which support the overall goals of the enterprise.

Transactions in Current Systems

- Major DBMSs do not in general follow SQL standard specifications in regards to directives surrounding transactions.
  - Each follows its own conventions.
  - Thus, the SQL standard will not be discussed here.
- The general convention is that transaction initiation is *implicit*.
  - It is not necessary (and in some cases not possible) to give an explicit `Begin Transaction` statement or the like.
  - On the other hand, it is generally possible to give a `Commit` or `Rollback` statement.
- It is also possible to give directives to set the isolation level.

Transaction Initiation and Commit in Current Systems

- There are two general models of transaction initiation:

  Session based: SQL statements are executed one after the other, but a commit occurs only at the end of the session or when an explicit `Commit` directive is issued.

  - Default for Oracle.

  Statement based (autocommit): A `Commit` occurs immediately after each SQL statement.

  - Default for the other four systems.

- In all cases, there is a directive to choose which of these models applies to a given session.

Long-Running Transactions

- As the name suggests, *long-running transactions* are those which take a "long" time to complete.
  - They often access many different data objects, although they may need some such objects for only short a short interval.
  - They may also involve human interaction.
- Long-running transactions pose a particularly difficult problem for concurrency control.
  - If an optimistic strategy is employed, then the risk is that transactions which have been running for a long time and are near completion must be aborted.
  - If a pessimistic strategy is employed, then the risk is that execution will be nearly serial and so there will be unacceptably long waits before a transaction is allowed to run.
- Solutions for dealing with long-running transactions must often be customized for the given application area.

Classical Reference Books on Concurrency Control

- This classical reference is available online at http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx. It contains a detailed presentation of classical MVCC.

  Bernstein, P. A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

- The following is still one of the best references on the basic theory of concurrency control. It is concise and well written.

  Papadimitriou, C., *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

Recent Reference Books on Concurrency Control

- This book is very current and presents an application-oriented perspective without going into detailed theory. It is a great book for obtaining the overall picture of transaction processing in the real world.

  Philip Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, second edition, 2009.

- This book is a comprehensive reference on the theory of concurrency control.

  Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.

Papers on Snapshot Isolation

- The following now-classical paper presents a simple yet formal model of modelling of transaction anomalies. It is the first paper to discuss snapshot isolation from a formal perspective and illustrate write skew. It is available for free download at http://arxiv.org/abs/cs/0701157.

  Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 1–10, 1995.

Papers on Snapshot Isolation 2

- The following two papers provide the theoretical foundations for augmenting SI to provide serializable isolation.

Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

Michael J. Cahill, Uwe Röhm, and Alan David Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4), 2009.

Papers on Snapshot Isolation 3

- The following paper examines SI in the context of classical schedules.

  Ragnar Normann and Lene T. Østby. A theoretical study of 'snapshot isolation'. In Luc Segoufin, editor, *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, ACM International Conference Proceeding Series, pages 44–49. ACM, 2010.

- The above paper is based upon an MSc thesis at the University of Oslo. While the above paper is not available for free online, the thesis is, at: http://www.duo.uio.no/sok/work.html?WORKID=74076

  Lene T. Østby. En teoretisk studie av "snapshot isolation". Masteroppgave, Institutt for informatikk, Universitetet i Oslo, 2008.

DBTech Resources on Transactions

- DBTech EXT is a consortium, funded by the EU, which develops educational materials in the DBMS area, with a focus upon hands-on use of real systems.
- Their main portal is here: http://dbtech.uom.gr/
- Of particular interest is the materials which they have developed for concurrency control and recovery, which may be found by clicking on the appropriate link of the above site.
- These material include not only papers, but also exercises and even a downloadable VBox image which contains Linux with the free versions of both Oracle and DB2 installed.
- Two of the participants, Martti Laiho and Fritz Laux, have written a paper entitled "On SQL Concurrency Technologies for Application Developers", which covers in detail how real systems handle concurrency control.
- It is is available for free downloaded at: http://www.dbtechnet.org/papers/SQL_ConcurrencyTechnologies.pdf.