Requirements of a DBMS

- Support for concurrent users without jeopardizing data consistency. (*concurrency control mechanism*).

- Ability to tolerate various types of failures. (*recovery algorithms*).

- For these reasons the DBMS supports the *transaction concept* as a computation model.

- Applications execute as transactions.

- A user sees a transaction as a *consistent* and *failure-resilient* execution of application(s).

- DBMS implements the *ACID* properties for transactions.

Why Permit Concurrency?

- Minimize *response time* (same reason why preemptive scheduling is used in time-sharing operating systems).

  Applications vary a lot in size. Most small, some large. So concurrency helps reduce average response time.

- Maximize *system throughput* by increasing resource utilization.

  Applications waiting for disk I/O, access to data currently being accessed by other applications, input from the terminal, etc. should not result in the CPU being idle.

- However, increasing concurrency blindly is bad– increases *conflicts*, decreases resource utilization.

- So number of concurrently executing applications should carefully be balanced to optimize throughput/response time.

- This task is complex and usually done by DBA with some system help.

Problems due to Concurrency

- Let R be a relation containing information about bank accounts.

- Let R be stored in a file f on disk.

- Let x and y be tuples containing information about two bank accounts in R, where x is located on page p_1 and y on page p_2 of file f.

- Consider an application AP1 that transfers \$100 from x to y.

- Consider another application AP2 that sums the total of balance in x and y.

- Let original balance in x be \$1000 and in y be \$2000.

- It is possible that due to concurrency between AP1 and AP2, AP2 sees $x = \$1000$ and $y = \$2100$ and reports sum as \$3100!!

- DBMS must prevent such executions.

Isolation

- synonymous to serializability
- A concurrent execution of transaction programs is serializable if it is **equivalent** to some sequential execution of the same applications
- Serializability guarantees preservation of system consistency if each transaction program/application does
         - proof by simple induction
- Thus, serializability is a sufficient condition for preserving database consistency.
- Is it necessary ?
  - In general the answer is yes, but lots of research on it …

Concurrency Control Mechanism's views Transactions

- A system consists of a collection of *indivisible* and *non-overlapping* objects $\{o_1, o_2, \ldots, o_n\}$. E.g., the set of tuples in all the relations.

- With each object is associated a domain of values.

- A state of the system is a mapping of an object to a value in its domain.

- A transaction is a sequence of *read* and *write* operations on the objects in the system.

- These operations are referred to as database operations.

- Besides the database operations, a transaction consists of certain *transaction* operations - begin, commit, and/or abort.

- Example:

begin	**begin**
A := B + 5	read(B)
	write(A)
end	**end**

Model of a Transaction

Formally, a transaction T_i is a partial order with an ordering relation $\prec_i$, where,

- $T_i \subseteq \{r_i(x), w_i(x) | x$ is an object$\} \cup \{a_i, c_i\} \cup \{b_i\}$.

- An abort (a_i) is in T_i if and only if a commit (c_i) is not in T_i.

- A commit c_i (or abort a_i) is the final operation of T_i.

- If $r_i(x), w_i(x) \in T_i$, then either $r_i(x) \prec_i w_i(x)$ or $w_i(x) \prec_i r_i(x)$.

- Each database operation is assumed to execute *atomically*.

- The execution of $r_i(x)$ in a state S results in T_i reading the value of x in state S (possibly into some program variable).

- The execution of $w_i(x)$ in state S results in a state S' in which x is assigned a value written by the $w_i(x)$ operation.

- The value assigned to x by the $w_i(x)$ operation is some function f of all the reads done by T_i previous (in $\prec_i$) to the execution of $w_i(x)$.

Example

- Consider a transaction T_i:

  begin $r_1(a)$ $w_1(b)$ $r_1(c)$ $w_1(c)$ **end**

- Assume that T_i executes from the state
  $S = \{(a, 5), (b, 10), (c, 15)\}$.

- Then,

  - $r_1(a)$ returns value 5.
  - $r_1(c)$ returns value 15.
  - $w_1(b)$ assigns b the value $f_1(a)$, for some function f_1.
  - $w_1(c)$ assigns c the value $f_2(a, c)$ for some function f_2.

Model of a Schedule

- A schedule is an interleaving of the transactions.

- Example of a schedule:

$$T_1 : r_1(x) \quad w_1(y).$$
$$T_2 : w_2(z) \quad w_2(x).$$
$$S : r_1(x) \quad w_2(z) \quad w_1(z) \quad w_2(x).$$

- Formally, $S = (\tau, \prec_S)$, where

  - τ is a set of transactions.
  - $\prec_S$ is a partial order over the operations belonging to transactions in τ. $\prec_S$ satisfies the following:

    For all $T_i \in \tau$, $\prec_i \subseteq \prec_S$.

Example

- Consider a DB with data items x and y.

- Initial DB state $\{(x,10), (y,20)\}$.

- Consider two application programs:
  AP1:   x := x - 5;   AP2   If $(x < 10)$
         y := y+ 5;   x:= y+2

- Consider the following execution:

  - AP1 reads value of x=10. AP1 writes $x = 5$.

  - AP2 reads $x = 5$. AP2 reads y $= 20$. AP2 writes x $= 22$.

  - AP1 reads y $= 22$. AP1 writes y $= 27$.

- Resulting Schedule:

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

- Resulting Transactions:

  T1: $r_1(x) \quad w_1(x) \quad r_1(y) \quad w_1(y)$

  T2: $r_2(x) \quad r_1(y) \quad w_1(y)$

- Notice that we have ignored the begin, commit, abort operations above.

- Whenever we ignore specifing the commit/abort operation, we are always talking about committed transactions.

Serializability of Schedules

- A schedule S consisting of transactions T1, T2, .. Tn is serializable if it is equivalent to some sequential execution of T1, .. Tn
- Equivalence between schedules is a relation defined over schedules.
- Different notions of equivalence lead to different defn of serializability
- Notice we are overloading the term serializability
  - Serializable execution of transaction programs versus serializable schedules
  - The former is what we desire, but the latter is what the concurrency controller can ensure
- But does ensuring serializability of schedules ensure that the execution is equivalent to some sequential execution of transaction programs?

Final State Serializability (FSR)

- A schedule is FSR if it is *final state equivalent* to some serial schedule.

- Final state equivalence: Two schedules S_1 and S_2 are equivalent if

  - They involve the same transactions.
  - Under all possible interpretations of the write operations– (that is, $w_i(x)$ of a transaction T in S could be any function f).

    For all initial states I of the system, $\{I\}S_1\{I'\}$ if and only if $\{I\}S_2\{I'\}$.

- Final state equivalence is a *syntactic* notion. The system does not try to exploit the exact functional interpretation of the transactions.
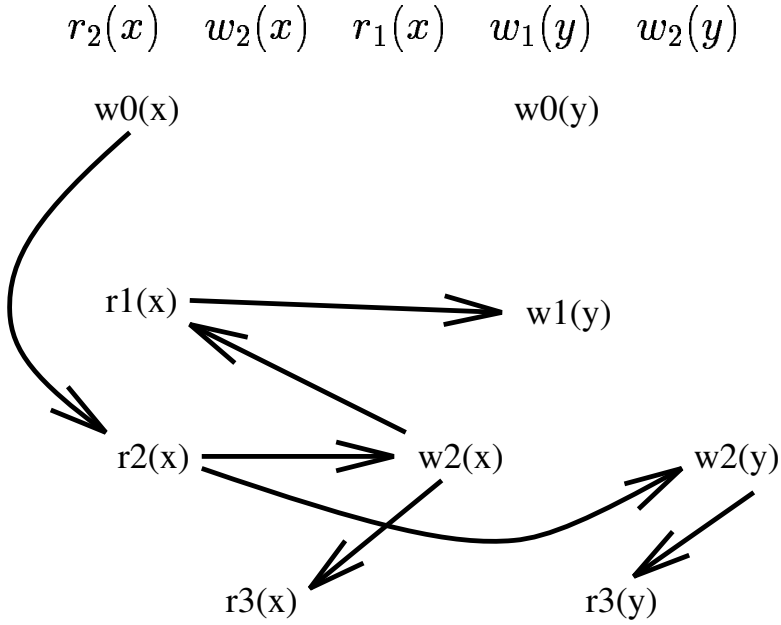
Checking Final State Equivalence

- How can we test if two schedules are final state equivalent?

- For a given schedule S let us define an *augmented* schedule $\widehat{S}$ that contains two new transactions T_0 and T_α besides the transactions in S.

- T_0 consists only of write operations, one for each object in the system.

- T_α consists only of read operations, one for each object in the system.

- The operations in $\widehat{S}$ appear in the same order as they appear in S except that all operations of T_0 appear before operation of any other transaction, and operations of T_α appear after operations of all the other transactions.

- The "pseudotransactions" T_0 defines the initial value of each object, and T_α "senses" the final state produced by the schedule S.

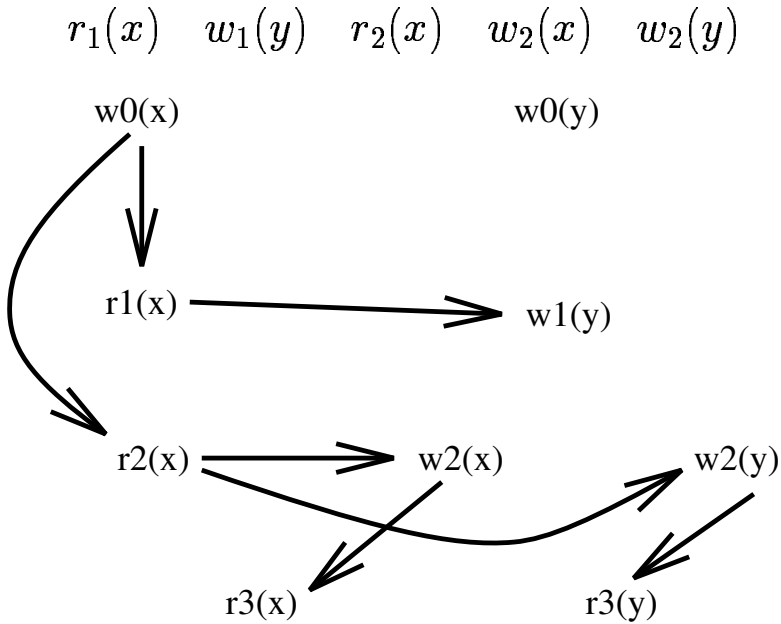# Graph Theoretic Characterization of FS Equivalence

- For the augmented schedule $\widehat{S}$ of S construct the following directed graph $D(S) = (V, E)$.

- The set of nodes V is the set of operations in all the transactions in $\widehat{S}$.

- The set of edges E contains an edge from a vertex corresponding to operation o_1 to the vertex corresponding to o_2, if $o_1 \prec_S o_2$, and either of the following holds:

  - $o_1 = r_i(x)$ and $o_2 = w_i(x)$ for some transaction T_i.
  - o_1 and o_2 belong to different transactions and o_2 *reads from* o_1.

- <u>reads from</u>: Let o_1 and o_2 be two operations in a schedule S. o_2 is said to *read from* o_1 if:

  - $o_1 = w_i(x)$, $o_2 = r_j(x)$ for some data item x.
  - $o_1 \prec_S o_2$, and there is no other operation between o_1 and o_2 in S that writes x.

- Consider the following schedule S_1:

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

w0(x)   w0(y)

r1(x) ⟶ w1(y)

r2(x) ⟶ w2(x)   w2(y)

r3(x)   r3(y)

- Consider the following schedule S_2:

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

w0(x)   w0(y)

r1(x) ⟶ w1(y)

r2(x) ⟶ w2(x)   w2(y)

r3(x)   r3(y)

- Construct from $D(S)$ a directed graph $D_1(S)$ by deleting all nodes of $D(S)$ from which no steps of T_α can be reached.

- $D_1(S_1)$:

  w0(x)   w0(y)

  r1(x)   w1(y)

  r2(x)   w2(x)   w2(y)

  r3(x)   r3(y)

- $D_1(S_2)$:

  w0(x)   w0(y)

  r1(x)   w1(y)

  r2(x)   w2(x)   w2(y)

  r3(x)   r3(y)
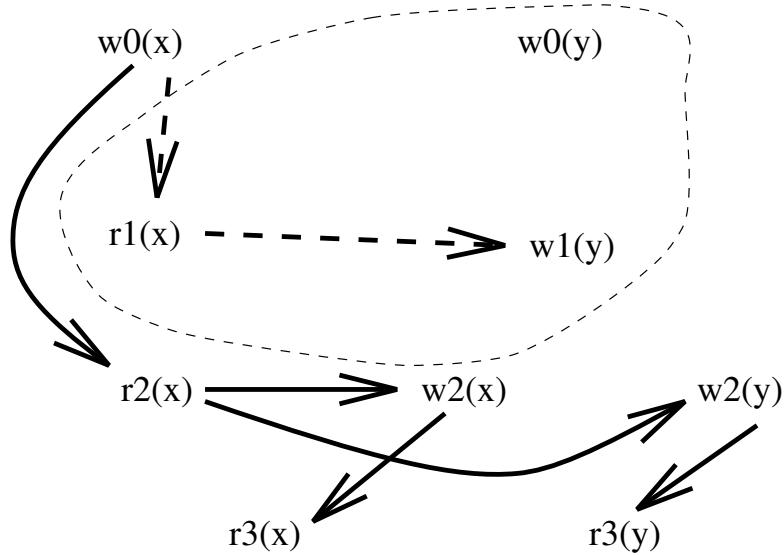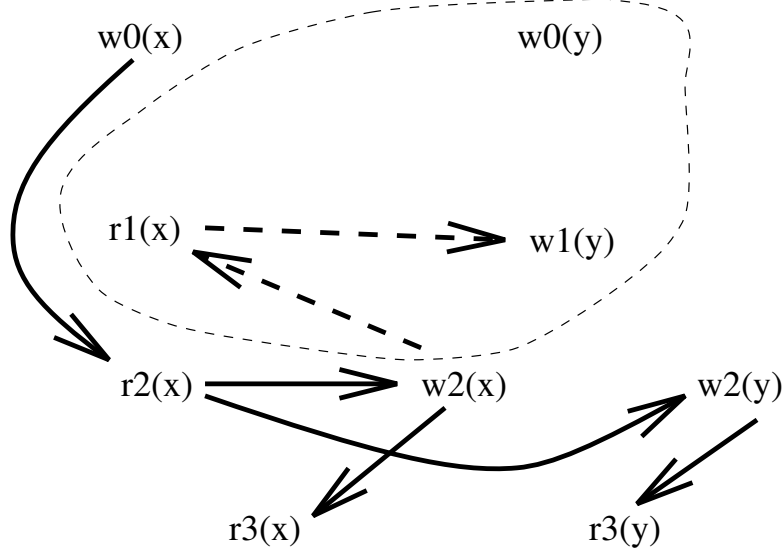
- Graph $D_1(S_1)$ is same as $D_1(S_2)$.

Final State Equivalence

- Claim: Two schedules S and S' are final-state equivalent if and only if they involve the same transactions and furthermore $D_1(s) = D_1(S')$.

- Final State Equivalence of two schedules can be tested in $O(n)$ time, where n is the length of the schedule.

Example

- Database constraint: X = Y AND Z >= 0

AP1:
  X = 7;
  Y = 7

AP2:
  Z = -1 ;
  if (X = Y)
    Z = 1

- Initial state: {x = 5, y =5,Z=2}
- Schedule S
  - w_2(z,-1), w_1(x,7), r_2(x,7), r_2(y,5), w_1(y,7)
- S is FSR
- Final state is: {X=7,Y=7,Z= -1} ← inconsistent!!!
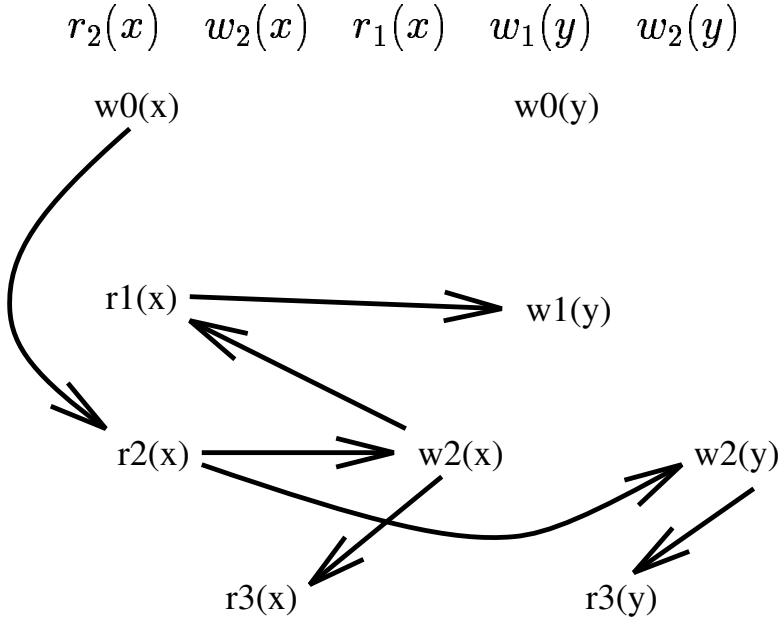
View Serializability

- A schedule is view serializable if it is *view equivalent* to some serial schedule.

- Two schedules S_1 and S_2 are view equivalent if:

  - Both S_1 and S_2 are over the same transactions.

  - For any transactions T_i and T_j in S_1 and S_2, if $o_i \in T_i$ *reads from* $o_j \in T_j$ in S_1, then o_i reads from o_j in S_2.

  - For each x, if T_i performs the final write on x in S_1, then T_i performs the final write on x in S_2.
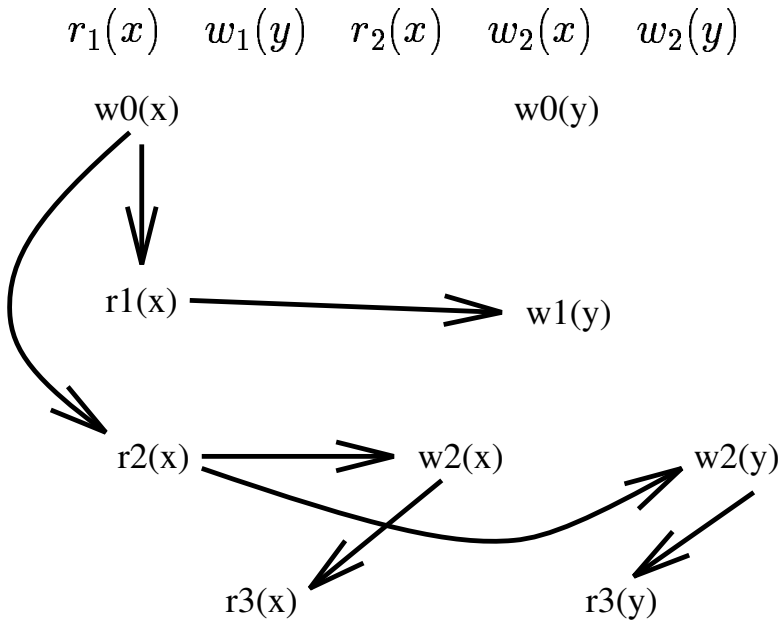
Graph Theoretic Characterization of View Equivalance

- Schedules S and S' are view equivalent if and only if they involve the same transaction and the graph $D(S) = D(S')$.

- Consider the following schedule S_1:

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

w0(x)   w0(y)

r1(x) ———————→ w1(y)

r2(x) ———————→ w2(x)   w2(y)

r3(x)   r3(y)

- Consider the following schedule S_2:

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

w0(x)   w0(y)

r1(x) ———————→ w1(y)

r2(x) ———————→ w2(x)   w2(y)

r3(x)   r3(y)

- S_1 and S_2 are not view equivalent since the graphs $D_1(S_1)$ and $D_1(S_2)$ are not identical.

VSR vrs FSR

- S is VSR $\Rightarrow$ S is FSR.

  - If schedules S_1 and S_2 be VE.
  - Thus, graph $D(S_1)$ is the same as $D(S_2)$.
  - Hence, graph $D_1(S_1)$ is the same as $D_1(S_2)$.
  - Hence, S_1 and S_2 are FSE. Hence proved.

- S is FSR $\not\Rightarrow$ S is VSR.

  - See example in previous slide of an FSR schedule that is not VSR.

- VSR ensures that each transaction's 'view' of the database (that is, the value of the data items read by a transaction) is the same as in the serial schedule.

- In contrast, FSR ensures that only the views of transactions that have *some impact* on the final state of the database is the same as in the serial schedule.

- Unlike FSR, if schedules are VSR, guarantees that the resulting state of the database is the same as the database state resulting from some serial execution of application programs

- Thus, DBMS can guarantee consistency by ensuring that schedules are VSR.

- Unfortunately, testing whether a schedule is view serializable is NP-complete.

Conflict Serializability

- A schedule S is conflict serializable, if it is *conflict equivalent* to some serial schedule.

- Schedule S and S' are conflict equivalent if and only if:

  - They contain the same transactions.

  - They order *conflicting operations* in the same way: that is, for any pair of conflicting operations o_i and o_j, if $o_i \prec_S o_j$, then $o_i \prec_{S'} o_j$.

- Operations o_i and o_j are said to conflict if

  - they belong to different transactions.

  - at least one of them is a write operation.

Graph Theoretic Characterization of CSR

- Given a schedule S construct the following directed graph $SG(S) = (V, E)$ referred to as the *serialization graph*.

- The set of nodes V in SG(S) correspond to transactions in S.

- There is an edge from T_i to T_j if there are conflicting operations o_i and o_j belonging to T_i and T_j in S such that o_i occurs before o_j in S.

- Consider a schedule:

$$w_1(x) \quad r_2(x) \quad w_2(y) \quad w_1(z) \quad w_3(y) \quad r_3(z)$$

- Its serialization graph is:
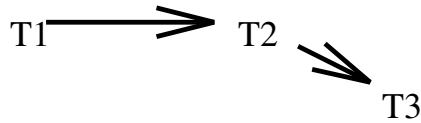
- A schedule S is CSR if and only if SG(S) is acyclic **(serializability theorem)**.

- Checking for acyclicity of S can be done is $O(n^2)$.

Further Requirements of the CCM

- The DBMS ensures that the schedules consisting of the committed transactions is conflict serializable.

- We will learn of efficient mechanisms to ensure conflict serializability later.

- Recall that another task of the DBMS is to have recovery algorithms to deal with failures. Specifically, it needs algorithms for:

  - *Transaction Abort:*
  - *System Restart:*
  - *Media Restart:*

An important Issue: Given an execution of transactions, can effects of the aborted transaction be always removed?

The answer is NO. For it to be possible to remove effects of the aborted transactions, the CCM needs to ensure not only serializability, but also *recoverability*.

Recoverability

- It may not be feasible to recover from all scenarios.

- Consider the following schedule:
  $w_1(x)$ $r_2(x)$ $w_2(y)$ c_2 a_1

- A schedule is said to be *recoverable* if T_i reads from T_j, then T_j commits before T_i commits.

- Is the following schedule recoverable:
  $w_1(x)$ $w_3(x)$ $r_2(x)$ $w_2(y)$ c_3 c_2 a_1

- Effects of the aborted transaction can be removed if a schedule is recoverable.

Avoiding Cascading Aborts

- Unfortunately, in a recoverable schedule, abortion of a transaction may result in cascading aborts of other transactions.

  $w_1(x)\ \ r_2(x)\ \ w_2(y)\ \ r_3(y)\ \ a_1\ \ a_2\ \ a_3$

- Cascading aborts can be prevented if schedules are *cascadeless* or *avoid cascading aborts* (ACA).

- A schedule S is ACA if in S T_i reads some data item x from T_j, then T_i reads x after T_j commits.

- Example: $w_1(x)\ \ w_2(x)\ \ w_1(y)\ \ c_1\ \ r_2(y)\ \ c_2$.

Strictness

- If schedules are ACA, the effects of a transaction cannot be undone by replacing before values.

- Assume initially x is 10 and y is 20.

- $w_1(x, 12)$ $w_2(x, 14)$ $w_1(y, 25)$ c_2 a_1

- Replacing current value of x by before value of $w_1(x, 12)$ will result in incorrectness.

- A schedule S is strict, if whenever transaction T_i reads from T_j or T_i overwrites value written by T_j, it does so after T_j commits or aborts.

- If schedules are strict, the abort of the transaction can be implemented by reestablishing the before values.

Discussion

- Should the system ensure executions are strict, ACA, or recoverable?

- How to choose?

  - *Efficiency:* transaction aborts are fairly frequent. The time it takes to abort a transaction should not be very large.

    * Efficient implementation of abort possible for strict and ACA schedules. But not for recoverable schedules.

  - *Simplicity:* The logic for abort should also be simple. For strict schedules, aborts are simple. For ACA the logic gets more complex.

  - *Concurrency:* We want high concurrency. How do these schedules compare in terms of concurrency.

    * Every strict schedule is also ACA.
    * Every ACA schedule is also recoverable.
    * Not every recoverable schedule is ACA.
    * Not every ACA schedule is strict.

    From efficiency of recovery perspective, schedules should be ACA or strict. From concurrency perspective ACA prefered over strict. From simplicity perspective, strict preferred over ACA. The extra concurrency offered by ACA over strict is not worth sacrificing simplicity.

- Commercial DBMS ensure strictness of schedules.

Summary of Concurrency Control

- Concurrent execution of transaction programs may result in 'undesirable' executions.

- The purpose of concurrency control mechanism is to only permit desirable interleavings of the transaction programs.

- Traditionally, desirable interleavings are defined as those that map a consistent database state into another consistent state.

- A state of the database is a mapping of data items to values.

- Consistent states are those in which integrity constraints are preserved.

Summary of Concurrency

- The concurrency control mechanism views the transaction programs as a sequence of read and write operations over the data items.

- Concurrent execution of the transactions results in a schedule.

- In order to ensure that the interleaving of the concurrent transactions results in a consistent database state, the CCM ensures serializability of the schedules.

- A schedule S is serializable, if it is equivalent to some serial execution of the same transactions.

- Different notions of equivalence define different notions of serializability.

- Final state serializability, view serializability, conflict serializability.

Is Preserving Consistency Good Enough?

- We motivated serializability from the need to preserve system consistency, where consistency means preservation of integrity constraints.

- However, is preserving consistency a sufficient guarantee to the users?

- Simply preserving consistency does not prevent read-only transactions from seeing an inconsistent state!

- Even if each transaction 'sees' a consistent state and system consistency is preserved, the execution may still not be acceptable!

Mehrotra et. al. RIDE 1991.