



Non è possibile visualizzare l'immagine.

Chapter 14: Transactions

BY MEANS OF TRANSACTIONS, A DBMS CAN GUARANTEE SOME DESIRED PROPERTIES REGARDING TO FAILURE HANDLING AND CONCURRENT EXECUTION

SET OF OPERATIONS : E. G. MONEY TRANSFER FROM ONE BANK ACCOUNT TO ANOTHER

SINGLE OPERATION FOR A LOGICAL POINT OF VIEW, EXECUTED AS IF IT WAS A SOURCE ATOMIC OPERATION [ACC OR X/NOTHING]

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Outline

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.



Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer \$50 from account A to account B:
 1. **read(A)**
 2. $A := A - 50$
 3. **write(A)**
 4. **read(B)**
 5. $B := B + 50$
 6. **write(B)**

THE READ AND WRITE OPERATIONS,
AS WE SHALL SEE, ARE THE MOST
IMPORTANT FROM OUR POINT OF VIEW
AS THEY ARE THE ONES INTERACTING
WITH THE DATABASE CONTENT
- Two main issues to deal with:
 - Failures of various kinds, such as hardware failures and system crashes
 - Concurrent execution of multiple transactions



Required Properties of a Transaction

- Consider a transaction to transfer \$50 from account A to account B:
 1. **read(A)**
 2. $A := A - 50$
 3. **write(A)**
 4. **read(B)**
 5. $B := B + 50$
 6. **write(B)**
 - **Atomicity requirement** (EITHER ALL OR NOTHING)
 - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
 - ▶ Failure could be due to software or hardware
 - The system should ensure that updates of a partially executed transaction are not reflected in the database
 - **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.
- THERE ARE SOME PROPERTIES THAT MUST BE GUARANTEED DURING TRANSACTION EXECUTION, TO PREVENT UNWANTED BEHAVIOURS FROM HAPPENING IN THE EVENT OF FAILURE OR CONCURRENT ACCESS*



Required Properties of a Transaction (Cont.)

A TRANSACTION STARTS FROM A CONSISTENT DB AND LEAVES IT
IN A CONSISTENT STATE AFTER ITS EXECUTION -

■ Consistency requirement in above example:

- The sum of A and B is unchanged by the execution of the transaction

■ In general, consistency requirements include

- ▶ Explicitly specified integrity constraints such as primary keys and foreign keys
- ▶ Implicit integrity constraints
 - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand

- A transaction, when starting to execute, must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
 - Erroneous transaction logic can lead to inconsistency

THUS, DURING TRANSACTION EXECUTION YOU MAY VIOLATE CONSTRAINTS - THEY HAVE nevertheless TO BE SATISFIED BY THE END OF THE TRANSACTION



Required Properties of a Transaction (Cont.)

- **Isolation requirement** — if between steps 3 and 6 (of the fund transfer transaction) , another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

CONCURRENT TRANSACTIONS SHOULD NOT INTERFERE WITH EACH OTHER

- | T1 | T2 |
|--------------------|------------------------------|
| 1. read(A) | |
| 2. $A := A - 50$ | |
| 3. write(A) | |
| 4. read(B) | read(A), read(B), print(A+B) |
| 5. $B := B + 50$ | |
| 6. write(B) | |

- Isolation can be ensured trivially by running transactions **serially**
 - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

WE WANT TO ALLOW THE EXECUTION OF A CONCURRENT SET OF TRANSACTIONS IN SUCH A WAY THAT THE RESULT OF SUCH CONCURRENT EXECUTION IS THE SAME AS THAT OF A SERIAL EXECUTION OF THE TRANSACTIONS -

THERE CAN BE MANY DIFFERENT SERIAL ORDERINGS, ANY OF THEM IS OK FOR US

SO WE ARE NOT INTERESTED IN THE OUTCOME PER SE - IT JUST NEEDS TO BE ONE OF THE POSSIBLE ONES -

NEED OF SOME HEURISTICS TO DO THAT

- INEFFICIENT \rightarrow THINK ABOUT AN AIRLINE COMPANY
- UNNECESSARY \rightarrow CONSTRAIN THE INTERACTION ONLY BETWEEN CRITICAL TRANSACTIONS



ACID Properties

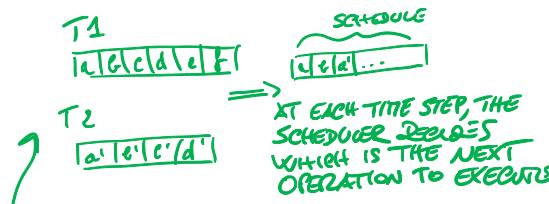
A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

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



Transaction State

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



Schedules

Schedule – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed

- A schedule for a set of transactions must consist of all instructions of those transactions
- Must preserve the order in which the instructions appear in each individual transaction. *IN THE EXAMPLE ABOVE, I CANNOT HAVE "e" COMING BEFORE "a"*
- A transaction that successfully completes its execution will have a **commit** instructions as the last statement
 - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

At commit instant, ALL OPERATIONS MADE BY THE TRANSACTION ON THE DATABASE BECOME PERMANENT



Schedule 1

- Let T_1 transfer \$50 from A to B , and T_2 transfer 10% of the balance from A to B .
- An example of a **serial** schedule in which T_1 is followed by T_2 :

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



Schedule 2

- A **serial** schedule in which T_2 is followed by T_1 :

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

OF COURSE, THEY
GIVE RISE TO
TWO DIFFERENT
RESULTS -



Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is **equivalent** to Schedule 1.

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

Note -- In schedules 1, 2 and 3, the sum “ $A + B$ ” is preserved.



Schedule 4

- The following concurrent schedule does not preserve the sum of “ $A + B$ ”

T_1	T_2
read (A) $A := A - 50$	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) ↑ <i>Variable is still the original value, without the subtraction</i>
write (A) read (B) $B := B + 50$ write (B) commit	$B := B + temp$ write (B) commit ↑ <i>The result cannot be obtained with any serial execution of T_1 and T_2 –</i> WE WANT A WAY TO ACCEPT SCHEDULE 3 AND REJECT SCHEDULE 4 ... A FAIRLY EASY WAY, COMPUTATIONALLY SPEAKING –



Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. **conflict serializability**

THIS WILL BE OUR HEURISTIC

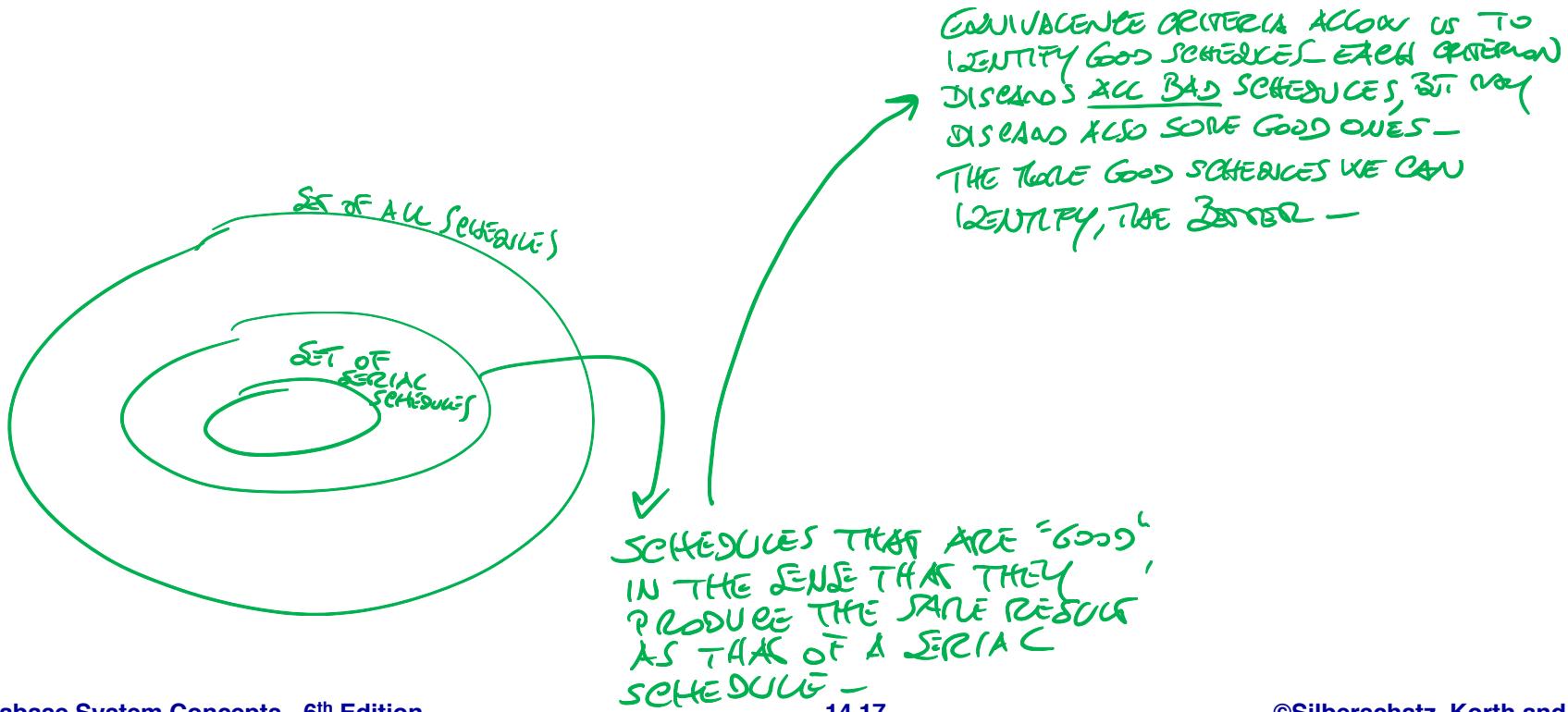


INSTEAD OF HAVING TO
COMPARE OUR RESULT
WITH THAT OF ANY
POSSIBLE SERIAL SCHEDULE
OF THE TRANSACTIONS



Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.





Conflicting Instructions

- Let I_i and I_j be two instructions of transactions T_i and T_j respectively. Instructions I_i and I_j **conflict** if and only if there exists some item Q accessed by both I_i and I_j , and at least one of these instructions wrote Q .
 1. $I_i = \text{read}(Q)$, $I_j = \text{read}(Q)$. I_i and I_j don't conflict.
 2. $I_i = \text{read}(Q)$, $I_j = \text{write}(Q)$. They conflict.
 3. $I_i = \text{write}(Q)$, $I_j = \text{read}(Q)$. They conflict
 4. $I_i = \text{write}(Q)$, $I_j = \text{write}(Q)$. They conflict
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
 - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

"Q" STANDS FOR AN ITEM, WHICH IS A GENERAL CONCEPT \Rightarrow CAN BE COLUMN, & ROW, A TABLE, ...



Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule



Conflict Serializability (Cont.)

Prns CONFLICT = { (R₁(X), W₂(A)), (W₁(X), R₂(X)), (W₁(X), W₂(A)), (R₁(B), W₂(B)), (W₁(B), R₂(B)), (W₁(B), W₂(B)) }

- Schedule 3 can be transformed into Schedule 6 -- a serial schedule where T_2 follows T_1 , by a series of swaps of non-conflicting instructions. Therefore, Schedule 3 is conflict serializable.

CONFlict GRAPH:

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

Schedule 3

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

Schedule 6

THUS, SCHEDULE 3 IS A "GOOD" ONE, SINCE IT CAN BE TRANSFORMED INTO SCHEDULE 6, WHICH IS SERIAL, AND THE FINAL RESULT OF THE TWO SCHEDULES IS THE SAME -



Conflict Serializability (Cont.)

$$\text{Conflict} = \left\{ (R_3(Q), W_1(Q)), (W_1(Q), R_3(Q)) \right\}$$

- Example of a schedule that is not conflict serializable:

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



- We are unable to swap instructions in the above schedule to obtain either the serial schedule $\langle T_3, T_4 \rangle$, or the serial schedule $\langle T_4, T_3 \rangle$.

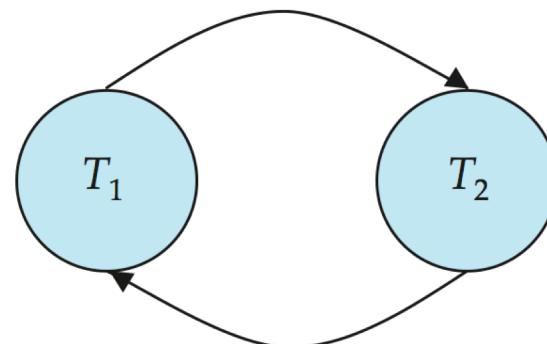
How can we establish if a given schedule is serializable or not w.r.t. conflicts without doing these thousands analyses based on swaps?

→ Precedence Graph



Precedence Graph

- Consider some schedule of a set of transactions T_1, T_2, \dots, T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names). ↳ Notes
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example**





Testing for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
 - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
 - That is, a linear order consistent with the partial order of the graph.

OPERATIONS
THEY NAME IS
W THE ORDER
OF # OF
VERTICES SQUARED

THERE ARE
OTHER NOTIONS
OF SERIALIZABILITY
OTHER THAN THE
ONE BASED ON
CONFLICTS

E.G. VIEW
SERIALIZABILITY

CAN BECOME TOUGH
GOOD SCHEDULES THAN
CSR, HOWEVER IT IS
EVEN TOUGH

