Database Systems and Transactions

- Database
  - concurrent access to shared data
  - DB state defined in terms of the data values: not static, dynamic
- DB correctness: consistency
  - internal consistency (semantic integrity)
  - mutual consistency
  - cannot be enforced at each action
- Transaction
  - partially ordered set of operations
  - a complete and consistent computation
  - atomicity, consistency, isolation, durability (ACID)
  - scheduler synchronizes concurrent operations

Database System Model

- Functional decomposition: abstract model
  - integrity checker
  - transaction manager (TM)
  - scheduler
  - data manager (DM)
    - recovery manager (RM)
    - cache manager (CM)
- Transaction manager
  - transaction_id, participant selection
- Scheduler
  - ordering execution
  - actions: execute, reject, delay
  - concurrency control techniques
  - serializability and recoverability

Database System Model (cont’d)

- Data manager
  - operates directly on the database and responsible for transaction termination
  - RM and CM
- Recovery manager
  - atomicity
  - resilient to failures: transaction, system, media
  - operations: start, commit, abort, read, write
- Cache manager
  - manage data movement interactions between volatile and stable storage
  - actions: fetch and flush

Transaction

- Transaction concept
  - a unit of program execution
  - consists of several operations to access/update data
  - ACID: atomicity, consistency, isolation, durability
- Consistency
  - execution in isolation must preserve DB consistency
- Atomicity
  A transaction is atomic if all actions are completed or none is performed, and intermediate states are not visible to other transactions.
  - implies a particular ordering on a given set of events
  - in principle, to preserve consistency, actions belong to the same transaction must remain atomic
**Transaction**

- Isolation
  - even if multiple T's executed concurrently, each should be unaware of other T's executing concurrently

- Durability
  - when T completes successfully, the changes it made must persist, even with system failures

- Correctness of concurrent execution
  - schedule: an execution history
  - serial execution: inefficient
  - interleaving operations of transactions as much as possible for performance
  - some interleaved schedules are equivalent to serial schedules: serializable execution

**Serializable Execution**

\(<ex> A = \{a_1(X), a_2(Y)\} \quad B = \{b_1(X), b_2(Y)\}\n
System requires either A → B or B → A for all operations (ai → bi or bi → ai for all i) to satisfy atomicity requirement for some ordering relationship (→)

a_1 a_2 b_1 b_2 \equiv a_1 b_1 a_2 b_2 \equiv a_1 b_1 b_2 a_2

Why? The ordering a_1 b_1 a_2 b_2 preserves the atomicity but the ordering a_1 b_1 b_2 a_2 does not.

- ordering actions serves the purpose of implementing atomic operations so as to preserve the consistency of the system state
- system may execute a set of transactions in any order as long as the effect is the same as that of some serial order
- if user wants a specific order, (s)he should enforce it (e.g., submitting T_2 after T_1 is committed)

**Serializability**

- Correctness criterion
  - serializability: correctness definition in DBS
  - all serializable executions are equally correct
  - scheduling algorithms enforce a partial/total ordering
  - in distributed systems, variable delays may disturb any particular ordering which is supposed to occur

- Equivalent execution
  - two schedules (executions) are equivalent if
    1) every read operation reads from the same write in both schedules
    2) both schedules have the same final writes

- Serialization graph
  - dependency graph, showing precedence relationship
  - serializability theorem

**Equivalent Execution**

\[
T_1 = r_1(x)r_1(z)w_1(x) \\
T_2 = r_2(y)r_2(z)w_2(y) \\
T_3 = w_3(x)r_3(y)w_3(z)
\]

\[
H_1 = w_3(x)r_3(y)r_3(y)w_3(z)r_1(x)r_2(z)w_2(y)r_1(x)w_1(x)
\]

Precedence relationship: \( T_3 \rightarrow T_1 \quad T_1 \rightarrow T_2 \)

\[
H_2 = w_3(x)r_3(y)w_3(z)r_2(y)r_2(z)w_2(y)r_3(x)r_3(z)w_1(x)
\]

Precedence relationship: \( T_3 \rightarrow T_2 \rightarrow T_1 \)

- \( H_2 \) is a serial execution.
- \( H_1 \) is equivalent to \( H_2 \).
- \( H_1 \) is a serializable execution.
Conflict and View Serializability

- **Conflict serializability**
  conflicting operations are ordered in the same way as in some serial execution
  --- topological sorting of the serialization graph

- **Topological sorting of SG(H)**
  sequence of all nodes in SG(H) such that if T_i appears before T_j in the sequence, there is no path from T_j to T_i in SG(H)
  
  \[ H = w_1(x) w_2(y) r_1(x) r_2(y) w_3(x) w_4(y) \]

  \[ \text{SG(H):} \]
  
  \[
  \begin{align*}
  T_1 & \rightarrow T_2 \\
  & \rightarrow T_3 \\
  T_1 & \rightarrow T_2 \rightarrow T_3 \\
  T_1 & \rightarrow T_3 \rightarrow T_2
  \end{align*}
  \]

- **View serializability**
  an execution is view serializable if it is view equivalent to some serial execution

- **View equivalence of H_1 and H_2**
  for the same set of transactions, if T_i reads x from T_j in H_i, then T_j reads x from T_i in H_2 (same reads-from relationship), and for each data object x, if w_i(x) is the final write on x in H_1, then it is also the final write in H_2 (same final write)
  
  \[ H = w_1(x) w_2(x) w_3(y) w_4(x) w_5(y) w_6(y) w_7(x) \]
  
  --- H is view serializable, but not conflict serializable

Properties of Schedules

- **Recoverability**
  - required to ensure that aborting a transaction does not change the semantics of committed ones
    \[ w_1(x) \ r_2(x) \ w_2(y) \ c_2 \]
  - not recoverable: what if T_1 aborts?
  - recoverable execution depends on commit order
  - T cannot commit until all values it read are guaranteed not to be aborted: delaying commit
  - cascaded abort is sometime mandatory
    \[ w_1(x) \ r_2(x) \ w_2(y) \ a_1 \]

- **Avoiding cascaded aborts**
  - achieved if every transaction reads only values written by committed transactions
  - must delay each r(x) until all transactions that issued w(x) is either committed or aborted

- **Restoring before images**
  - implementing transaction abort by simply restoring before images of all writes is very convenient
    \[ w_1(x) \ w_2(x) \ a_1 \ a_2 \]
  - value of x must be restored to the initial value, not the value written by T_1
  - solution: delay w(x) until all transactions that have written x are either committed or aborted

- **Strictness**
  - executions that satisfy both requirements
  - delay both r(x) and w(x) until all transactions that have written x are either committed or aborted
    \[ w_1(x) \ w_1(y) \ w_2(z) \ c_1 \ r_2(x) \ a_2 \]
Properties of Synchronization

- **Recoverability (RC)**
  - reads-from relationships
  - RC if $T_i$ reads from $T_j$ ($i=j$) and $c_i \in H$, then $c_j < c_i$

- **Avoiding cascaded aborts (ACA)**
  - ACA if $T_i$ reads from $T_j$ ($i=j$) then $c_j < r_i[x]$

- **Strictness (ST)**
  - strict if whenever $w_i[x] < o_i[x]$ ($i=j$)
    then either $a_j < o_i[x]$ or $c_j < o_i[x]$

Relationships among Synchronization Properties

- **Theorem:** ST < ACA < RC