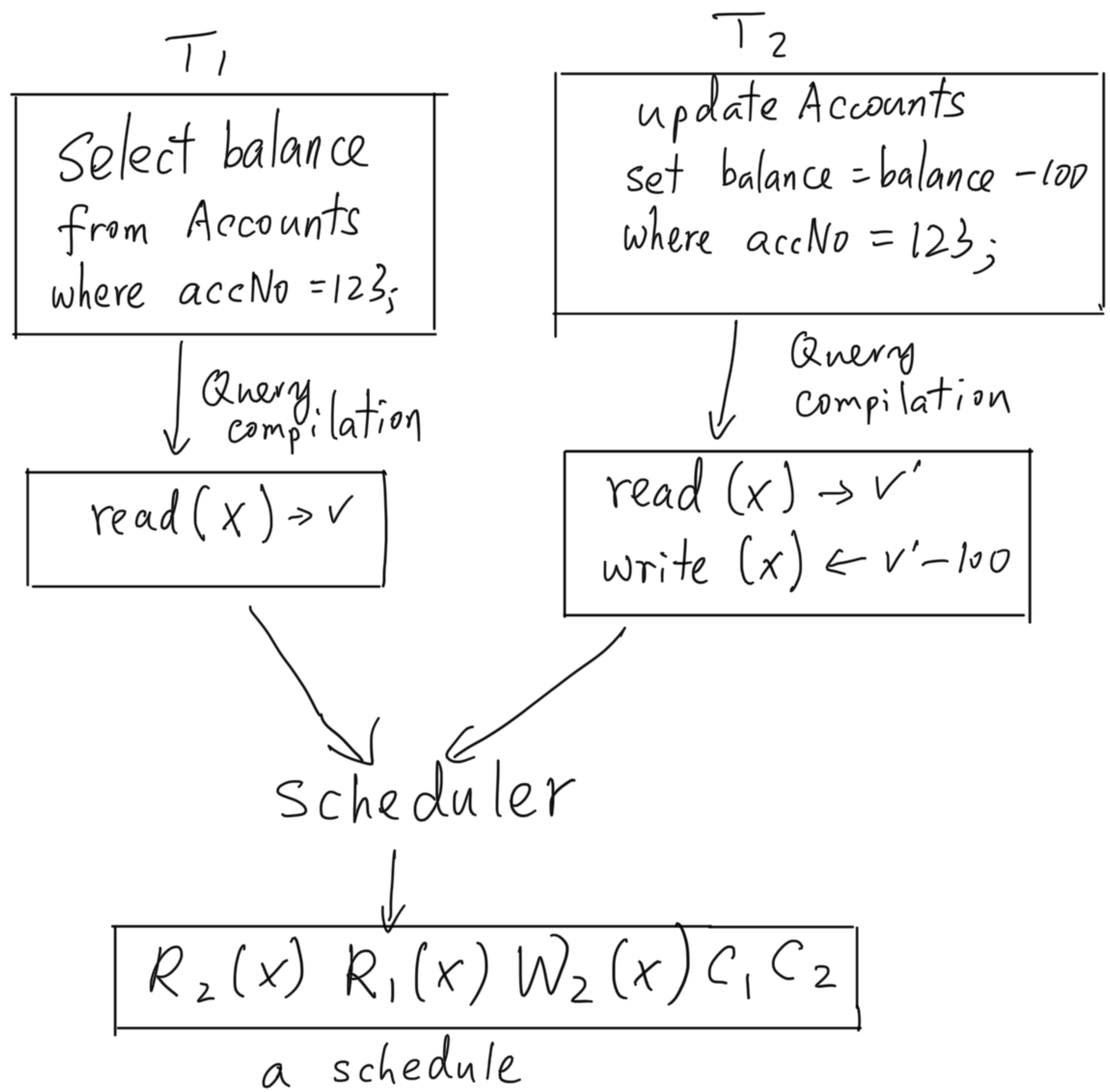
Database Management Systems

Transaction Management

Schedules

General Process



Scheduler

- takes read/write requests from transactions
- for each request, takes one of the following actions:
    - execute
    - delay
    - ignore
    - reject
- the output from the scheduler is called a schedule.
- Scheduler should not need to understand the transaction semantics. The conventional assumption for the scheduler is:
  Any database element that a transaction T writes is given a value that depends on the database state in such a way that no arithmetic coincidences occur.

Schedule

- Notation
  - $R_i(x)$: transaction T_i reads object x
  - $W_i(x)$: transaction T_i writes object x
  - C_i: transaction T_i commits
  - A_i: transaction T_i aborts
- A transaction T_i is a sequence of operations
- A schedule S for a set of transactions $T_1,\ldots,T_k$ includes every operation $O_i \in T_i$ and these operations are ordered the same way as in T_i

Correctness

- When a database server processes several concurrent transactions, it must appear as if the transactions have been executed sequentially (in some/any order).

- If a database server really processes those transactions sequentially, the generated schedule is called a serial schedule. A serial schedule must be correct.

- If transaction T_i appears to precede T_j, then it means that T_j will "see" all the updates done by T_i, and T_i will not see any updates done by T_j.

Equivalent Schedules

- If two schedules are equivalent, they are equivalent on any database instances. (Note: think about the database instances as test cases.)
- The same principle holds for queries. If a query is right, it will return the right data on any database instances.
- Two operations are conflict if they
  - belong to different transactions
  - access the same database object
  - at least one of them is a write operation
- Two schedules are conflict equivalent if every pair of conflicting operations are ordered the same way in both schedules.

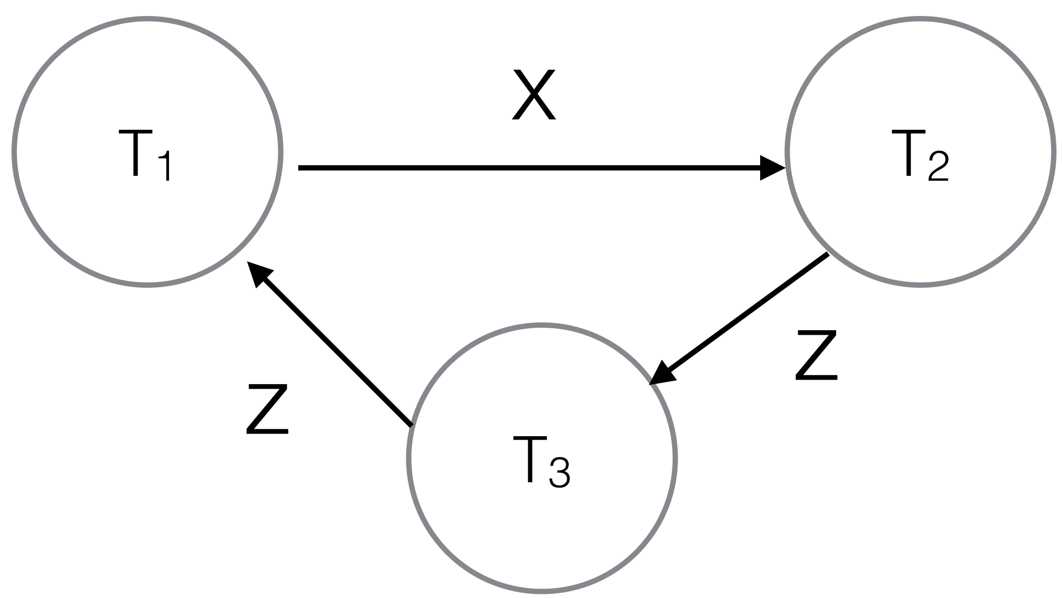
Conflict Serializability

- If a schedule is conflict equivalent to a serial schedule, then the schedule is a conflict serializable schedule.
- A conflict serializable schedule is guaranteed to preserve computational effects.
- We use serialization (precedence) graph to test whether a schedule is conflict serializable.
    - A serialization (Precedence) graph SG(S) for a schedule S is a directed graph with nodes labeled by transactions, and an edge from T_i to T_j is in SG(S) if and only if $O_i[x]$ precedes $O_j[x]$ in S where $O_i[x]$ and $O_j[x]$ are conflicting operations.
    - Theorem: A schedule S is conflict serializable if and only if SG(S) is an acyclic graph.

Example (I)

$R_1(x)R_2(y)R_1(y)R_2(z)R_3(y)W_3(z)W_2(x)R_1(z)$

x: R_1W_2
y: $R_2R_1R_3$
z: $R_2W_3R_1$

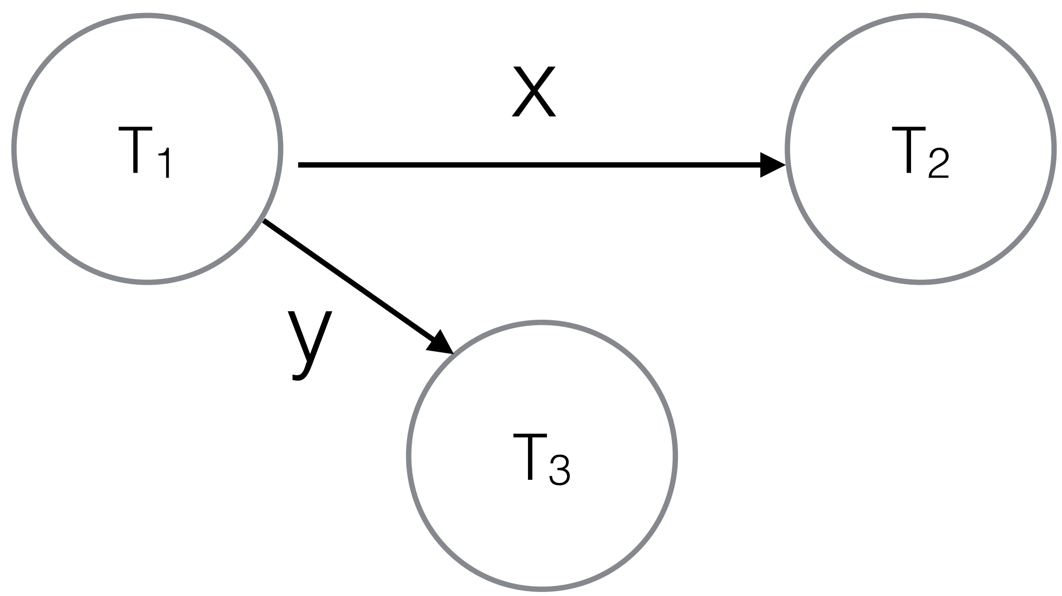


Cyclic graph —> not conflict serializable

Example (II)

$R_1(x)R_1(y)W_1(x)R_2(x)R_3(y)R_1(z)W_3(y)W_1(z)W_2(x)$

x: $R_1W_1R_2W_2$
y: $R_1R_3W_3$
z: R_1W_1



Acyclic graph —> conflict serializable
Conflict equivalent to both the serial schedules
$T_1T_2T_3$ and $T_1T_3T_2$

Other Properties of Schedules

- Conflict serializable schedule says nothing about how each transaction in the schedule would terminate itself.

- Recoverable Schedules (may need cascading rollback/abort): A schedule is recoverable if each transaction commits only after each transaction from which it has read data has committed.

- Cascadeless Schedules: A schedule avoids cascading rollback if transactions may read only values written by committed transactions (no reading dirty value).

Example

- Recoverable Schedule (E means either Commit or Abort):
  $R_1(x)R_1(y)W_1(x)R_2(x)R_3(y)R_1(z)W_3(y)W_1(z)W_2(x)\underline{E_1E_3E_2}$
  - the order of E_2 and E_3 doesn't matter
  - If T1 decides to abort, then T2 must abort itself too because T2 read the dirty data (x) that should be there (written by T1).
- Cascadeless Schedule (again, E means either Commit or Abort):
  $R_1(x)R_1(y)W_1(x)R_3(y)R_1(z)W_3(y)W_1(z)\underline{E_1R_2(x)}W_2(x)\underline{E_3E_2}$
  - If T1 commits, then T2 would read the x value written by T1 (a committed transaction).
  - If T1 aborts, then T2 would read the original x value (before touched by T1).
  - Either way, T2 can decide to commit or abort without ever consider how T1 ends itself.