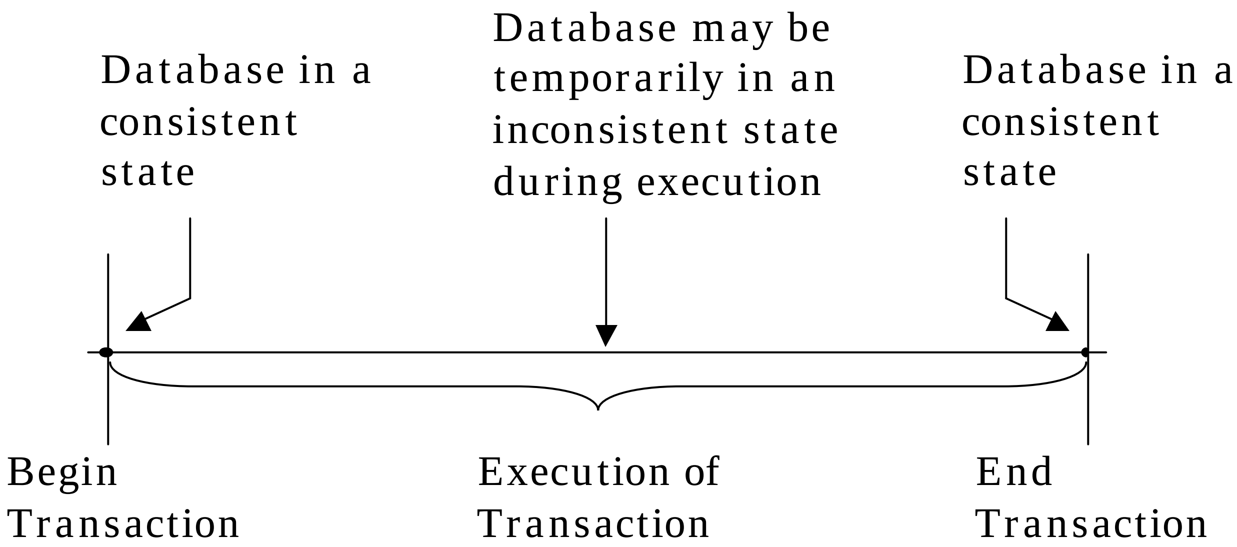
Transaction

- A transaction is a collection of actions that make consistent transformations of system states while preserving system consistency.
  - concurrency transparency
  - failure transparency



Transaction Example – A Simple SQL Query

```
…
main() {
  …
  EXEC SQL UPDATE Project
     SET Budget = Budget * 1.1
     WHERE Pname = `CAD/CAM';
  EXEC SQL COMMIT RELEASE;
  return(0);
  …}
```

Example Database

Consider an airline reservation example with the relations:

FLIGHT(<u>FNO, DATE</u>, SRC, DEST, STSOLD, CAP)

CUST(<u>CNAME</u>, ADDR, BAL)

FC(<u>FNO, DATE, CNAME</u>,SPECIAL)

Example Reservation Transaction

```
…
main {
…
  EXEC SQL BEGIN DECLARE SECTION;
     char flight_no[6], customer_name[20];
     char day;
  EXEC SQL END DECLARE SECTION;
  scanf(flight_no, day, customer_name);
  EXEC SQL  UPDATE FLIGHT
     SET    STSOLD = STSOLD + 1
     WHERE  FNO = :flight_no AND DATE = :day;
  EXEC SQL  INSERT
     INTO   FC(FNO, DATE, CNAME, SPECIAL);
     VALUES(:flight_no,:day,:customer_name, null);
  printf("Reservation completed");
  EXEC SQL COMMIT RELEASE;
  return(0);}
```

Termination of Transactions

```
…
main {
…
  EXEC SQL BEGIN DECLARE SECTION;
     char flight_no[6], customer_name[20];
     char day; int temp1, temp2;
  EXEC SQL END DECLARE SECTION;
  scanf(flight_no, day, customer_name);
  EXEC SQL   SELECT STSOLD,CAP INTO :temp1,:temp2
      FROM   FLIGHT
      WHERE  FNO = :flight_no AND DATE = :day;
  if temp1 = temp2 then {
      printf("no free seats");
      EXEC SQL ROLLBACK RELEASE;
      return(-1);}
  else {
      EXEC SQL UPDATE FLIGHT
        SET    STSOLD = STSOLD + 1
        WHERE FNO = :flight_no AND DATE = :day;
      EXEC SQL INSERT
        INTO   FC(FNO, DATE, CNAME, SPECIAL);
        VALUES (:flight_no, :day, :customer_name, null);
      EXEC SQL COMMIT RELEASE;
      printf("Reservation completed");
      return(0);}
}
```

Characterization

- Read set (RS)
  - The set of data items that are read by a transaction
- Write set (WS)
  - The set of data items whose values are changed by this transaction
- Base set (BS)
  - $RS \cup WS$

Formalization

Let

- $o_{ij}(x)$ be some operation o_j of transaction T_i operating on data item x, where $o_j \in$ {read,write} and o_j is atomic
- $OS_i = \cup_j \, o_{ij}$
- $N_i \in$ {abort,commit}

Transaction T_i is a partial order $T_i = \{\Sigma_i, <_i\}$ where

❶ $\Sigma_i = OS_i \cup \{N_i\}$

❷ For any two operations $o_{ij}, o_{ik} \in OS_i$, if $o_{ij} = R(x)$ and $o_{ik}=W(x)$ for any data item x, then either $o_{ij} <_i o_{ik}$ or $o_{ik} <_i o_{ij}$

❸ $\forall o_{ij} \in OS_i, \; o_{ij} <_i N_i$

Example

Consider a transaction T:

Read(x)

Read(y)

$x \leftarrow x + y$

Write(x)

Commit

Then

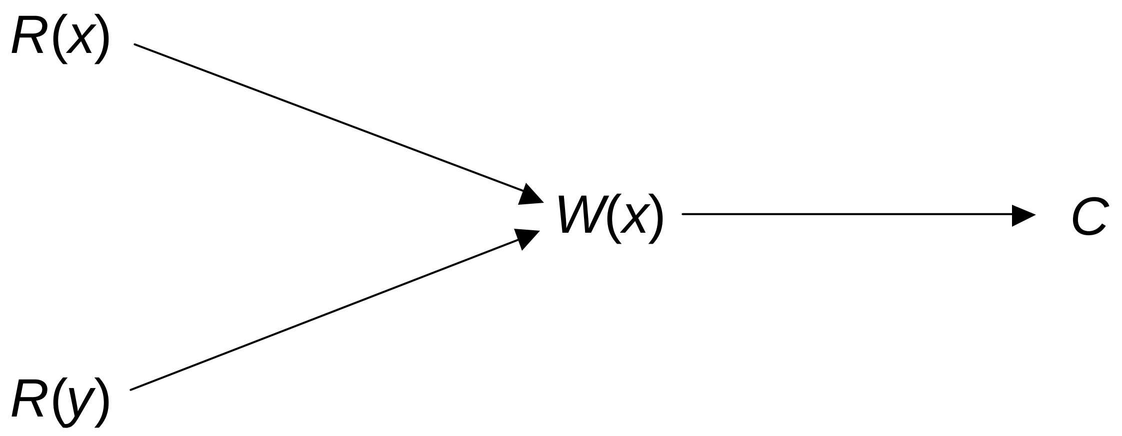
$\Sigma = \{R(x), R(y), W(x), C\}$

$< \; = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$

DAG Representation

Assume

$< = \{(R(x), W(x)), (R(y), W(x)), (R(x), C), (R(y), C), (W(x), C)\}$



Properties of Transactions

ATOMICITY

- all or nothing

CONSISTENCY

- no violation of integrity constraints

ISOLATION

- concurrent changes invisible $\Rightarrow$ serializable

DURABILITY

- committed updates persist

Atomicity

- Either all or none of the transaction's operations are performed.
- Atomicity requires that if a transaction is interrupted by a failure, its partial results must be undone.
- The activity of preserving the transaction's atomicity in presence of transaction aborts due to input errors, system overloads, or deadlocks is called transaction recovery.
- The activity of ensuring atomicity in the presence of system crashes is called crash recovery.

Consistency

- Internal consistency
  - A transaction which executes *alone* against a *consistent* database leaves it in a consistent state.
  - Transactions do not violate database integrity constraints.
- Transactions are correct programs

Isolation

- Serializability
  - If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order.
- Incomplete results
  - An incomplete transaction cannot reveal its results to other transactions before its commitment.
  - Necessary to avoid cascading aborts.

Isolation Example

- Consider the following two transactions:

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x+1$		$x \leftarrow x+1$
	Write(x)		Write(x)
	Commit		Commit

- Possible execution sequences:

T_1:	Read(x)	T_1:	Read(x)
T_1:	$x \leftarrow x+1$	T_1:	$x \leftarrow x+1$
T_1:	Write(x)	T_2:	Read(x)
T_1:	Commit	T_1:	Write(x)
T_2:	Read(x)	T_2:	$x \leftarrow x+1$
T_2:	$x \leftarrow x+1$	T_2:	Write(x)
T_2:	Write(x)	T_1:	Commit
T_2:	Commit	T_2:	Commit

Consistency Degrees (due to Jim Gray)

- Degree 0
  - Transaction *T* does not overwrite dirty data of other transactions
  - Dirty data refers to data values that have been updated by a transaction prior to its commitment
- Degree 1
  - *T* does not overwrite dirty data of other transactions
  - *T* does not commit any writes before EOT

Consistency Degrees (cont'd) (due to Jim Gray)

- Degree 2
  - *T* does not overwrite dirty data of other transactions
  - *T* does not commit any writes before EOT
  - *T* does not read dirty data from other transactions
- Degree 3
  - *T* does not overwrite dirty data of other transactions
  - *T* does not commit any writes before EOT
  - *T* does not read dirty data from other transactions
  - Other transactions do not dirty any data read by *T* before *T* completes.

SQL-92 Isolation Levels

Phenomena:

- Dirty read
  - T_1 modifies x which is then read by T_2 before T_1 terminates; T_1 aborts $\Rightarrow$ T_2 has read value which never exists in the database.
- Non-repeatable (fuzzy) read
  - T_1 reads x; T_2 then modifies or deletes x and commits. T_1 tries to read x again but reads a different value or can't find it.
- Phantom
  - T_1 searches the database according to a predicate while T_2 inserts new tuples that satisfy the predicate.

SQL-92 Isolation Levels (cont'd)

- Read Uncommitted
  - For transactions operating at this level, all three phenomena are possible.
- Read Committed
  - Fuzzy reads and phantoms are possible, but dirty reads are not.
- Repeatable Read
  - Only phantoms possible.
- Anomaly Serializable
  - None of the phenomena are possible.

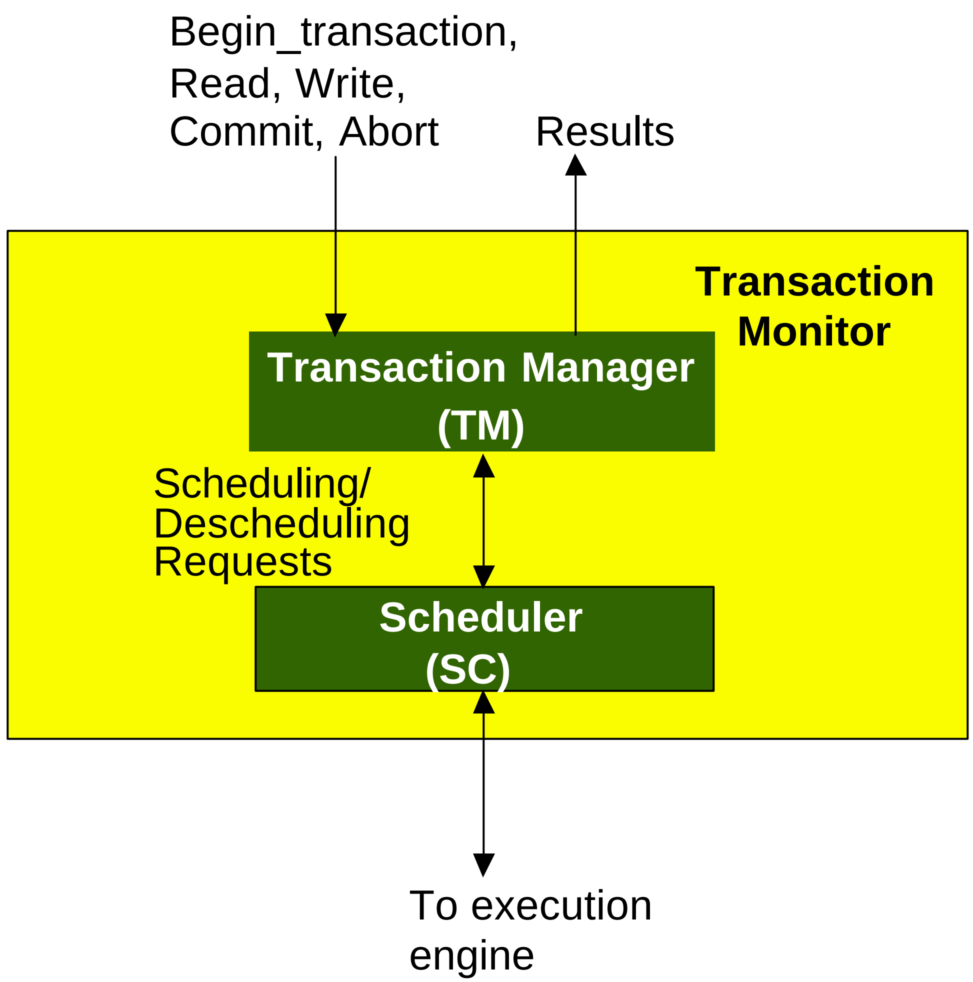
Durability

- Once a transaction commits, the system must guarantee that the results of its operations will never be lost, in spite of subsequent failures.
- Database recovery

Transactions Provide…

- *Atomic* and *reliable* execution in the presence of failures
- *Correct* execution in the presence of multiple user accesses
- Correct management of *replicas* (if they support it)

Architecture



Transaction Execution