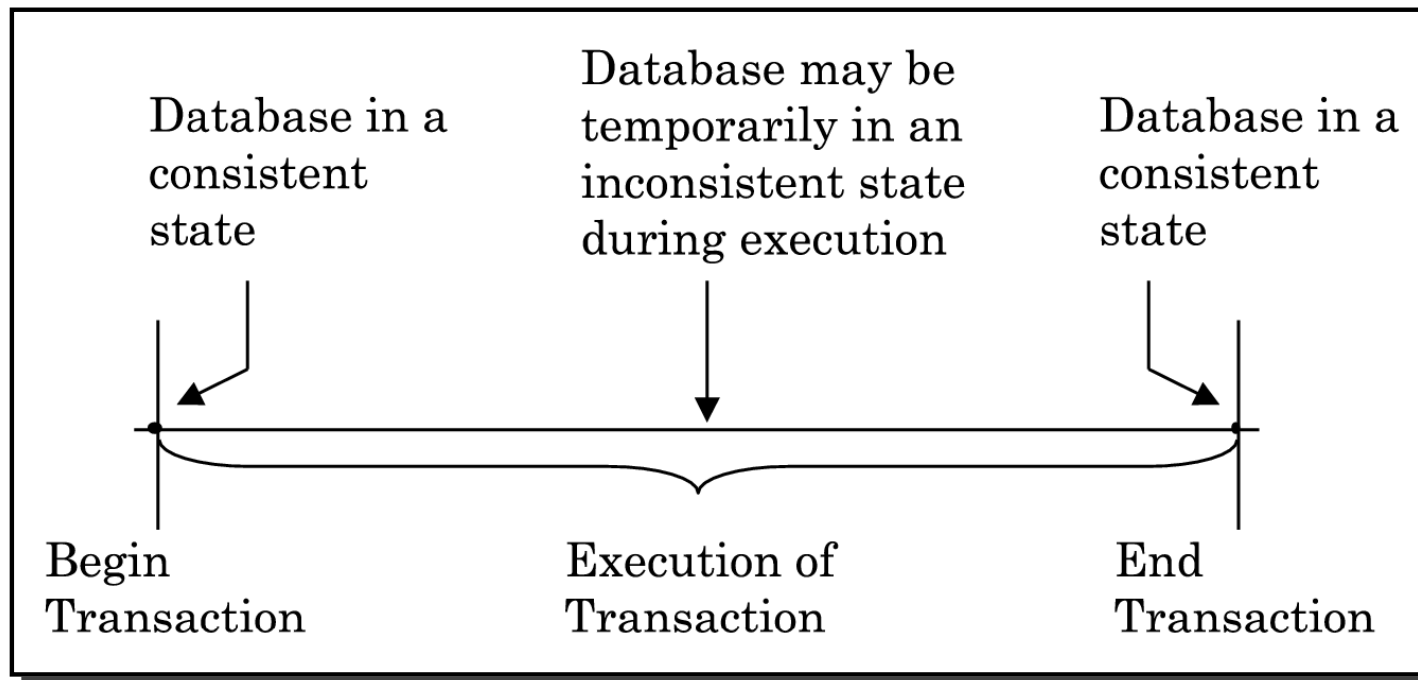

Chapter 8: Introduction to Transaction Management

- Definition and Examples
- Properties
- Classification
- Processing Issues

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Definition

- **Transaction:** A collection of actions that transforms the DB from one consistent state into another consistent state; during the execution the DB might be inconsistent.



- **States** of a transaction
 - **Active:** Initial state and during the execution
 - **Paritally committed:** After the final statement has been executed
 - **Committed:** After successful completion
 - **Failed:** After the discovery that normal execution can no longer proceed
 - **Aborted:** After the transaction has been rolled back and the DB restored to its state prior to the start of the transaction. Restart it again or kill it.

Example

- **Example:** Consider an SQL query for increasing by 10% the budget of the CAD/CAM project. This query can be specified as a transaction by providing a name for the transaction and inserting a begin and end tag.

```
Transaction BUDGET_UPDATE  
begin  
  EXEC SQL  
  UPDATE PROJ  
  SET     BUDGET = BUDGET * 1.1  
  WHERE   PNAME = "CAD/CAM"  
end.
```

Example ...

- **Example:** Consider an airline DB with the following relations:

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)

CUST(CNAME, ADDR, BAL)

FC(FNO, DATE, CNAME, SPECIAL)

- Consider the reservation of a ticket, where a travel agent enters the flight number, the date, and a customer name, and then asks for a reservation.

Begin_transaction Reservation

begin

```
input(flight_no, date, customer_name);
```

```
EXEC SQL UPDATE FLIGHT
```

```
    SET    STSOLD = STSOLD + 1
```

```
    WHERE FNO = flight_no AND DATE = date;
```

```
EXEC SQL INSERT
```

```
    INTO    FC(FNO, DATE, CNAME, SPECIAL);
```

```
    VALUES (flight_no, date, customer_name, null);
```

```
output("reservation completed")
```

end.

Example ...

- **Example (contd.):** A transaction always terminates – commit or abort. Check the availability of free seats and terminate the transaction appropriately.

```
Begin_transaction Reservation
begin
  input(flight_no, date, customer_name);
  EXEC SQL SELECT STSOLD,CAP
    INTO          temp1,temp2
    FROM          FLIGHT
    WHERE         FNO = flight_no AND DATE = date;
  if temp1 = temp2 then
    output("no free seats");
    Abort
  else
    EXEC SQL UPDATE FLIGHT
      SET STSOLD = STSOLD + 1
      WHERE FNO = flight_no AND DATE = date;
    EXEC SQL INSERT
      INTO          FC(FNO, DATE, CNAME, SPECIAL);
      VALUES (flight_no, date, customer_name, null);
    Commit
    output("reservation completed")
  endif
end.
```

Example ...

- Transactions are mainly characterized by its Read and Write operations
 - Read set (RS): The data items that a transaction reads
 - Write set (WS): The data items that a transaction writes
 - Base set (BS): the union of the read set and write set

- **Example (contd.):** Read and Write set of the “Reservation” transaction

RS[Reservation] = { FLIGHT.STSOLD, FLIGHT.CAP }

WS[Reservation] = { FLIGHT.STSOLD, FC.FNO, FC.DATE,
FC.CNAME, FC.SPECIAL }

BS[Reservation] = { FLIGHT.STSOLD, FLIGHT.CAP,
FC.FNO, FC.DATE, FC.CNAME, FC.SPECIAL }

Formalization of a Transaction

- We use the following notation:
 - T_i be a transaction and x be a relation or a data item of a relation
 - $O_{ij} \in \{R(x), W(x)\}$ be an atomic *read/write* operation of T_i on data item x
 - $OS_i = \bigcup_j O_{ij}$ be the set of all operations of T_i
 - $N_i \in \{A, C\}$ be the termination operation, i.e., *abort/commit*
- Two operations $O_{ij}(x)$ and $O_{ik}(x)$ on the same data item are in **conflict** if at least one of them is a *write* operation
- A **transaction** T_i is a **partial order** over its operations, i.e., $T_i = \{\Sigma_i, \prec_i\}$, where
 - $\Sigma_i = OS_i \cup N_i$
 - For any $O_{ij} \in \{R(x) \vee W(x)\}$ and $O_{ik} = W(x)$, either $O_{ij} \prec_i O_{ik}$ or $O_{ik} \prec_i O_{ij}$
 - $\forall O_{ij} \in OS_i (O_{ij} \prec_i N_i)$
- Remarks
 - The partial order \prec is given and is actually application dependent
 - It has to specify the **execution order** between the conflicting operations and between all operations and the termination operation

- **Example:** Consider the following transaction T

```
Read(x)
Read(y)
x ← x + y
Write(x)
Commit
```

- The transaction is formally represented as

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

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

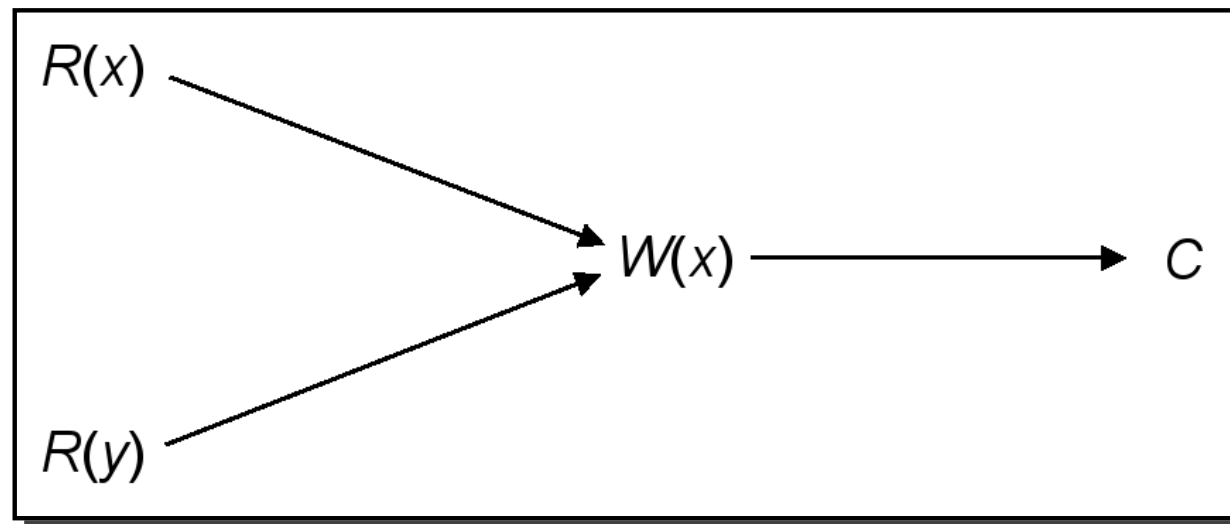
Formalization of a Transaction ...

- **Example (contd.):** A transaction can also be specified/represented as a directed acyclic graph (DAG), where the vertices are the operations and the edges indicate the ordering.

– Assume

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

– The DAG is



Formalization of a Transaction ...

- **Example:** The reservation transaction is more complex, as it has two possible termination conditions, but a transaction allows only one
 - BUT, a transaction is the **execution** of a program which has obviously only one termination
 - Thus, it can be represented as two transactions, one that aborts and one that commits

Transaction T1:

$$\Sigma = \{R(STSOLD), R(CAP), A\}$$

$$\prec = \{(R(STSOLD), A), (R(CAP), A)\}$$

Transaction T2:

$$\Sigma = \{R(STSOLD), R(CAP), \\ W(STSOLD), W(FNO), W(DATE), \\ W(CNAME), W(SPECIAL), C\}$$

$$\prec = \{(R(STSOLD), W(STSOLD)), \dots\}$$

```
Begin_transaction Reservation
begin
  input(flight_no, date, customer_name);
  EXEC SQL SELECT STSOLD,CAP
    INTO      temp1,temp2
    FROM      FLIGHT
    WHERE     FNO = flight_no AND DATE = date;
  if temp1 = temp2 then
    output("no free seats");
    Abort
  else
    EXEC SQL UPDATE FLIGHT
      SET   STSOLD = STSOLD + 1
      WHERE FNO = flight_no AND DATE = date;
    EXEC SQL INSERT
      INTO   FC(FNO, DATE, CNAME, SPECIAL);
      VALUES (flight_no, date, customer_name, null);
    Commit
    output("reservation completed")
  endif
end.
```

Properties of Transactions

- The **ACID** properties

- **A**tomicity

- * A transaction is treated as a single/atomic unit of operation and is either executed completely or not at all

- **C**onsistency

- * A transaction preserves DB consistency, i.e., does not violate any integrity constraints

- **I**solation

- * A transaction is executed as if it would be the only one.

- **D**urability

- * The updates of a committed transaction are permanent in the DB

- **Atomicity**

- Either **all or none** of the transaction's operations are performed
- Partial results of an interrupted transactions must be undone
- **Transaction recovery** is the activity of the restoration of atomicity due to input errors, system overloads, and deadlocks
- **Crash recovery** is the activity of ensuring atomicity in the presence of system crashes

- **Consistency**

- The consistency of a transaction is simply its correctness and ensures that a transaction transforms a consistent DB into a consistent DB
- Transactions are **correct** programs and do not violate database integrity constraints
- **Dirty data** is data that is updated by a transaction that has not yet committed
- Different **levels of DB consistency** (by Gray et al., 1976)
 - * Degree 0
 - Transaction T does not overwrite dirty data of other transactions
 - * Degree 1
 - Degree 0 + T does not commit any writes before EOT
 - * Degree 2
 - Degree 1 + T does not read dirty data from other transactions
 - * Degree 3
 - Degree 2 + Other transactions do not dirty any data read by T before T completes

- **Isolation**

- Isolation is the property of transactions which requires each transaction to see a consistent DB at all times.
- If two concurrent transactions access a data item that is being updated by one of them (i.e., performs a **write** operation), it is not possible to guarantee that the second will read the correct value
- Interconsistency of transactions is obviously achieved if transactions are executed serially
- Therefore, if several transactions are executed concurrently, the result must be the same as if they were executed serially in some order (→ serializability)

Properties of Transactions ...

- **Example:** Consider the following two transactions, where initially $x = 50$:

```
T1: Read(x)
    x ← x+1
    Write(x)
    Commit
```

```
T2: Read(x)
    x ← x+1
    Write(x)
    Commit
```

- Possible execution sequences:

```
T1: Read(x)
T1: x ← x+1
T1: Write(x)
T1: Commit
T2: Read(x)
T2: x ← x+1
T2: Write(x)
T2: Commit
```

```
T1: Read(x)
T1: x ← x+1
T2: Read(x)
T1: Write(x)
T2: x ← x+1
T2: Write(x)
T1: Commit
T2: Commit
```

– Serial execution: we get the correct result $x = 52$ (the same for $\{T_2, T_1\}$)

– Concurrent execution: T_2 reads the value of x while it is being changed; the result is $x = 51$ and is incorrect!

- SQL-92 specifies 3 phenomena/situations that occur if proper isolation is not maintained
 - **Dirty read**
 - * T_1 modifies x which is then read by T_2 before T_1 terminates; if T_1 aborts, T_2 has read value which never exists in the DB:
 - **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 P while T_2 inserts new tuples that satisfy P

- Based on the 3 phenomena, SQL-92 specifies different isolation levels:
 - **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 is used to achieve the task

Classification of Transactions

- **Classification** of transactions according to various criteria

- **Duration** of transaction

- * On-line (short-life)
- * Batch (long-life)

- **Organization** of **read** and **write** instructions in transaction

- * General model

$$T_1 : \{R(x), R(y), W(y), R(z), W(x), W(z), W(w), C\}$$

- * Two-step (all reads before writes)

$$T_2 : \{R(x), R(y), R(z), W(x), W(z), W(y), W(w), C\}$$

- * Restricted (a data item has to be read before an update)

$$T_3 : \{R(x), R(y), W(y), R(z), W(x), W(z), \mathbf{R}(w), W(w), C\}$$

- * Action model: each (read,write) pair is executed atomically

$$T_2 : \{[R(x), W(x)], [R(y), W(y)], [R(z), W(z)], [R(w), W(w)], C\}$$

Classification of Transactions ...

- **Classification** of transactions according to various criteria ...

- **Structure** of transaction

- * **Flat** transaction

- Consists of a sequence of primitive operations between a begin and end marker

```
Begin_transaction Reservation
...
end.
```

- * **Nested** transaction

- The operations of a transaction may themselves be transactions.

```
Begin_transaction Reservation
...
Begin_transaction Airline
...
end.
Begin_transaction Hotel
...
end.
end.
```

- * **Workflows** (next slide)

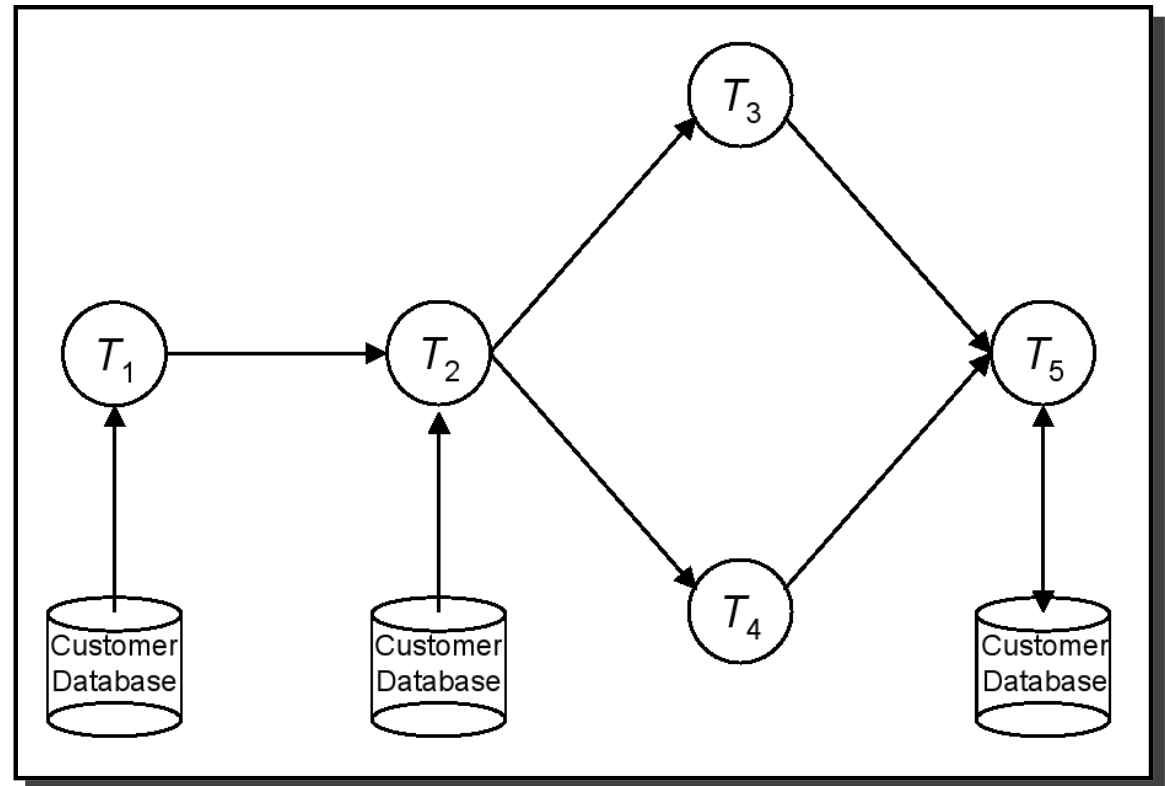
Classification of Transactions ...

- **Workflows:** A collection of tasks organized to accomplish a given business process
 - Workflows generalize transactions and are more expressive to model complex business processes
 - Types of workflows:
 - * Human-oriented workflows
 - Involve humans in performing the tasks.
 - System support for collaboration and coordination; but no system-wide consistency definition
 - * System-oriented workflows
 - Computation-intensive and specialized tasks that can be executed by a computer
 - System support for concurrency control and recovery, automatic task execution, notification, etc.
 - * Transactional workflows
 - In between the previous two; may involve humans, require access to heterogeneous, autonomous and/or distributed systems, and support selective use of ACID properties

Classification of Transactions ...

- **Example:** We extend the reservation example and show a typical workflow

- T_1 : Customer request
- T_2 : Airline reservation
- T_3 : Hotel reservation
- T_4 : Auto reservation
- T_5 : Bill



Transaction Processing Issues

- Transaction structure (usually called transaction model)
 - Flat (simple), nested
- Internal database consistency
 - Semantic data control (integrity enforcement) algorithms
- Reliability protocols
 - Atomicity and Durability
 - Local recovery protocols
 - Global commit protocols
- Concurrency control algorithms
 - How to synchronize concurrent transaction executions (correctness criterion)
 - Intra-transaction consistency, isolation
- Replica control protocols
 - How to control the mutual consistency of replicated data

Conclusion

- A transaction is a collection of actions that transforms the system from one consistent state into another consistent state
- Transaction T can be viewed as a partial order: $T = \{\Sigma, \prec\}$, where Σ is the set of all operations, and \prec denotes the order of operations. T can be also represented as a directed acyclic graph (DAG)
- Transaction manager aims to achieve four properties of transactions: atomicity, consistency, isolation, and durability
- Transactions can be classified according to (i) time, (ii) organization of reads and writes, and (iii) structure
- Transaction processing involves reliability, concurrency, and replication protocols to ensure the four properties of the transactions