Transaction Management

Motivation

- **Database Management System Front End:**
  1. *(i)* Accept query from user
  2. *(ii)* Process query
  3. *(iii)* Output result

  Step *(ii)* - ***core*** of DBMS.

- **Query Processing:**

High Level	Low Level
Query Evaluation	Data storage
Use of index structures	Index organization
Query rewrite rules	File system organization
Query evaluation plans	**Transaction Management**
Cost models	

Transaction (externally) Execution of any user program by a DBMS.

Transaction (internally) A series of **reads** and **writes**.

- **Read database access** (see Fig. 1)
  - Data is stored in *records* on *pages* on *physical storage* devices (e.g., hard disks).
  - Pages are retrieved from physical storage and loaded into *main memory buffer*.
  - Records are retrieved from *pages* in *main memory buffer* and passed to the *transaction program* where they are stored as *variables*.
- **Write database access** (see Fig. 1)
  - Data stored in the variables of the *transaction program* is written into a *page* in *main memory buffer*
  - Pages of *main memory buffer* are written to *physical storage*.

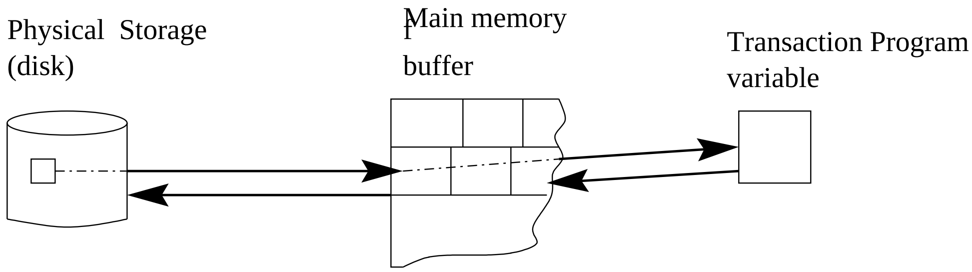

Figure 1: Data Flow in a Database Management System.

ACID Transactions

- **Transaction Managers** of DBMS operate in the following framework:
  - multi-user environment;
  - transaction requests come from many users;
  - transaction history is known;
  - future transactions cannot be predicted;
- **Transaction Managers** must
  - Ensure **integrity** and **consistency** of the data;
  - Be able to process multiple transactions **concurrently**;
  - Recover from **system crashes** (**recovery manager**);

ACID properties of Transactions:

(A)tomicitiy Transaction manager executes either *all* actions of the transaction or *none*. Incomplete transactions should not result in premanent changed in the database.

(C)onsistency Each transaction run by itself must preserve the *consistency* of the database.

(I)solation Each transaction should be protected from effects of other transactions being executed alongside.

(D)urability Once the user informed about successful completion of a transaction, its effects must become *permanent* in the database and should be able to survive system crashes.

Schedules

Transaction A sequence of **actions**.

Action: One of the following:

Read: R(A): Read content of database object A into transaction program memory

Write: W(A): Write new content into database object A

Commit: Successful temnination of transaction

Abort: **Un**successful termination of transaction

- DBMS usually has to manage a set of transactions (set of sequences of actions).

Schedule: A list of actions from a set of transactions $\mathcal{T}$ which preserves the order of actions for *each* transaction $T \in \mathcal{T}$.

  - in other words: *topologically sorted* list of actions from $\mathsf{actions}(\mathcal{T})$.

Complete schedule: Schedule that contains either commit or abort for all transactions $T \in \mathcal{T}$.

- Two transactions are **interleaved** in a (complete) schedule S if the actions of one of them appear *after* the second one starts but *before* it commits or aborts.

Serial schedule: A complete schedule **without** interleaving transactions.

Why Serial Schedules are Bad ?

Example 1 *Consider the following two transactions on the University of Kentucky accouting database:*

1. *T1: Compute* 7.65% *of every employee's monthly salary.*
2. *T2: Find Alex Dekhtyar's monthly contribution to the retirement fund.*

T1 needs to access all 20-40 thousands of records in the `salary` relation.
*T2 needs to access **one** record from the `retirement` relation.*

In a serial schedule where T1 is issued first and T2 is issued seconds later, no action from T2 will be executed until T1 is committed (or aborted). This may take a very long time.

Serializability

- **Consistent database state**
  - defined by database manager/designer
  - integrity constraints
  - key/participation constraints
- **Transactions must preserve database consistency**
  - If a transaction started in a consistent state, it must end its execution in one.
- **Note:** serial schedules result in consistent database states.
- **Serializable schedule** over a set of committed transactions - a schedule whose effect on any database in a consistent state is *guaranteed* to be identical to **some** serial schedule.

- **Note:** Different serial schedules of the same set of transactions may result in different final states of the database.
- Equivalence to **any** serial schedule is sufficient for **serializability**!!!

- **Serializable schedule** over a set of transactions $\mathcal{T}$ - a schedule whose effect on any database in a consistent state is guaranteed to be equivalent to **any serial schedule** of the set of **all committed transactions** from $\mathcal{T}$.
  - Idea: Influence of aborted transactions **should not be felt**.

Conflicts

Non-serial schedules may be subject to the following three types of conflicts (problems leading to non-serializability):

1. **Write-Read (WR)**: Reading uncommitted data
2. **Read-Write (RW)**: Unrepeatable read
3. **Write-Write (WW)**: Writing uncommitted data

Write-Read (WR) Conflict

Description: **WR** conflict occurs when one transaction changes a particular database object, then another transaction reads it after which the first transaction aborts.

Example: Consider the following schedule of actions for transactions T1 and T2:

T1:W(A); T2:R(A);T2:W(B); T1:Abort; T2:Commit.

T1 assigns new value(s) to object A but later aborts, causing the change to be undone. However, in the meantime, T2 reads the value(s) of object A as assigned by T1 and continues execution writing the value(s) of object B, which quite possibly can depend on the value(s) obtained from A.

Read-Write (RW) Conflict

Description: **RW** conflict occurs when one transaction changes reads value(s) from a particular database object, this object is then overwritten by second transaction, after which first transaction re-reads the value of the object.

Example: Consider the following schedule of actions for transactions T1 and T2:

T1:R(A); T2:W(A);T1:R(A);T1:W(B) T1:Commit; T2:Commit.

In a serial schedule first and second read of A by T1 yield the same result. In the schedule above, the results of these two reads are different (assuming T2 changed A), therefore, the rest of the transaction T1 is possibly affected.

Write-Write (WW) Conflict

Description: **WW** conflict is similar occurs when writes interleaved from different transactions result in an inconsistent state of the database.

Example: Consider the following schedule of actions for transactions T1 and T2:

T1:W(A); T2:W(B);T1:W(B);T2:W(A) T1:Commit; T2:Commit.

Each transaction by itself results in a consistent database state. However, in the schedule above the result stored in A comes from transaction T2 and the result stored in B comes from transaction T1. These two values may be inconsistent.