This lecture

This lecture introduces the concepts underlying database transactions.

Concurrency

One of the most important properties of a modern DBMS is its ability to manage multiple client sessions simultaneously and transparently to the users of the database.

It is important for a DBA to understand how such concurrency control is managed by the database as it can have a significant impact on the overall performance of the database.

Transactions

We define a transaction as any one execution of a user program.

In this context, a user program consists of a number of statements that read and write database objects (i.e. tables, values etc), before finally committing at which point any changes to the state of the DB are made permanent (i.e. written to disk).

For certain applications, it is critical that all the statements in a transaction run to completion without interference from other users.
Concurrency
Bank Transfers
The “canonical example” of an application where correct treatment of transactions is critical is transferring money in a bank.
For example, suppose that a user at an ATM transfers money between two accounts.

```sql
UPDATE accounts SET balance = balance - 500  
    WHERE id = 1;
UPDATE accounts SET balance = balance + 500  
    WHERE id = 2;
```

It is crucial that either both statements occur or neither do — for example, if the computer crashes after the first one has occurred then the system must be able to recognize and recover from that.

Transaction Properties
A transaction-safe database engine must ensure that the following four properties — known by the acronym ACID — are maintained.

- Atomicity
- Consistency
- Isolation
- Durability

ACID
Atomicity
The word atomic is used in a number of contexts to denote indivisible.
In a DB context, transactions are atomic if the system ensures that they cannot be “half-done” — in other words, the user is guaranteed that either the entire transaction completes or it fails and has no effect on the database.
The bank transfer example above is one application where users would rely on the atomicity of transactions.

ACID
Consistency
Transactions must preserve the consistency of the database.
More precisely, if the database is in a consistent state, and a transaction is executed to completion on its own (i.e. with no concurrently executing transactions) then the state of the database after the transaction should also be consistent.
This is basically a fancy way of saying that the user’s programs should be correct. Transaction consistency is therefore the responsibility of the user not the DBMS.
Isolation

*Isolation* means that the user of the DB should be able to execute a transaction without regard for concurrently executing transactions.

In other words, the user’s actions should be *isolated* from the actions of other users — at least for the duration of the transaction.

---

Durability

*Durability* means that once the user is informed of the successful completion of a transaction, then its effects on the database are persistent.

Thus the user should be shielded from any possible problems (eg system crashes) that might occur after being notified that the transaction has successfully completed.

---

Atomicity and Durability

Ensuring *atomicity* requires the DBMS to be able to *undo* the effect of earlier statements if the entire transaction is aborted, either by the DBMS itself (if a later statement fails) or for some external reason (system crash, power cut etc).

The basic mechanism used for this is that the DBMS maintains a *log* of all changes to the database. Every action that causes a change to the state of the database is *first* recorded in the logfile, which is then saved to disk. Finally the new state of the database is written to disk.

---

Write Ahead Log

The property that changes are *logged* before they are actually made on disk is called *write ahead log*.

If a transaction is aborted, then the DBMS can consult the log in order to determine which actions need to be undone in order to restore the database to its initial state.

In the case of a system crash, the *recovery manager* uses the log to determine whether there are any completed transactions that still need to be written to disk.

Complete details of the logging process and the recovery manager are complicated and require a detailed understanding of the physical aspects of computers.
Interleaving

There would be no problem with isolation if the DBMS were able to simply run each transaction to completion at a time before starting the next one.

However in practice, it is vital to interleave the actions of transactions in order for the system to be usable in practice.

Motivation for Interleaving

Interleaving transactions (properly) allows multiple users of the database to access it at the same time.

While one transaction is performing an I/O task, another one can perform a CPU-intensive task thus maximising the throughput of the system.

Strict serial execution of transactions would be impractical because large numbers of short transactions would become “queued up” behind a long running transaction waiting for it to finish.

Thus managing a collection of interleaved transactions is a fundamental task for a DBMS.

Notation

We use the notation $R(O)$ and $W(O)$ to indicate the actions of reading a database object $O$ and writing a database object $O$.

Then a transaction can be considered to be a sequence of reading and writing actions ending when the transaction commits.

Interleaving Anomalies

There are a variety of anomalies that can arise from an unfortunate choice of schedule for interleaved transactions.

Each of these anomalies could leave the database in an inconsistent state that could not arise if the two transactions were not interleaved.

- Dirty Reads
- Nonrepeatable Reads
- Phantoms
Dirty Reads

A *dirty read* occurs when one transaction reads a database value that has been altered by a transaction that has not yet committed.

Two major problems can arise from dirty reads:

- The database may be in a temporarily inconsistent state due to the partially completed transaction.
- The partially completed transaction may subsequently be aborted restoring the value to its original state.

Dirty Read Example

Suppose that $T_1$ transfers $100$ from account $A$ to account $B$, while $T_2$ adds 5% interest to each account, and the following schedule is used:

\[
\begin{array}{c|c|c}
\text{Time} & T_1 & T_2 \\
\hline
R(A) & W(A) & R(A) \\
W(A) & & W(A) \\
R(B) & W(B) & R(B) \\
W(B) & & W(B) \\
\end{array}
\]

Dirty Read Example cont.

If $A$ and $B$ have a $1000$ balance initially then this schedule would proceed as follows:

- $T_1$ deducts $100$ from $A$ so balance is $900$.
- $T_2$ adds 5% interest to $A$ so balance is $945$.
- $T_2$ adds 5% interest to $B$ so balance is $1050$.
- $T_1$ adds $100$ to $B$ so balance is $1150$.

Neither of the two possible serial schedules (i.e. $T_1$ first, then $T_2$ or vice versa) would give these values, and in fact $5$ interest has been lost.

The fundamental problem is that $T_1$ put the DB into an inconsistent state, and $T_2$ used the inconsistent values before $T_1$ could restore the DB.

Unrepeatable Reads

An unrepeatable read is essentially the dirty-read problem in reverse order in that a value gets changed by another transaction *after* it has been read, rather than before.

In this situation, transaction $T_1$ reads a value which is then changed by $T_2$. If $T_1$ subsequently re-reads the value then it gets a different value, even though it hasn’t changed it.

This violates the isolation property because transaction $T_1$ should be able to complete as though it is the only transaction currently executing.
Phantoms

A *phantom* is a variant of the unrepeatable read problem that occurs when one transaction performs a *SELECT* statement with some selection criteria, and then subsequently another transaction inserts a new row.

If the first transaction now uses the same criteria again for a subsequent *SEARCH* or *UPDATE* statement, then a new row will suddenly appear, known as a *phantom row.*