Transactions, Concurrency, and Recovery

At this point, we leave the purely theoretical realm of relational databases and enter into issues that arise in real-world usage, such as:

- Real-world databases frequently perform operations that span multiple tables, queries, and updates — also known as transactions
- Real-world databases are accessed by multiple users at the same time — this is concurrency
- Stuff happens (in a bad way) with real-world databases — so we need recovery

Your Job vs. the Database’s Job

- Generally, transaction management falls upon the database programmer — after all, the database can’t know how your individual database activities fit into the bigger picture or context
- However, concurrency control is typically left to the database, under the simple guideline that we want individual users to feel like they have the database “to themselves,” within reason
- The recovery system goes both ways: a DBMS should allow it, but it sometimes relies on human intervention
Transactions

- The idea behind a transaction (also known as a unit of work) is straightforward — it acknowledges that a single logical operation on a database may encompass multiple queries, inserts, deletes, or updates.
- Transactions are driven by the application — i.e., the domain determines what operations form logical units of work — and so, the database can’t figure them out on its own.
- Thus, transactions must be explicitly delimited by the developer via begin transaction/end transaction directives.

The ACID Test

- Transactions are supposed to have four properties:
  - Atomicity: transaction success is all-or-nothing.
  - Consistency: transaction shouldn’t break any data rules.
  - Isolation: transactions must not stomp over each other.
  - Durability: transaction effects should persist, even after a system failure.
- In explaining these properties, note how, without loss of generality, we can view all database activity in terms of read(X) and write(X), where X is some data item.
The “Classic” Transaction

• The canonical example for transactions is a bank transfer of some amount between accounts A and B

• In terms of read and write, we can describe the transaction $T_i$ as follows:

```plaintext
read(A);
A := A – amount;
write(A);
read(B);
B := B + amount;
write(B);
```

Note how the “ACID test” ensures that this transaction does do what we think it should do in terms of its real-world meaning:

• Atomicity means that either all of $T$ occurs or not at all; otherwise, you may have a transfer to or from nowhere

• Consistency dictates that $T$ shouldn’t create or destroy money — it’s a bank transfer after all

• Isolation means that a concurrent activity shouldn’t affect $T$’s calculations — again, we otherwise run the risk of creating or destroying money wrongly

• Durability means that $T$ should “stick” — if the servers crash after this transaction succeeds, the user shouldn’t have to do it over
Transaction Life Cycle

Here’s a laundry list of terms to know:

• A successful transaction is committed
• A failed transaction is aborted
• When aborted, a transaction’s intermediate changes need to be rolled back
• If a committed transaction needs to be undone, we need to perform a new compensating transaction
• Regardless of success or failure, a transaction is terminated when its operation concludes

Transaction State

While a transaction is “running,” it may be in one of the following states:

• Active is the initial state, and indicates that the transaction is in progress
• Partially committed indicates that the transaction has executed its final statement
• Failed indicates that something went wrong
• Aborted means that rollback has taken place
• Committed means that the transaction is complete
Transaction State Management

- Transaction starts out *active*
- If any error condition is detected, then it is *failed*; upon rollback, it is *aborted*
- At this point, a decision may be made to either *restart* the transaction or to *kill* it
- If the transaction makes it to the final statement, then it is *partially committed*; the database must now make sure that these changes are truly permanent (i.e., the *durability* property), after which the transaction may now be viewed as *committed*.

External Factors

- Some transactions involve *observable external writes* — for example, intermediate output to a printer, Web page, monitoring display
- In general, most systems refrain from doing this at all until the transaction successfully completes
- On the other hand, the issue may be the *failure* to perform an observable external write, such as a breakdown in an ATM’s cash dispensing function right after performing a withdrawal electronically
- A usual, it’s important to know the domain well
Concurrent Transactions

• Ideally, we perform transactions one at a time, or *serially*, as they arrive at the server, since they are after all self-contained units of work

• However, this is not the most efficient way to do it; for improved throughput, better resource utilization, and reduced waiting time, we want to allow *concurrent execution* of transactions

• We define *correct* concurrent execution as equivalence to *serial* execution — the concurrent transactions must execute as if one or the other strictly came first

Serializability

• Ensuring equivalence to serial execution is the database’s job, since it knows what transactions are “coming in” and in what order; the transactions themselves don’t (and shouldn’t) care about who else is running at the same time

• In performing concurrent transactions, the database creates a *schedule* for when the operations within each transaction will be performed

• This schedule is said to be *serializable* if its effects are equivalent to serial execution
Conflict Serializability

• One approach to scheduling operations from different transactions — say $l_i$ and $l_j$ from different transactions $T_i$ and $T_j$ — is to determine how they might conflict with each other

• First off, if the operations affect different data items, then there is no conflict — they can go in any order

• So we need to worry only about cases where $l_i$ and $l_j$ will either read or write the same data item $Q$

• With a little reasoning, one can see that $l_i$ and $l_j$ will conflict if either operation is a write($Q$)

• Based on this view, we may only swap the order of operations between transactions if they do not conflict — that is, they either operate on different data items or are both read operations

• Concurrent schedules $S$ and $S'$ are said to be conflict equivalent if they can be transformed into each other through a series of swaps of non-conflicting operations

• A schedule is then conflict serializable if it is conflict equivalent to a serial schedule

• Note that conflict serializability is a conservative definition of serializability — there are schedules for certain transactions that are not conflict serializable but do yield equivalent results to a serial schedule
View Serializability

- Instead of conflict equivalence, an alternative criterion is view equivalence, where the equivalence of two schedules is determined by what they “see” for every data item Q in the transaction:
  - Preserve the transaction that reads Q’s initial value
  - Preserve the order where a read(Q) in one transaction depends on a write(Q) in the other
  - Preserve the transaction that writes Q’s final value
- A schedule is view serializable if it is view equivalent to a serial schedule

- Every conflict serializable schedule is also view serializable — makes sense, since conflict serializability is “stricter” than view serializability
- But not vice versa — there are view serializable schedules that are not conflict serializable
- These schedules are easy to characterize — they are the ones that contain blind writes, or writes to a data item that are not preceded by a read
- If you think about it, this characterization makes sense; blind writes prevent conflict equivalence since they can’t be swapped with other operations, but they do accommodate view equivalence because they are not dependent on a previous read operation
Transaction Recoverability

- When scheduling multiple transactions, we may also have dependencies across transactions — that is, $T_j$ may depend on the result of $T_i$.
- But what if $T_i$ fails and must abort? $T_j$ must be aborted also, since it isn’t allowed to see the “side effects” left by $T_i$.
- A recoverable schedule is one that, if $T_j$ reads data that is written by $T_i$, will commit $T_i$ before it commits $T_j$.
- That way, if something goes wrong in $T_i$, we can abort $T_j$ as well.

- In practice, a recoverable schedule is not enough — for example, if you have a large number of sequentially dependent transactions, you probably won’t want to abort that many in the event that the first transaction fails and aborts.
- This situation is called cascading rollback.
- To avoid cascading rollback, we prefer to enforce a cascadeless schedule instead of just a recoverable schedule — a cascadeless schedule is one where, for transactions $T_i$ and $T_j$ such that $T_j$ reads data that is written by $T_i$, the commit operation for $T_i$ must appear before the dependent read operation in $T_j$.
- Every cascadeless schedule is also recoverable.
Isolation and Transaction Schedules

• The main concern with proper scheduling of concurrent transactions is that we preserve the isolation property of transactions — they must operate as if they had the database “to themselves”

• We’ve specified the criteria for schedules that do this — they must be conflict or view serializable, and cascadeless too

• A concurrency-control scheme is some algorithm that generates such a schedule, but which also tries to reap the benefits of concurrent execution

• Designing an acceptable concurrency-control scheme requires that we can prove that the schedules that the scheme generates are serializable

• A simple and efficient conflict serializability test involves creating a directed graph of the schedule being tested, called a precedence graph

• An edge joins two transactions in the precedence graph if they involve conflicting operations (i.e., at least one write for the same item), allowing us to conclude:
  ◊ Equivalent schedules have the same precedence edges
  ◊ Conflict serializable schedules do not have cycles

• A topological sorting converts the partial order of the precedence graph into a linear order, and this is equivalent to a serializability order for the transactions