Transactions

- Serializability
- Isolation Levels
- Atomicity
Database systems are normally being accessed by many users or processes at the same time.

- Both queries and modifications.

Unlike operating systems, which support interaction of processes, a DMBS needs to keep processes from troublesome interactions.
Example: Bad Interaction

- You and your spouse each take $100 from your shared bank account at different ATM’s at about the same time.
  - The DBMS better make sure one account deduction doesn’t get lost.

- Compare: An OS allows two people to edit a document at the same time. If both write, one’s changes get lost.
A DBMS is expected to support “ACID transactions,” processes that are:

- **Atomic**: Either the whole process is done or none is.
- **Consistent**: Database constraints are preserved.
- **Isolated**: It appears to the user as if only one process executes at a time.
- **Durable**: Effects of a process do not get lost if the system crashes.
Transactions in SQL

- SQL supports transactions, often behind the scenes.
  - Each statement issued at the generic query interface (the facility wherein one types queries and other SQL statements) is a transaction by itself.
  - However, SQL allows the programmer to group several statements into a single transaction. The SQL command START TRANSACTION is used to mark the beginning of a transaction. There are two ways to end a transaction (see next page).
The SQL statement COMMIT causes a transaction to complete.
- Its database modifications are now permanent in the database.
The SQL statement ROLLBACK also causes the transaction to end, but by **aborting**.

- No effects on the database.

Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it.
An Example: Interacting Processes

- Assume the usual $\text{Sells}(\text{store}, \text{candy}, \text{price})$ relation, and suppose that Joe’s Market sells only Twizzlers for $2.50 and Kitkats for $3.00.

- Sally is querying $\text{Sells}$ for the highest and lowest price Joe charges.

- Joe decides to stop selling Twizzlers and Kitkats, but to sell only Smarties at $3.50.
Sally executes the following two SQL statements, which we call \((\text{min})\) and \((\text{max})\), to help remember what they do.

\[(\text{max})\] 
SELECT MAX(price) FROM Sells
WHERE store = 'Joe's Market';

\[(\text{min})\] 
SELECT MIN(price) FROM Sells
WHERE store = 'Joe's Market';
Joe’s Program

At about the same time, Joe executes the following steps, which have the mnemonic names (del) and (ins).

(del) DELETE FROM Sells
WHERE store = 'Joe’s Market';

(ins) INSERT INTO Sells
VALUES('Joe’s Market', 'Smarties', 3.50);
Interleaving of Statements

- Although (max) must come before (min), and (del) must come before (ins), there are no other constraints on the order of these statements, unless we group Sally’s and/or Joe’s statements into transactions.
Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min).

Joe’s Prices: 2.50, 3.00  2.50, 3.00  3.50
Statement:  (max) (del) (ins) (min)
Result: 3.00 3.50

Sally sees MAX < MIN!
Fixing the Problem by Using Transactions

- If we group Sally’s statements (max) (min) into one transaction, then she cannot see this inconsistency.

- She sees Joe’s prices at some fixed time.
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.
Another Problem: Rollback

Suppose Joe executes \((\text{del})(\text{ins})\), now as a transaction, but after executing these statements, thinks better of it and issues a ROLLBACK statement.

What if Sally executes her statements after \((\text{ins})\) but before the rollback, she sees a value, 3.50, that never existed in the database.
If Joe executes \((\text{del})(\text{ins})\) as a transaction, Sally (ideally) should not see its effect until the transaction executes COMMIT.

If the transaction executes ROLLBACK instead, then its effects can never be seen.
Isolation Levels

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time.
- How a DBMS implements these isolation levels is highly complex, and a typical DBMS provides its own options.
Choosing the Isolation Level

Before a transaction starts, we can say:

```
SET TRANSACTION ISOLATION LEVEL X
```

where $X =$

- SERIALIZABLE or
- REPEATABLE READ or
- READ COMMITTED or
- READ UNCOMMITTED
Serializable Transactions

- If Sally = \((\text{max})(\text{min})\) and Joe = \((\text{del})(\text{ins})\) are each transactions, and Sally runs with isolation level SERIALIZABLE, then she will see the database either before or after Joe runs, but not in the middle.

- It’s up to the DBMS vendor to figure out how to do that, e.g.:
  - True isolation in time.
  - Keep Joe’s old prices around to answer Sally’s queries.
Isolation Level Is Personal Choice

- Your choice, e.g., run serializable, affects only how you see the database, not how others see it.

- **Example**: If Joe runs serializable, but Sally doesn’t, then Sally might see no prices for Joe’s Market.
  - i.e., it looks to Sally as if she ran in the middle of Joe’s transaction.
Read-Committed Transactions

- If Sally runs with isolation level READ COMMITTED, then she can see only committed data, but not necessarily the same data each time.

Example: Under READ COMMITTED, the interleaving \((\text{max})(\text{del})(\text{ins})(\text{min})\) is allowed, as long as Joe commits.
  - Sally sees \(\text{MAX} < \text{MIN}\).
Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time.
  - But the second and subsequent reads may see *more* tuples as well.
Suppose Sally runs under REPEATABLE READ, and the order of execution is \((\text{max})(\text{del})(\text{ins})(\text{min})\).

- \((\text{max})\) sees prices 2.50 and 3.00.
- \((\text{min})\) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by \((\text{max})\).
Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never).

- Example: If Sally runs under READ UNCOMMITTED, she could see a price 3.50 even if Joe later aborts.
Chapter 8: Views and Indexes
Declaring Views

The simplest form of view definition is

\[
\text{CREATE VIEW } \text{<view-name>} \text{ AS } \text{<view-definition>};
\]

The view definition is a SQL query.

Example:  \[
\text{CREATE VIEW MovieProducer AS}
\text{SELECT title, name}
\text{FROM Movies, MovieExec}
\text{WHERE producerC# = cert#};
\]
Example:

CREATE VIEW MovieProducer(movieTitle, prodName) AS
    SELECT title, name
    FROM Movies, MovieExec
    WHERE producerC#, cert#
View Removal

Example:

DROP VIEW MovieProducer
Indexes in SQL

Example:

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year=1990;
```
Declaring Indexes

Example:

CREATE INDEX YearIndex ON Movies(year);

CREATE INDEX KeyIndex ON Movies(title, year);
Chapter 9

SQL In a Server Environment
Combining SQL and Conventional Programming Languages
Shortcomings of SQL

- Relational data model doesn't match well with data model of conventional programming languages (e.g., data structure mismatch)
- No pointers, loops or branches in SQL
- No convenient input and output (e.g., formatting)
We have seen only how SQL is used at the generic query interface --- an environment where we sit at a terminal and ask queries of a database.

Reality is almost always different.

Programs in a conventional language like C are written to access a database by “calls” to SQL statements.
Three ways to combine:

- Persistent Stored Modules (code stored in the DB schema and executed on command from a user)
- Embed SQL statements in programs written in some ordinary language
- Call-level interfaces
  - SQL/CLI (SQL standard, for use with C)
  - JDBC (for use with Java)
Persistent Stored Modules

- A recent SQL standard
- Mechanism for user to store in the DB schema functions and procedures that can be used in SQL statements
- The functions and procedures are written in a simple general-purpose language
- Includes if, loops, variable declarations, as well as SQL queries and updates
- See Chapter 9 for more info.
Embedded SQL and CLI's

host language + embedded SQL

preprocessor

host language + function calls (CLI)

host-language compiler

SQL library

object-code program
Host Languages

- Any conventional language can be a host language, that is, a language in which SQL calls are embedded.
- The use of a host/SQL combination allows us to do anything computable, yet still get the very-high-level SQL interface to the database.
Connecting SQL to the Host Language

- *Embedded SQL* is a standard for combining SQL with seven languages.
- CLI (*Call-Level Interface*) is a different approach to connecting C to an SQL database.
- JDBC (*Java Database Connectivity*) is a way to connect Java with an SQL database (analogous to CLI).
Key idea: Use a preprocessor to turn SQL statements into procedure calls that fit with the host-language code surrounding.

All embedded SQL statements begin with EXEC SQL, so the preprocessor can find them easily.
Issues for Embedded SQL

- how to transfer data between host language and SQL -- use shared variables
- how to handle multiple tuples returned by a query -- notion of a "cursor"
- how to execute SQL statements that are not known at compile time ("dynamic SQL")

See Chapter 9 for more details.
Instead of using a preprocessor, we can use a library of functions and call them as part of an ordinary C program.

- The library for C is called SQL/CLI = “Call-Level Interface.”
- Embedded SQL’s preprocessor will translate the EXEC SQL … statements into CLI or similar calls, anyway.
Java Database Connectivity (JDBC) is a library similar to SQL/CLI, but with Java as the host language.

JDBC/CLI differences are often related to the object-oriented style of Java, but there are other differences.
Overview of JDBC

- A "driver" for the database system to be used must be loaded. Result is creation of a DriverManager object.
- A *connection object* is obtained from the DriverManager in a somewhat implementation-dependent way.
- We’ll start by assuming we have `myCon`, a connection object.
JDBC provides two classes:

- **Statement** = an object that can accept a string that is an SQL statement and can execute such a string.
- **PreparedStatement** = an object that has an associated SQL statement ready to execute.
Creating Statements

- The Connection class has methods to create Statements and PreparedStatements.

```java
Statement stat1 = myCon.createStatement();
PreparedStatement stat2 = myCon.createStatement(
    "SELECT candy, price FROM Sells 
    WHERE store = '7-11'");
```

Java trick: `+` concatenates strings.

`createStatement` with no argument returns a Statement; with one argument it returns a PreparedStatement.
JDBC distinguishes queries from modifications, which it calls “updates.”

Statement and PreparedStatement each have methods `executeQuery` and `executeUpdate`.

- For Statements, these methods have one argument: the query or modification to be executed.
- For PreparedStatements: no argument.
Example: Update

- stat1 is a Statement.
- We can use it to insert a tuple as:

```java
stat1.executeUpdate(
    "INSERT INTO Sells " +
    "VALUES('Safeway', 'Kitkat', 3.00)"
);
```
Example: Query

- `stat2` is a `PreparedStatement` holding the query "SELECT candy, price FROM Sells WHERE store = '7-11'."
- `executeQuery` returns an object of class `ResultSet` --- we’ll examine it later.
- The query: `ResultSet Menu = stat2.executeQuery();`
An object of type ResultSet is something like a cursor (from PSM).

Method `next()` advances the “cursor” to the next tuple.
  - The first time `next()` is applied, it gets the first tuple.
  - If there are no more tuples, `next()` returns the value FALSE.
When a ResultSet is referring to a tuple, we can get the components of that tuple by applying certain methods to the ResultSet.

Method $\text{get}X(i)$, where $X$ is some type, and $i$ is the component number, returns the value of that component.

- The value must have type $X$. 

Accessing Components of Tuples
Example: Accessing Components

- Menu is the ResultSet for the query "SELECT candy, price FROM Sells WHERE store = '7-11' ".
- Access the candy and price from each tuple by:

```java
while ( Menu.next() ) {
    theCandy = Menu.getString(1);
    thePrice = Menu.getFloat(2);
    /* do something with theCandy and thePrice */
}
```