

Chapter 2

Basics via a Case Study

We all know how WebReg looks like and operates from this side. Now, let's have a look into the other side. The goal is clear: we want to let the students be able to register courses from their home PCs. More specifically, we could have the following *statement of objectives*:

1. Authenticate faculty and students as users of the system.
2. Add and drop courses.
3. Obtain status reports on a particular student.
4. Maintain information about students and courses.
5. Enter final grades for those who complete courses.

This list is often provided as a starting point, but not specific enough.

What next?

We will use this case as a running example to make quite a few points for the concepts related to database development, including both the design and manipulation of such a database.

Once we have a basic idea, we will then meet with the users such as the registrar, faculty and students to expand this brief description into a formal *requirements document*, which is given near the end of this book (§14.2).

In this unit, we will merely check through some of the relevant concepts of databases and transaction processing that are needed for this system.

Relational databases

As already mentioned, a database is the heart of most of the transaction processing system, in the sense that at every time point, the database must contain an accurate description of the real world of the enterprise this database is modeling.

In this case, the database is the only source of information about who is taking what courses.

Most of the current databases follow the *relational model (RDB)*, where data is stored in a *table*. For example, one of the registration data piece might look like the following:

Id	Name	Address	Status
1111	John Doe	123 Main St.	Freshman
2222	Mary Smith	1 Lake St.	Freshman
1234	Joe Blow	6 Yard Ct.	Junior

Tables and their company

A table is a *set* of rows, thus, by the definition, it contains no duplicates. For example, the previous table contains 3 rows (tuple). It also contains 4 columns. In general, each column (attribute) states a particular fact about each row and is associated with a domain (type). For example, the first column states, for each row, his/her id number, is of an integer type.

This model is based on the concept of a *mathematical relation*, covered in either MA2200 or MA3200. This concept captures the notion that elements of different sets can be related to each other. For example, John Doe, an element of the set of all humans, is related to 123 Main St., a member of the set of all addresses, and to 1111, a member of all the ids.

Another perspective

A row is often call a *tuple*, and then a relation is just a collection of such tuples. A relation can also be thought of in terms of a *predicate*, which is either true or false. For example, “You do like math.”, “It is raining outside.” “ $3+5=9$ ”, etc.

Then, a relation R can be thought of as predicate R , such that $R(x, y, z)$ is true iff the tuple (x, y, z) is in R .

For example, $(1111, JohnDoe, 123MainSt., Freshman)$ is in the relation since it is true, i.e., “A freshman John Doe, with id 1111, does live in 123 Main St..” .

Table creation

The creation of a table can be specified in *SQL* as follows:

```
CREATE TABLE Student (  
    Id      Integer,  
    Name    Char(20) Not Null,  
    Address Char(50),  
    Status  Char(10) Default 'freshman'  
  
    PRIMARY KEY (Id));
```

One of its implementation in MySQL is the following:

```
create table Student (  
    Id INT Not Null Primary key,  
    Name Char(20) Not Null,  
    Address Char(50),  
    Status Char(20) default 'freshman');
```

Homework: Exercises 2.1, 2.2 and 2.4.

Database operations...

In a real situation, a table could be quite large. There are 4,000 plus students, thus the rows; and we also want to keep more information in such a table, thus more columns. There could be also more tables, e.g., we might want to have a *Transcript* table that contains students' grades for each and every course they have taken.

A database is usually controlled by a DBMS, e.g., Oracle, Mysql, etc.. When a user wants to apply a query or an update to the database, s/he submits a request to DBMS. Such an operation can be either get some information from the rows of one or more tables, modify some rows, add or delete some rows into or from a table. For example, when a student just transferred in, we want to add her information into the *Student* table.

...are based on mathematics.

The above operations are implemented via operations defined for mathematical relations. Typically, a *unary* operation might take a table, T , as its input, and produce another one, containing a subset of the rows taken from T . For example, when I want to have a roster for this course, the DBMS will have a look at the *Transcript* table and collect only those who are taking CS3600 in Fall 2008.

On the other hand, a binary operation will take two tables as inputs, and sends something back. For example, when you try to register for CS3600, the DBMS will generate an intersection of two tables for MA2200 and CS2370, and only those that show up in this intersection will be allowed to register for CS3600.

Why this way?

We can show that any query can be expressed exactly as a combination of some basic relational operations. We can thus prove beyond any doubt the correctness of the operation.

In practice, once a query is submitted to the database, it will be converted into a mathematical expression first, and then a *query optimizer* can use the mathematical properties such as commutativity and associativity to find out an equivalent but more efficient expression before carrying it out.

We will further discuss it in *Query processing basics* later.

Basics of SQL

The basic idea is that an application describes *what it wants* and the DBMS decides *how to get it*. The SQL standard provides a way for us to describe what we want.

The basic structure is the same for all the query and/or update operations:

```
Select attributes
From tables
Where conditions
```

For example

```
mysql> Select Name From Student
      -> Where Id='111111111';
+-----+
| Name   |
+-----+
| Jane Doe |
+-----+
1 row in set (0.00 sec)
```

What is going on?

This query wants the DBMS to send back the *Name* part of all the tuples from the table *Student* that satisfies the condition that its *Id* part equals to '111111111'.

In general, given such a query statement, the *Select* part lists the columns it wants, the *From* part lists the data sources, and the *Where* part specifies the conditions.

Procedurally, the DBMS will scan every row, and for each row, it will check if it satisfies the condition, and sends back the required parts if it does.

Considering the data volume, certain optimization is certainly involved.

Homework: 2.3.

More examples

The following wants both the Id and the Name of all the senior students.

```
mysql> Select Id, Name From Student
      -> Where Status='senior';
+-----+-----+
| Id          | Name          |
+-----+-----+
| 23456789   | Homer Simpson |
| 987654321  | Bart Simpson  |
+-----+-----+
2 rows in set (0.00 sec)
```

When you want everything back, you simply use a '*' as follows:

```
mysql> Select * From Student
      -> Where Status='senior';
```

```

+-----+-----+-----+-----+-----+-----+
| Id      | Name           | Address  | Status  | Age  | GPA  |
+-----+-----+-----+-----+-----+-----+
| 23456789 | Homer Simpson | Fox 5 Tv | senior  | 21  | 3.3  |
| 987654321 | Bart Simpson  | Fox 5 Tv | senior  | 22  | 3.6  |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

In some cases, a user might want to have aggregated information, such as “How many seniors do we have?”.

```

mysql> Select count(*) From Student
      -> Where Status='senior';
+-----+
| count(*) |
+-----+
|         2 |
+-----+
1 row in set (0.02 sec)

```

We can certainly put on more complex conditions, i.e., conjunctive or disjunctive conditions.

For example,

```
mysql> Select Id, Name From Student
      -> Where Status='senior' AND Id>'555555555';
+-----+-----+
| Id      | Name          |
+-----+-----+
| 987654321 | Bart Simpson |
+-----+-----+
1 row in set (0.00 sec)
```

We can also get those who are either freshman or sophomore:

```
mysql> Select Id, Name From Student  
      -> Where Status in ('freshman', 'sophomore');
```

```
+-----+-----+  
| Id          | Name          |  
+-----+-----+  
| 111111111  | Jane Doe     |  
| 111223344  | Mary Smith   |  
| 666666666  | Joseph Public|  
+-----+-----+
```

```
3 rows in set (0.01 sec)
```

We can also get information from multiple tables.

```
mysql> Select Name, CrsCode, Grade
      -> From Student, Transcript
      -> Where Status='senior' AND StudId=Id;
```

Name	CrsCode	Grade
Homer Simpson	CS305	A
Homer Simpson	EE101	B
Bart Simpson	CS305	C
Bart Simpson	MGT123	B

4 rows in set (0.08 sec)

Question: How did it work?

Optimization

A very important feature of SQL is that the programmer does not specify how to find those information.

On the other hand, a database comes with various *indices*, which will help the DBMS to find the information quickly. For example, if we put an index on the Id attribute, then an index file for this column will be a collection of (Id, *pointer*) pair where for each Id value, a *pointer* points where the row with this Id value is kept. The existence of such an index will speed up the query process.

Question: How?

The optimizer also makes use of the properties of the relational operations to further improve the efficiency of query processing.

Update operations

Databases keep on changing. As we saw already, we can also update the tables.

Update Student

```
Set Status ='sophomore'
```

```
Where Id='111111111'
```

We can also add something in, or ...

```
Insert Into Student (Id, Name, Address, Status)
```

```
Values
```

```
('9999', 'Winston, Chad', '10 Downing St.', 'senior')
```

... take something out.

```
Delete From Student
```

```
Where Id='111111111'
```

Homework: 2.5 and 2.6.

Consistency

As we mentioned, in many cases, a database is used to model the state of some real-world enterprise. In such cases, a transaction has to guarantee the correspondence between the data base and the real-world state by updating the database.

To make sure this is always done, a set of requirements are enforced to the operations of such transaction processing. First and foremost, a transaction preserves all *database integrity constraints* when getting access and updating a database.

An example of an integrity rule is that “The number of students registered for a course can’t exceed the capacity of the room assigned for this course.” .

Question: What is the capacity of this room?

Integrity rules

IC0: The database contains the unique Id of each student.

IC1: The database contains a list of prerequisites for each course and, for each student, a list of completed courses. A student can't take a course w/o having taken all the prerequisites of that course.

Question: How does *it* enforce *IC1*?

IC2: The database contains the maximum number of students allowed to take each course and the number of students currently registered for each course, which can't be larger than the previous number.

IC3: It might be possible to determine the number of students registered for a particular course in two ways: the number stored as a count in the information describing the course; or by counting from a table describing the students by counting the number of student records that indicate s/he has registered for that course.

These two ways must yield the same result all the times.

Homework: 2.9.

Besides maintaining these rules, each transaction must update the database in such a way that the new state reflects the state in the real world. For example, if it is John who just registered for CS3600, but the transaction adds Mary as the new student, the integrity rule might be upheld, but this is certainly not what happened.

Thus, consistency has two aspects: the transaction designer has to make sure that before and after a transaction is completed, the database is both in a state that all the integrity constraints are satisfied; and this state of the database correctly models the current state of the enterprise that it is modeling.

Homework: 2.7, and 2.8.

What does the TP monitor do?

The transaction monitor is responsible for ensuring *atomicity*, *durability*, and (the requested level of) *isolation*.

Atomicity means that the transaction either runs to its completion or it has no effect at all.

For example, in our registration application, a student either registered for a course, or did not. There can't be such a thing of partial registration, which will lead the database in an inconsistent state.

Durability means that once in, no information will be lost, unless it is explicitly taken out.

Thus, once registered, the system must keep this information.

The multiple transaction case

There is often a need to run many transactions together. For example, when withdrawing money, we get to check the balance, compare that with the amount of withdraw, if everything checks out, then make a deduction

Isolation deals with such a case.

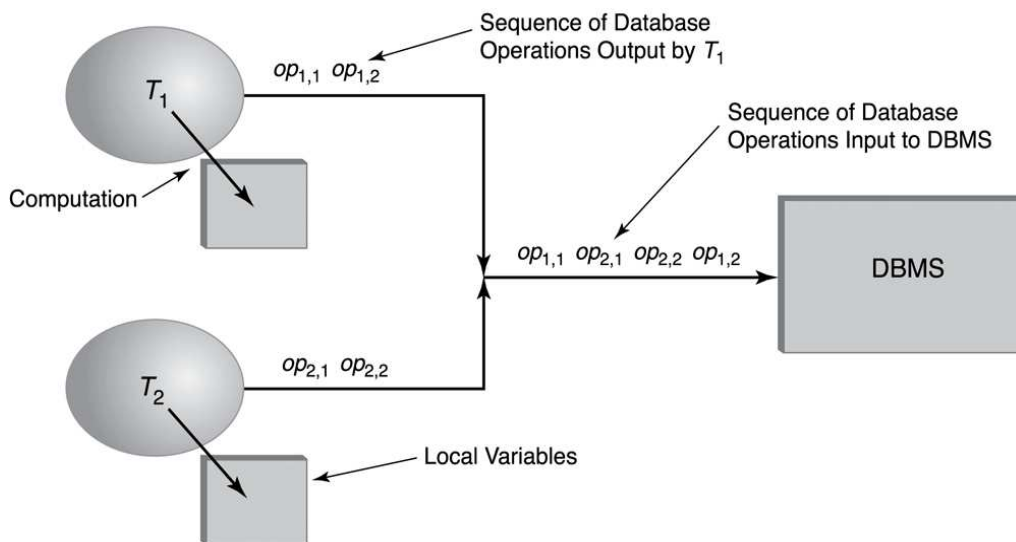
We say a set of transactions are executed serially, we mean that one is completely done before another starts.

When a set of transactions run this way, any consistent state will lead to another consistent state if all of them in this set are consistent.

Now the bad news.

The bad news is that many applications have strict requirement demand such as minimal response time and largest throughput, thus it can't afford to run serially.

Instead, the transactions in a set have to run *concurrently*, i.e., the parts of those transactions will run in an interleaving way, as determined by a transaction schedule.



So what?

When transactions are executed concurrently, the consistency of the individual transactions can't guarantee the consistency of the set as a whole. For example, assume that the cap for CS3600 is 25, and the current number is 24. If two students come to register at about the same time, and the transaction schedule is the follows, the constraint will be violated.

Trans. A	time	Trans. B
-	↓	-
Retrieve <i>cur_reg</i>	t_1	-
-	↓	-
-	t_2	Retrieve <i>cur_reg</i>
-	↓	-
Update <i>cur_reg</i>	t_3	-
-	↓	-
-	t_4	Update <i>cur_reg</i>
-	↓	-

The *isolation* rule now kicks in

Thus, we have to require that even it can be done concurrently, it has to do it correctly. In other words, the overall effect of the schedule must be the same as if the transactions had executed serially in some order.

Concurrent schedules that satisfy this condition is called *serializable*.

Those four conditions are collectively referred to as the *ACID* property. If it is satisfied, then the database will be a consistent and up-to-date model of the real world, and the transactions will always send back correct and up-to-date information.

Homework: 2.11.