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 remotely. More specifically, we could have the following statement of objectives:

1. Authenticate faculty and students as users of the system. (login script?)
2. Add and drop course. (Associated scripts)
3. Obtain status reports on a particular student. (Associated scripts)
4. Maintain information about students and courses (Course, Student).
5. Enter final grades for those who complete courses (Professor, Teaching, Transcript).

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

We will use this application 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). The goal is that the final product does meet users's needs.

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

We will discuss the operational details in later units.
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, registration related data in this case.

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

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Freshman</td>
</tr>
<tr>
<td>2222</td>
<td>Mary Smith</td>
<td>1 Lake St.</td>
<td>Freshman</td>
</tr>
<tr>
<td>1234</td>
<td>Joe Blow</td>
<td>6 Yard Ct.</td>
<td>Junior</td>
</tr>
</tbody>
</table>
Tables and their company

An RDB table is a set of rows, thus, by the definition, it contains no duplicates, and the order of those rows does not matter, conceptually speaking.

For example, the previous table contains three rows, each containing four columns. Each column (attribute) states a particular fact about each row and is associated with a domain (type).

As mentioned in the preview, this model is based on the relation concept, which 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. John lives there.
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.” (true), “It is raining outside.” (false), “3+5=9” (false), etc.

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

The *Closed World* assumption states that if it is not *there*, it is false. Then, the Student table collects all the true information about who lives where.

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..”.
How to create a table?

The creation of a table is 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));
```

Here is how we will do it in MySQL:

```
mysql> create table Student (  
    -> Id INT Not Null Primary key,  
    -> Name Char(20) Not Null,  
    -> Address Char(50),  
    -> Status Char(20) default 'freshman');
Query OK, 0 rows affected (0.00 sec)
```

**Homework:** Exercises 2.1, 2.2 and 2.4.
How to work with a table?

In a real situation, a table could be quite large. There are over 5,000 plus students here, thus this many rows in the current Student instance; and we also want to keep more information for each student, e.g., Age, GPA, etc., 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. (Cf. Exercise 2.4)

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.

For example, when a student just transferred in, we want to add her information into the Student table in degreeWorks.
...are based on mathematics.

The above operations are implemented via operations defined for mathematical relations.

Typically, a \textit{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 2017.

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, e.g., MA2250 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 as a combination of some basic relational operations. We can thus prove, beyond any doubt, reasonable or not, 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.

*Have you read through the Labwork Sampler yet?*

We will further discuss some of the related issues 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. Thus, SQL is a declarative language, but not a procedural one such as Python.

Select attributes
From table(s)
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 value equals to “111111111”.

In general, given such a query, the “Select” part lists the columns it wants, the “From” part lists the data sources, i.e., table(s), and the “Where” part specifies the conditions.

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

Considering the data volume, certain optimization is certainly involved, as we just discussed.

**Homework:** 2.3.
More on Select

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

```sql
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:

```sql
mysql> Select * From Student
    
        -> Where Status='senior';
```
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 will investigate other aggregated operators such as `avg`, `max`, `min`, `sum`, etc., later on.
More on Where

We can certainly put on more complex Boolean conditions, such as conjunctive (and (\(\land\))) or disjunctive (or (\(\lor\))) conditions.

For example,

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

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

```sql
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)
```

**Question:** Is there another way to say it?

**Answer:** How about this one?

```sql
mysql> Select Id, Name From Student
    -> Where Status='freshman' or Status='sophomore';
```
More on From

We can also get information from multiple tables.

```sql
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?

**Answer:** Cartesian product....
Optimization

A very important feature of SQL is that the programmer does not specify how to find those information. It is up to the DBMS to dig it out..., quickly with various indices, that we can specify with data bases.

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 because of the binary sort algorithm.

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 database and the real-world state by updating promptly.

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 to, and updating, a database.

An example of an integrity constraint is that “The number of students registered for a course cannot exceed the capacity of the room assigned for this course.”.

**Question:** What is the capacity of this room?
More stuff...

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

**Question:** Why do we need **IC1**?

**IC1:** The database contains a list of prerequisites for each course and, for each student, a list of completed courses. A student cannot 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, the *cap*, and the number of enrolled students currently registered for each course, which cannot be larger than the cap.
IC3: It might be possible to determine the number of students registered for a particular course in two ways: a number stored 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.

This is an example of the classic trade off between time and space.

Such integrity rules work as crazy glue to integrate tables into a related database, but not a loose collection of tables.

Some of them, the keys, are application independent, while the others, room capacity, depend on specific application.

Homework: 2.9.
Another aspect of consistency

Besides maintaining these rules, each transaction must update the database in such a way that the new instance 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 the current database instance agrees with 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 cannot be such a thing of “partial registration”, which will lead the database into an inconsistent state.

**Durability** means that once in, no information will be lost, unless it is explicitly deleted or dropped.

Thus, once registered, the system must keep this information.
The multiple transaction case

Recall that a transaction is a sequence of DB operations. 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.

There is often a need to run multiple transactions together, e.g., we have to allow multiple customers to withdraw “at the same time”.

We say a set of transactions are executed serially, or sequentially, 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.

*Isolation* is to deal with this situation because....
... we do have bad news.

The bad news is that many applications demand minimal response time and largest throughput, so it cannot afford to run sequentially.

In fact, transactions in a set run *concurrently*, i.e., the parts of those transactions will run in an interleaving way, as determined by a transaction schedule (More talk of this mess in OS.)

For the above case, there are six ($= \frac{4!}{2!2!}$) ways to carry them out, four of them will be *interleaved*, i.e., mixed, and all will lead to inconsistency. 😞
So what?

When transactions are executed concurrently, the consistency of the individual transactions cannot 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 consistency constraint will be violated as we will add in 26 students (?).

<table>
<thead>
<tr>
<th>Trans. A</th>
<th>time</th>
<th>Trans. B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve cur_reg</td>
<td></td>
<td>Retrieve cur_reg</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Update cur_reg</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

\[
\begin{array}{ccc}
\text{Trans. A} & \text{time} & \text{Trans. B} \\
\text{Retrieve \textit{cur\_reg}} & t_1 & \text{Retrieve \textit{cur\_reg}} \\
\text{Update \textit{cur\_reg}} & t_2 & - \\
\text{Update \textit{cur\_reg}} & t_3 & - \\
\text{Update \textit{cur\_reg}} & t_4 & - \\
\end{array}
\]
The *isolation* rule now kicks in

Thus, we have to require that, even it can be done concurrently, it has to do it correctly. (Correctness always takes the precedence over efficiency.) In other words, the overall effect of the schedule must be the same as if the transactions had been 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.