We now start the design part of an RDB. We first present an architecture for a database system all such databases share. The ANSI/SPARC architecture of RDB consists of three levels: internal, conceptual, and external levels.

At the internal level, everything is represented as a bunch of bits, then bytes, fields, records, tables, which can be accessed with indices. Conceptually speaking, a database is a collection of related tables. Externally, each application gets access to just a portion of such a database.

A database at the physical level, i.e., details of how records are kept in files are regarded as the physical level of data modeling, specified via a physical schema.
Physical level

Physical schema describes details of how data is stored: tracks, cylinders, indices etc. Early data-intensive applications worked directly with these physical schemas, since computers systems were quite slow at that time, and the applications were pretty primitive, and we really wanted to minimize the associated space.

A serious issue with this approach is that most of the work were hard-coded to deal with physical representation. Thus, changes to data structure are difficult to make, and application code becomes complex since it must deal with details. Finally, rapid implementation of new features is almost impossible.

The Y2k bug is certainly a good example in this regard. In this case, an abstraction of Date type should have solved the whole problem.
The conceptual level

When we have to support an application that needs frequent change and rapid implementation of new stuff, such as queries, a conceptual level of data modeling becomes necessary, which provides a data description in terms of what human can understand, without implementation details. For example,

```
Student(Id:INT,Name:String,Address:String,
       Status:String)
```

At this level, the entire information content of a database is represented “as is”. It is defined via a conceptual schema.

Besides the information content, the conceptual view also includes other stuff such as the security and integrity constraints.
The same as record?

It might look similar to the definition of a record for a file, but the information as described in this conceptual level is implementation independent. It might be stored differently physically from system to system, and they might even be stored in the same files.

This possibility of being portable makes the data modeling business much more flexible, and is referred to as the physical data independence.

Instead of working directly with the physical levels of the files, an application will only work with the conceptual level, and let the DBMS take care of the mapping between those two levels. If the physical implementation needs a change, e.g., size of int, the only thing that needs to change is the mapping itself inside DBMS, but not the application codes.
The external level

In the relational model, the *external level* is often specified in the same way as that for the conceptual level, with the difference being that there is only one conceptual level, but many different external views from different users.

An external schema specifies a *view* of the data from a user’s, or an application’s, perspective.

For example, with the *registration* database, for the Registrar, its external view might only consist of students’s academic information as reflected in the *Transcript* table, but not how much he owes to the university, which is taken care of by the external view associated with the Bursar office through a *FinantialHold* table.
More globally,...

Plymouth State presents a conceptual view to the outside world via, e.g., www.plymouth.edu and the annual *Academic Catalog*; and each student and faculty/staff has his/her own external view of the courses, the professors, the various offices, the dining hall, etc.

The Internal view is expressed via the budget book, the syllabi, the professors who actually teach each and every class, etc., and who knows where and how this stuff is kept.

The conceptual/internal mapping: People come and go, the budget changes all the time, but the conceptual view stays roughly the same over a certain period.
Another example: CS3600

The conceptual view could be the catalog description of this course, the internal one is a combination of the textbook, the professor, and the notes; while each of you has your own external perception of this course: its purpose, its usefulness, its difficulty degree, etc..

The conceptual/internal mapping: I now use a different book, thus the order of the subjects, and subjects covered in this course, even the programming languages, are different. Thus, the internal representation of this course has been changed.

Since the conceptual view has to remain the same, which calls for a different mapping. To get the same level of knowledge, you now have to make more efforts within a shorter period of time, based on one book plus two lab notes, etc..
Another independence

Applications tailored to the needs of a particular user group can be designed to use the external schemas appropriate for these groups, as provided by the conceptual level. At the current stage, such a content is really just a simple union of all of the individual external schemas (Think about that buffet table.)

The mapping between these two levels is again taken care of by the DBMS, so that any change to the conceptual level will have no impact on the user level. We will discuss later on how to implement such an independence with views.

As a result, applications are insulated from changes of both the conceptual level and the physical level. It thus needs no change when either or both of the latter two levels change over the years.

**Question:** What happens if such an independence does not exist... 😞
Worth thousands of words
Data model

A data model consists of a set of notions and languages to describe the following components:

1. Conceptual and external schemas, which describe the structure of data at some level (e.g., tables, attributes, constraints, domains), with a data definition language (DDL).

2. Constraints, which specifies a condition that the data items in the database must satisfy.

For example, no age value can be negative. There are four seasons, twelve months, .... We cannot rent out more DVDs than what we have, etc..
Data model (Cont’d)

3. *Operations on data*, as described via a *data manipulation language*. This is usually the most important and interesting part of any model. SQL is often the standard DML.

4. We sometimes can also use *Storage definition language* to write directives that influence the physical schema (affects performance, not semantics)

Notice we have yet to discuss anything about a data model implementation. Such a model was implemented in an hierarchical and/or network fashion, but it is now dominantly implemented in ....
The main attraction of this modeling technique is that it is built around a simple, powerful and solid concept of mathematical relation, when specifying and manipulating the databases.

Moreover, as we already went through, this solid and uniform background makes it possible that the so constructed mathematically based expressions (queries) can be analyzed by DBMS and further transformed to equivalent, but more efficient, expressions automatically in the process of query optimization.
Basic concepts

The central concept is that of a relation, with two parts: a schema (structure) and its current instance, sort of like a variable, its type and the current value. (Cf. Solution to Exercise 2.2)

A Relation instance is just a table with rows and named columns, which is actually what we will work with most of the time. Sometimes, we just call it a “relation” or a “table”.

The rows in a table are mathematically called tuples, and commonly known as records. It is important to know that all the rows in the same table share the same number of columns, and this number, mathematically, is called its arity.

Since a relation instance is a set of tuples, no two tuples can be same. Thus, it makes sense to talk about the number of rows in a table, which is called its cardinality, or the size of this set.
Just a bit more

Below shows a relation instance, or a table, that we have seen $n$ times already.

<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>

All the columns, also called *attributes*, come with their own names.

**Question:** Why the orders of the rows and columns are not important?

**Answers:** The order of rows is not important since a table is a set. The order of columns is not important since they are named.
A bit on domains

A domain is essentially a type. Just like any int value is taken from a set of integers, the value of a particular column of a table is taken from a set of attribute domain values. For example, the Address column of the Student relation is taken from a set of strings.

An important requirement for the values in a domain is the data atomicity, which really means that those values appear indivisible to the relational operators since there is no way to tell how they are actually implemented within the DBMS.

**Homework:** Dig a bit more out of data atomicity, then complete 3.1.
A big picture

Our goal in this unit is to show how we could integrate a collection of tables into a cohesive database, with the help of a variety of integrity constraints.

We will then show how we could get to such a collection of tables from a verbal description via the **Entity/Relationship** approach in the next unit, Unit 4.

In Unit 5, we will query and update such a database with the help of **relational algebra**.

Then, in Unit 6, we will get back to the process of how to simplify and minimize such a database through a **normalization** process.

After showing some real applications, we are ready for the big bird. 😊.
Now the schema

This rather conceptual stuff consists of a few pieces:

1. It must have a unique name across the database, e.g., Student. As we saw, it is case sensitive in terms of MySQL, neither student nor STUDENT will work. 😞

2. The name of an attribute has to come together with its domain, e.g., Id INTEGER. How much space do we have to kick in?

3. A collection of integrity constraints which puts some restrictions on its instances.

For example, the values of a particular attribute in all tuples are unique (primary key), and the values of a particular attribute, age, in all tuples are greater than 0.
An example

Is it the 1,001st times that have looked at the following stuff? 😊?

Student(Id:INT,Name:String,Address:String, Status:String)

it says that the Student relation could have exactly four attributes: Id, Name, Address, and Status, together with their associated data types of either INT or String.

Besides those standard domains, a user can also define her own domains, such as SSN. However, this feature depends on a specific DBMS implementation.

For example, even the most recent version of MySQL (5.7) does not allow us to do just that.
A bit more formally,…

given a relation schema $S$ and one of its instances $s$, then a *type constraint* specifies that $s$ must satisfy the following two conditions:

1. Each column in $s$ must correspond to an attribute in $S$, and shares the same name.

   You cannot *use* something that is not *defined* in the table structure.

2. For each attribute-domain pair in $S$, the values that appear in a column in $s$ must belong to the domain of the corresponding attribute in $S$.

   Each value in the table (a database instance) has to stick to its type as defined in the structure. Type consistency?
An example

Thus, the schema

\[
\text{Student}(\text{Id:INT, Name:String, Address:String, Status:String})
\]

specifies the following as an instance

<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>

To summarize, a RDB consists of two things: a \textit{database schema}, sort of a type specification; and a \textit{database instance}, the data in it \textit{at the moment}.

\[\text{int } i=5;\]
Our running example

We usually set up the schema only once, but we may revise it later, but use the associated instances quite a bit, which is typically called the database.

We will work the Registration database as a running example throughout this course.

Let’s have a look at its schema, and one of its ancient instances 😐.
A collection (I)

Below collects the database schemas that we are going to use with the Registration database, *without any constraints attached*.

Student (Id: INT, Name: STRING, Address: STRING, Status: STRING)

Professor (Id: INT, Name: STRING, DeptId: DEPTS)

Department(DeptId: DEPTS, Name: STRING)

Course (DeptId: DEPTS, CrsName: STRING, CrsCode: COURSES, Descr: STRING)

Transcript (CrsCode: COURSES, StudId: INT, Grade: GRADES, Semester: SEMESTERS)

Teaching (ProfId:Integer, CrsCode:String, Semester:String)
A database instance

<table>
<thead>
<tr>
<th>Professor</th>
<th>Id</th>
<th>Name</th>
<th>DeptId</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>101202303</td>
<td>John Smyth</td>
<td>CS</td>
</tr>
<tr>
<td></td>
<td>783432188</td>
<td>Adrian Jones</td>
<td>MGT</td>
</tr>
<tr>
<td></td>
<td>121232343</td>
<td>David Jones</td>
<td>EE</td>
</tr>
<tr>
<td></td>
<td>864297531</td>
<td>Qi Chen</td>
<td>MAT</td>
</tr>
<tr>
<td></td>
<td>555666777</td>
<td>Mary Doe</td>
<td>CS</td>
</tr>
<tr>
<td></td>
<td>009406321</td>
<td>Jacob Taylor</td>
<td>MGT</td>
</tr>
<tr>
<td></td>
<td>900120450</td>
<td>Ann White</td>
<td>MAT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course</th>
<th>CrsCode</th>
<th>DeptId</th>
<th>CrsName</th>
<th>Descr</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS305</td>
<td>CS</td>
<td>CS</td>
<td>Database Systems</td>
<td>On the road to high-paying job</td>
</tr>
<tr>
<td>CS315</td>
<td>CS</td>
<td>CS</td>
<td>Transaction Processing</td>
<td>Recover from your worst crashes</td>
</tr>
<tr>
<td>MGT123</td>
<td>MGT</td>
<td>MGT</td>
<td>Market Analysis</td>
<td>Get rich quick</td>
</tr>
<tr>
<td>EE101</td>
<td>EE</td>
<td>EE</td>
<td>Electronic Circuits</td>
<td>Build your own computer</td>
</tr>
<tr>
<td>MAT123</td>
<td>MAT</td>
<td>MAT</td>
<td>Algebra</td>
<td>The world where $2 \times 2 \neq 4$</td>
</tr>
</tbody>
</table>
### Transcript

<table>
<thead>
<tr>
<th>STUDID</th>
<th>CRSCode</th>
<th>SEMESTER</th>
<th>GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>666666666</td>
<td>MGT123</td>
<td>F1994</td>
<td>A</td>
</tr>
<tr>
<td>666666666</td>
<td>EE101</td>
<td>S1991</td>
<td>B</td>
</tr>
<tr>
<td>666666666</td>
<td>MAT123</td>
<td>F1997</td>
<td>B</td>
</tr>
<tr>
<td>987654321</td>
<td>CS305</td>
<td>F1995</td>
<td>C</td>
</tr>
<tr>
<td>987654321</td>
<td>MGT123</td>
<td>F1994</td>
<td>B</td>
</tr>
<tr>
<td>123454321</td>
<td>CS315</td>
<td>S1997</td>
<td>A</td>
</tr>
<tr>
<td>123454321</td>
<td>CS305</td>
<td>S1996</td>
<td>A</td>
</tr>
<tr>
<td>123454321</td>
<td>MAT123</td>
<td>S1996</td>
<td>C</td>
</tr>
<tr>
<td>023456789</td>
<td>EE101</td>
<td>F1995</td>
<td>B</td>
</tr>
<tr>
<td>023456789</td>
<td>CS305</td>
<td>S1996</td>
<td>A</td>
</tr>
<tr>
<td>111111111</td>
<td>EE101</td>
<td>F1997</td>
<td>A</td>
</tr>
<tr>
<td>111111111</td>
<td>MAT123</td>
<td>F1997</td>
<td>B</td>
</tr>
<tr>
<td>111111111</td>
<td>MGT123</td>
<td>F1997</td>
<td>B</td>
</tr>
</tbody>
</table>

### Teaching

<table>
<thead>
<tr>
<th>PROFID</th>
<th>CRSCode</th>
<th>SEMESTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>009406321</td>
<td>MGT123</td>
<td>F1994</td>
</tr>
<tr>
<td>121232343</td>
<td>EE101</td>
<td>S1991</td>
</tr>
<tr>
<td>555666777</td>
<td>CS305</td>
<td>F1995</td>
</tr>
<tr>
<td>864297531</td>
<td>MGT123</td>
<td>F1994</td>
</tr>
<tr>
<td>101202303</td>
<td>CS315</td>
<td>S1997</td>
</tr>
<tr>
<td>900120450</td>
<td>MAT123</td>
<td>S1996</td>
</tr>
<tr>
<td>121232343</td>
<td>EE101</td>
<td>F1995</td>
</tr>
<tr>
<td>101202303</td>
<td>CS305</td>
<td>S1996</td>
</tr>
<tr>
<td>900120450</td>
<td>MAT123</td>
<td>F1997</td>
</tr>
<tr>
<td>783432188</td>
<td>MGT123</td>
<td>F1997</td>
</tr>
<tr>
<td>009406321</td>
<td>MGT123</td>
<td>F1997</td>
</tr>
</tbody>
</table>
Integrity constraints

Such a constraint (IC) makes a statement about all the valid instances of a database. *They glue the tables together.*

Some of them are based on certain business rules, e.g., “No employees should make more than what her boss does.” 😞, and are often listed in the requirement document (Cf. §14.2)

Other constraints, such as type and domain constraints, are based on the schema design, and specified by the database designers.

Since they are part of schema, they are typically given in the schema design. They can also be added, deleted, and updated later when more information becomes available.

Once in, they are enforced by the DBMS.
Various kinds of IC

An IC can be either *intra-relational*, when only one relation is involved; thus given in a relational schema, e.g., all Id values are unique (the key constraint); or it can be *inter-relational*, where several relations are involved, thus it has to be specified in a database schema. For example, every employee of the ITS department must be an employee of the university (the “foreign key” constraint).

Some of the constraints can be *dynamic*, in the sense that they put a restriction on the evolution of valid instances. This is particularly useful when enforcing business rules. For example, “Salary must not change more than 5% 😊 per transaction”; or “The marital status cannot be changed from single to divorced (?)”.

Again, support for such rules depends on a particular DBMS system.
Key constraint

We already saw an example: value of the attribute Id must be unique in any instance of the Student relation.

For a more complicated case, in the Transcript relation, neither StudId or (StudId, CrsCode) is necessarily unique.

StudId cannot be unique since a student usually takes several courses.

(StudId, CrsCode) is not unique either since one might fail a course, thus has to take it again.

<table>
<thead>
<tr>
<th>StudId</th>
<th>CrsCode</th>
<th>Semester</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>CS3600</td>
<td>F2016</td>
<td>F</td>
</tr>
<tr>
<td>1111</td>
<td>CS3600</td>
<td>F2017</td>
<td>A-</td>
</tr>
<tr>
<td>1234</td>
<td>CS3600</td>
<td>F2016</td>
<td>B+</td>
</tr>
</tbody>
</table>
What should be unique?

Nevertheless, (StudId, CrsCode, Semester) has to be unique for all the tuples. Let’s assume it is not, then, at least two rows in some instance will agree on these three attributes. For example,

<table>
<thead>
<tr>
<th>StudId</th>
<th>CrsCode</th>
<th>Semester</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>CS3600</td>
<td>F2016</td>
<td>B</td>
</tr>
<tr>
<td>1111</td>
<td>CS3600</td>
<td>F2016</td>
<td>C</td>
</tr>
<tr>
<td>1234</td>
<td>CS3600</td>
<td>F2016</td>
<td>B+</td>
</tr>
</tbody>
</table>

Since any instance of this Transcript table, like any RDB table, has to be a set, thus contains no duplicates. Since these two rows agree on the first three attributes, they must disagree on the last piece, i.e., Grade. In this case, 1111 gets two different grades when taking the same course in the same semester, a clearly violation of common sense.

Hence, every tuple in this table must have different values in this triplet.
Is uniqueness enough?

A key must have this uniqueness property, i.e., no two values of such a group of attributes could be the same.

**Question:** Is it the only feature that we want for a key?

Since any instance of a RDB table is a set, no two rows of any instance can be the same. Thus, the group of all the attributes of any table must have this uniqueness property.

Should we use the whole thing as a key? We could, but often don’t want to, since it might be too big to work with.

In fact, out of all the unique identifiers, we choose the minimum identifier, i.e., a minimum set that uniquely identify all the tuples of this table as a key for this table.
A bit more formally...

A key constraint of a relation schema, $S$, is a sequence of attributes $A_1, \ldots, A_n, n \geq 1$, in $S$, with the following property: 1). **Uniqueness:** Given a relation instance $s$ of $S$, **at most one** (?) row in $s$ can contain a particular set of values, $a_1, \ldots, a_n$, for the attributes $A_1, \ldots, A_n$.

2). **Minimality:** No subset of $A_1, \ldots, A_n$ is a key constraint.

A Key is a set of attributes mentioned in a key constraint.

For example, (StudId, CrsCode, Semester) in the Transcript table is a key. In particular, it is a minimal unique identifier, as none of its subsets is also a key. (?) Thus, it could not be smaller 😊.
On the minimality issue

**Question:** Why should \((\text{Id}, \text{Name})\) not be a key of the \textit{Student} table, although it is a unique identifier?

**Answer:** Since in that table, \((\text{Id})\) itself already uniquely identifies the tuples.

**Question:** Why do we include the minimality requirement?

**Answer:** To make it better. For example, in this case, if we use \((\text{Id}, \text{Name})\), but not \((\text{Id})\) as the key, we \textit{could but must not} have the following instance, which beats the purpose of using \textit{Id} as a unique identifier.

<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>1111</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>

Minimality is interesting such as the minimum drinking age, minimum wage, minimum daily balance, etc..
A couple of points

1. A superset of a key must be a unique identifier: If key($K$) is a key constraint in schema $S$ and $K \subseteq L$, then legal instances of $S$ cannot have distinct tuples that agree on every attribute in $L$. Indeed, given two legal rows $s$ and $t$, if they agree on everything in $L$, they must agree on everything in $K$. Thus, $K$ wouldn’t be a key, as assumed. 😞

We thus call such an $L$ a super key, which is not always a key, since if $K \subset L$, $L$ does not satisfy the minimality property.

2. Every relation has a super key. Just take the whole set of attributes, but it usually is not used as the key (?)

3. A schema can have more than one keys.

**Homework:** 3.2 and 3.3.
Multiple keys

When a schema have multiple keys, we call them *candidate keys*. One of them is then designated as the *primary key*.

A primary key might not have any semantic significance, but commercial DBMS often use the primary key as a hint in optimizing the storage structures to enable more efficient access to the data whenever such a primary key value is given. As we said earlier, all the rows in an instance will be sorted in terms of the primary key.

Technically, a DBMS automatically creates an index file for a primary key. Then, we can use binary search ($O(\log n)$), instead of sequential search ($O(n)$), to look for things.

Recall that, when $n = 10^6$, the difference is between 20 and half of a million.
A collection (II)

Below collects the database schemas for the registration system together with their keys.

Student (Id: INT, Name: STRING, Address: STRING, Status: STRING)
  Key: {Id}
Professor (Id: INT, Name: STRING, DeptId: DEPTS)
  Key: {Id}
Course (DeptId: DEPTS, CrsName: STRING, CrsCode: COURSES, Descr: STRING)
  Key: {CrsCode}, {DeptId,CrsName}
Transcript (CrsCode: COURSES, StudId: INT, Grade: GRADES, Semester: SEMESTERS)
  Key: {StudId,CrsCode,Semester}
Department(DeptId: DEPTS, Name: STRING)
  Key: {DeptId}
Teaching (ProfId:Integer, CrsCode:String, Semester:String)
  Key: {CrsCode,Semester}

What does the last key imply for Teaching?
Referential integrity

In a relational database, it is often the case that the value of one table refers to the values of another related table. For example, the professor ID in the Teaching table is the same as that in the Professor table. In many cases, it will be an integrity violation if such a reference does not exist.

For example, if, for a given professor ID in the Teaching table, we cannot find the same value in the Professor table. It might mean that we have yet to hire her yet, how could we let her teach?

Such a requirement that the referenced tuples must exist in the referenced table is called a referential integrity. The foreign-key constraint is an important example of such integrity constraints.
A bit more formality

Let $S$ and $T$ be two relational schemas, $F$ be a list of attributes in $S$, and $\text{key}(K)$ be a key constraint in $T$.

We say that relation instances $s$ for $S$ and $t$ for $T$ satisfy the foreign-key constraint, “$S(F)$ references $T(K)$” and that $F$ is a foreign key of $S$ if and only if, for any row $x$ in $s$, there is a row $y$ in $t$, such that $x(F) = y(K)$.

What does this mean? 😊
In the above example, every value of $D$ in table $T_1$ refers to a value of $E$ in table $T_2$. Thus, $D$ is a foreign key of $T_1$, referring to $E$ in $T_2$, in other words, “$T_1(D)$ refers to $T_2(E)$”.

This association only needs to be a one-way street. For example, there is nothing in $T_1$ for $v7$.

Indeed, a professor has been hired, but she has not been allocated to teach any course. Could she be a research faculty, a clinic faculty? 😊
Something special

In a foreign-key constraint, the referring and the referred relations need not be distinct. For example, *every manager is also an employee.*

Thus, for the following relational schema, where Id serves as the primary key of the Employee table,

\[
\text{Employee(}\text{Id:INT,Name,String,MngrID:INT)}
\]

we will add on the following constraint:

\[
\text{Employee(MngrId) references Employee(Id)}
\]

Even a manager needs to be paid, just like every employee... 😊
A collection (III)

Transcript(StudId) references Student(Id)

“Every student who has been taking a class is enrolled, i.e., paid.”

Transcript(CrsCode) references Course(CrsCode)

Teaching(ProfId) references Professor(Id)

Teaching(CrsCode) references Course(CrsCode)

Transcript(CrsCode, Semester) references Teaching(CrsCode, Semester)

Question: What does the above FK requirement specify?

Answer: You cannot cook your transcript. 😊
Other referential constraints

We might want to enforce the rule that “no professor teaches an empty class.”

Teaching(CrsCode,Semester)
    references Transcript(CrsCode,Semester)

The above essentially states that for a specific (CrsCode,Semester) pair of values in every row in the Teaching table, there must exist a row in the Transcript table with the same pair of values. In other words, at least one student has taken or is taking this course.

**Question:** Is this IC a foreign key requirement?

**Answer:** No, since this pair of attributes is not a candidate key in the Transcript table.
Semantic constraints

The constraints that we have seen so far deal with the *structure* of data. They must be held no matter what will be entered as its instances.

Others are intended to implement certain business rules, or rather dependent on instances. We refer to them as *semantic rules*.

For example, the number of students registered for a course could not be more than the capacity of the classroom, where the class is schedule to meet. No one makes more than her buss does, etc..

**Homework:** 3.4, 3.5 and 3.7.

Have you started **Project (I)** yet?
What have we got?

Hopefully, we have turned a bunch of separate sandy particles into a piece of sturdy rock. 😊.

Now that we know what to do, we will see how to do it, i.e., how to use SQL statements to specify all these stuff so that a computer “understands” them, as well.
How do we do it?

We will talk about things based on the SQL-92 standard, although most database vendors do not fully support it. We will even peek into the newer and future versions, such as SQL-1999 and SQL-2003, with the trigger and some object-oriented features being the most important features.

Once we are done with the talk, we will walk through the implementation details, in MySQL, with Part (II) of the lab notes, *A Gentler Introduction to MySQL Database Programming*.

In particular, we will convert the Registration database into a MySQL database.
Specify the types

The type of the Student table is given as follows, in SQL

Create table Student (  
    Id Integer,  
    Name Char(20),  
    Address Char(20),  
    Status Char(20))

The specification for the other tables are just as simple.

**Homework:** 3.8, 3.9, 3.12 and 3.15.
System catalog

The DBMS must have a way to get all such information when translating a DML statement into an executable program later. It thus collects all the relevant information in a system catalog, sort of a huge collection about the structure of all the tables it has seen for far.

Such a catalog is actually a collection of stuff. For example, the Columns table collects the information of every attribute of all the tables the system has seen so far.

Notice that the attributes of this Column table is also entered in itself. Such a bootstrapping process is initiated automatically when a new database is created.

In MySQL, such stuff are collected in a database, InformationSchema.
Create key constraint

It is also easy to put this piece in.

Create table Course (  
    CrsCode    Char(6),  
    DeptId     Char(4)  
    CrsName    Char(20),  
    Descr      Char(100),  
    Primary key (CrsCode),  
    //Candidate key  
    Unique     (DeptId,CrsName))

The specification for the other tables are just as simple.

**Labwork:** We will *soon* set them all up in next week, following Section 2 of Part (II) of the lab notes, *A Gentler Introduction to MySQL Database Programming.*
Where is the beef?

A relation consists of rows, which in turn consists of known values. But, when we enter such a row, it might not have all the values. For example, when we enter a student, she might not have a major yet. We thus have a special value holder, null, and use it to fill some of the values we don’t know yet, and later on replace it with something more specific, when we get them.

But, we can’t afford to do it all the time. For example, we can’t give a null as the value for any primary keys. We thus could do something like the following:

Create table Student (  
  Id Integer,  
  Name Char(20), Not null  
  Address Char(20),  
  Status Char(20)) Default 'freshman',  
  Primary key (Id))
How to add semantic constraints?

This is specified using the `check` clause, when a conditional expression is checked to see if the condition is met. When being placed inside a `Create table` clause, it is used as an intra-relational constraint on that relation only, thus every row of that table has to meet its requirement, but nothing else.

For example,

```sql
Create table Transcript (  
    StudId   Integer,  
    CrsCode Char(6),  
    Semester Char(6),  
    Grade    Char(1),  
    Check (Grade in ('A','B','C','D','F')),  
    Check (StudId>0 AND StudId<100000))
```
An issue

What happens if there is nothing to check?

*It would be declared that the condition is satisfied.* For example, in the following:

Create table Employee (  
  Id Integer,  
  Name Char(20),  
  Salary Integer,  
  DeptId Char(4),  
  MagrID Integer,  
  Check (0<(Select Count(*) From Employee)),  
  Check ((Select Count(*) From Manager) < (Select Count(*) From Employee))

When the Employee table is empty, the first check actually go through, so does the second.

**Question:** What happens if the very first tuple is a manager?
More general stuff

SQL provides a general mechanism of Assertio{}n to specify a general constraint. For example,

Create Assertion ThouShaltNotFireEveryone
  Check (0<(Select Count(*)
    From Employee))

Create Assertion WatchAdminCosts
  Check ((Select Count(*)
    From Manager)
  < (Select Count(*)
    From Employee))

While the first constraint is for one table only, the second has an inter-relational nature, and must be satisfied by both tables.

**Question:** Why do we give it a name?

**Answer:** We can take it out, (drop), if it is no longer needed....
How bad could it be?

Assume that we keep salary information of employees and managers separately, and we want to make sure that no one will get more than his boss.

Create Assertion ThouShaltNotOurEarnYourBoss
Check (Not Exists
(Select * From Employee, Manager
 Where Employee.Salary>Manager.Salary
 AND Employee.MngrID=Manager.ID))

Don’t feel bad if you don’t understand the details. We will get to this later.

**Question:** Can we do it with MySQL?

**Answer:** Not in MySQL 5.7.
User-defined domains

SQL provides a way to define your own domain by making use of the Check facility. For example,

Create Domain Grades Char(1)
  Check (Value in ('A','B','C','D','F','I'))

We can then use the domain Grade as usual.

Below is another example

Create Domain UpperDivisionStudents Integer
  Check (Value in (Select Id From Student
      Where Status in ('senior', 'junior')
      And Value is not null))

This feature has yet to be implemented in MySQL.
Foreign-key constraints

It is pretty straightforward.

Create table Teaching (  
  ProfId Integer,  
  CrsCode Char(6),  
  Semester Char(6),  
  Primary key (CrsCode,Semester),  
  Foreign key (CrsCode)  
    references Course,  
  Foreign key (ProfId)  
    references (Professor(Id))  
)

If it refers to the same table, the reference part can be dropped.
MySQL implementation

Starting with version 5.5., MySQL has added the syntax of foreign key constraints, but it is not real....

When creating new tables where you will place foreign keys, do this.

```sql
mysqld> Create Table TableOne (  
    someNumber Integer Not Null,  
    someStuff Char(10),  
    Primary key (someNumber)),  
    Foreign key (someStuff)  
        references TableTwo(stuffId)  
) ENGINE = INNODB;
```

For tables with foreign keys declared, you have to use the following to let them take effect.

```sql
mysql> Alter table <tablename> ENGINE = INNODB;
```

Check out the further reading item at the end of this chapter....
Chicken and egg

Let’s assume the following assumption:

Create table Department (  
  DeptId    Char(4),  
  Name      Char(40),  
  Budget    Integer,  
  MngrId    Integer,  
  Foreign key (MngrId)  
    references Employee(Id))

**Question:** What happens when we enter this one before the Employee table?

**Answer:** What do you mean by “enter”? You can certainly enter the structure, but you cannot *populate* the Department table before populating the Employee table... since it would violate the foreign key, as the Employee table is empty, thus nothing to refer to.
What to do?

A solution: Create and then populate the Employee table first.

Another issue: There might be a foreign key in the Employee table, saying she must belong to a valid department.

Alter table Employee
  Add constraint EmpDeptConstr
    Foreign key (DepartmentId) references Department(DeptId)

Thus, we can’t fill up Employee table first, either. This is a typical deadlock (We will discuss a lot more about deadlock in CS4310).

Solution: Drop the foreign key constraint from the Department table, fill it up, add the foreign key back, and then fill the Employee table.
Reactive constraints

When a constraint is found violated, the corresponding transaction is usually aborted. However, some remedial actions might be more appropriate.

Assume that a row (0007, MGT123, F1994) is added to the Teaching table, and assume that the Professor table does not have a ProfId of 0007, then this insertion is simply rejected.

When constraint is violated because of a deletion, SQL provides more options.

When a professor leaves the university, thus triggers a violation, one thing we can do is to replace the ProfId value in the Teaching table with null; or we can add in a constraint saying that if a referencing tuple exits, it can’t be deleted. Of course, another one is that all the referencing rows will be deleted.
An example

Create table Teaching (  
    ProfId Integer,  
    CrsCode Char(6),  
    Semester Char(6),  
    Primary Key (CrsCode,Semester),  
    Foreign key (ProfId)  
        reference Professor (Id)  
            On delete No action  
            On Update Cascade,  
    Foreign key (CrsCode)  
        references Course(CrsCode)  
            On Delete Set null  
            on Update Cascade)

Here, we have provided four triggers for potential deletion and updates.

For example, when a professor record is deleted, all the teaching records will stay same; but when the Id part of a professor record is updated, all the referencing rows in the Teaching table will have that piece updated, too.
General triggers

SQL also provides a general mechanism to deal with similar, but more general situation. For example, whenever someone changes the course and/or semester in the Transcript table, then the associated grade has to be set to null.

In other words, if, after such a change, the grade is not set null, then this change will not be allowed:

Create trigger CrsChangeTrigger
   After update of CrsCode, Semester
   on Transcript
   When (Grade is not null)
   Rollback

This one has been implemented in MySQL, and it should be part of your project. Hopefully, we will have time to talk about it further in a later chapter.
Database views

We once talked about three levels of abstraction of a data model. In particular, the external level is an individual user’s perspective of the database. In SQL, such a level is accomplished with the Create View clause.

From a user’s perspective, a view is just like a table, which can be queried, updated, or its access can be controlled.

But, a view is not a table in the sense that it is not physical, i.e., it does not have its own space. All its values are derived from other tables. It is just a repackage of information stored somewhere else.

Check out Fun Reading in the course page for various views of the same Mt. Washington.
An example

Below defines a view based on some of the existing tables, of the pair of professors and students such that the student took the professor’s course at some point.

Create view ProfStud(Prof,Stud) As
    Select Teaching.ProfId, Transcript.StudId
    From Transcript,Teaching
    Where Transcript.CrsCode=Teaching.CrsCode
        And Transcript.Semester=Teaching.Semester

Once such a view is defined and entered into the system catalog, all the applications can use it just like it is an ordinary table.

This important feature was not implemented in MySQL until version 5. We will discuss its application further in a later chapter.

**Question:** How does the system know it is a view, but not a base table?
Change things

It is unavoidable that we will change our minds about something later on, particularly with a database which we might have to use for a long time. It is pretty easy for us to add in stuff.

```sql
Alter table Student
  Add Column Gpa Integer Default 0

Alter table Student
  Add Constraint GpaRange
  Check (Gpa>=0 And Gpa<=4)

Alter table Teaching
  Add Constraint TeachKey
  Unique (ProfId,Semester,Time)
```
Take stuff out

In order to take something out, you get to give it a name.

Create table Teaching ( 
    ProfId    Integer, 
    CrsCode   Char(6), 
    Semester  Char(6), 
    Constraint TrKey Primary Key (CrsCode,Semester), 
    Constraint ProfFK Foreign key (ProfId) 
        reference Professor (Id) 
    Constraint CrsFk  Foreign key (CrsCode) 
        references Course(CrsCode) 
    Constraint IdRange Check (ProfId>0 And ProfId<10000))

Then, you can do the following:

Alter table Transcript Drop Constraint TrKey
Access control

The creator of a table or other objects such as constraints or views are often assumed to be the owner and have all the access rights. The owner can grant some of these rights to the others via the grant clause.

Grant Select, Update(ProfId)
   On StuRegSystem.Teaching
   To JohSmyth, MaryDoe with Grant option

We can do the same to the views.

Grant select On ProfStud To alumnus

Access rights can be revoked with the Revoke clause.