Contents presented here have been taken from 2nd Chapter Introduction to Relational Databases of the book Database Systems An Application-Oriented Approach 2nd Edition by Michael Kifer, Arthur Bernstein and Philip M. Lewis

Introduction to Relational Databases

Table, Rows and columns

In Relational databases data is stored in tables.

For example, Student Registration System might include the STUDENT table.

Table 1: The table STUDENT. Each row describes a single student.

Id	Name	Address	Status
1111111	John Doe	423 Main St.	Freshman
666666666	Joseph Public	666 Hollow Rd.	Sophomore
111223344	Mary Smith	1 Lake St.	Freshman
987654321	Bart Simpson	Fox 5 TV	Senior
023456789	Homer Simpson	Fox 5 TV	Senior
123454321	Joe Blow	6 Yard Ct.	Junior

- A table contains a set of rows. Each row contains information about one student.
- Each column of the table describes the student in a particular way.
- In example above the columns are id, Name, Address, and Status.
- Each column has an associated type, called its domain, from which the value in a particular row for that column is drawn. For example the domain for Id is integer and the domain for Name is string.

This database model is called "relational" because it is based on the mathematical concept of relation.

A **mathematical relation** captures the notion that elements of different sets are related to one another.

For example, *John Doe*, an element of the set of all humans, is related to *123 Main St.*, an element of the set of all addresses, and to *111111111,* an element of the set of all Ids.

A relation is a set of **tuples**.

For example, of the table **STUDENT**, we might define a relation called **STUDENT** containing the tuple (1111111, John Doe, 423 Main St., Freshman).

Table 2: Correspondence between tables and relations

Tables	Relations
Rows	Tuples
Columns	Attributes

Operations on tables are mathematically defined

In most applications, the database is under the control of a database management systems (DBMS). When an application wants to perform an operation on the database, it does so by making a request to the DBMS.

A typical operation might

- extract some information from the rows of one or more tables,
- add rows, or
- delete rows.

In addition to the fact that tables in the database can be modeled by mathematical relations, operations on the tables can also be modeled as mathematical operations on the corresponding relations.

Thus, a particular unary operation might take a table, T, as an argument and produce a result table containing a subset of the rows of T.

For example:

1. an instructor might want to display the roster of students registered for a course. Such a request might involve scanning the TRANSCRIPT table, locating the rows corresponding to the course, and returning them to the application.

2. a particular binary operation might take two tables as arguments and construct a new table containing the union of the rows of the argument tables.

A complex query against a database might be equivalent to an expression involving many such relational operations involving many tables.

Because of this mathematical description, relational operations can be precisely defined and their mathematical properties, such as commutativity and associativity, can be proven.

This mathematical description has important practical implications. Commercial DBMSs contain a query optimizer module that converts queries into expression involving relational operations and then uses these mathematical properties to simplify those expressions and thus optimize query execution.

SQL:

Basic SELECT statement

An application describes the access that it wants the DBMS to perform on its behalf in a language supported by the DBMS. We are particularly interested in SQL, the most commonly used database language, which provides facilities for accessing a relational database and is supported by almost all commercial DBMSs.

The basic structure of the SQL statements for manipulating data is straightforward and easy to understand. Each statement takes one or more tables as arguments and produces a table as a result. For example, to find the name of the student whose Id is 987654321, we might use the statement

```
SELECT Name
FROM STUDENT
WHERE Id = 987654321
```

More precisely, this statement asks the DBMS to extract from the table named in the FROM clause—that is, the table STUDENT—all rows satisfying the condition in the WHERE clause—that is, all rows whose Id column has value 987654321—and then from each such row to delete all columns except those named in the SELECT clause—that is, Name. The resulting rows are placed in a result table produced by the statement. In this case, because Ids are unique, at most one row of STUDENT can satisfy the condition, and so the result of the statement is a table with one column and at most one row.

Thus, the FROM clause identifies the table to be used as input, the WHERE clause identifies the rows of that table from which the answer is to be generated, and the SELECT clause identifies the columns of those rows that are to be output in the result table.

The result table generated by this example contains only one column and at most one row. As a somewhat more complex example, the statement

```
SELECT Id , Name
FROM STUDENT
WHERE Status = 'senior'
```

returns a result table containing two columns and multiple rows:

Table 3: The database table returned by the SQL SELECT statement

Id	Name
987654321	Bart Simpson

Id	Name
023456789	Homer Simpson

the Ids and names of all seniors. If we want to produce a table containing all the columns of STUDENT but describing only seniors, we use the statement

```
SELECT *
FROM STUDENT
WHERE Status = 'senior'
```

The asterisk is simply shorthand that allows us to avoid listing the names of all the columns of STUDENT. In some situations the user is interested not in outputting a result table but in information about the result table. An example is the statement

```
SELECT COUNT(*)
FROM STUDENT
WHERE Status = 'senior'
```

which returns the number of rows in the result table (i.e., the number of seniors). COUNT is referred to as an aggregate function because it produces a value that is a function of all the rows in the result table. Note that in this case, the SELECT statement produces a table that has only one row and one column.

The WHERE clause is the most interesting component of the SELECT statement, it contains a general condition that is evaluated over each row of the table named in the FROM clause. Column values from the row are substituted into the condition, yielding an expression that has either a true ora false value. If the condition evaluates to true, the row is retained for processing by the SELECT clause and then stored in the result table. Hence, the WHERE clause acts as a filter.

Conditions can be much more complex than we have seen so far: A condition can be a Boolean combination of terms. If we want the result table to contain information describing seniors whose Ids are in a particular range, for example, we might use

```
WHERE Status = 'senior' AND Id > '888888888'
```

OR and NOT can also be used. Furthermore, a number of predicates are provided in the language for expressing particular relationships. For example, the IN predicate tests set membership.

```
WHERE Status IN ('freshman',    sophomore   ')
```

Additional aggregates and predicates and the full complexity of the WHERE clause will be discussed separately in upcoming chapters.

Multi-table SELECT statements

The result table can contain information extracted from several base tables. Thus, if we have a table TRANSCRIPT with columns StudId, CrsCode, Semester, and Grade, the statement

```
SELECT Name, CrsCode, Grade
FROM STUDENT, TRANSCRIPT
WHERE StudId = Id AND Status = 'senior'
```

can be used to form a result table in which each row contains the name of a senior, a particular course she took, and the grade she received.

The first thing to note is that the attribute values in the result table come from different base tables: Name comes from STUDENT; CrsCode and Grade come from TRANSCRIPT. As in the previous examples, the FROM clause produces a table whose rows are input to the WHERE clause. In this case the table is the Cartesian product of the tables listed in the FROM clause: a row of this table is the concatenation of a row of STUDENT and a row of TRANSCRIPT. Many of these rows make no sense. For example, Bart Simpson's row in STUDENT is not related to a row in TRANSCRIPT describing a course that Bart did not take. The first conjunct of the WHERE clause ensures that the rows of TRANSCRIPT for a particular student are associated with the appropriate row of STUDENT by matching the Id values of the rows of the two tables. For example, if TRANSCRIPT has a row *(987654321, CS305, F1995, C)*, it will match only Bart Simpson's row in STUDENT, producing the row *(Bart Simpson, CS305, C)* in the result table.

Query optimization.

One very important feature of SQL is that the programmer does not have to specify the algorithm the DBMS should use to satisfy a particular query.

For example, tables are frequently defined to include auxiliary data structures, called indices, which make it possible to locate particular rows without using lengthy searches through the entire table. Thus, an index on the Id column of the STUDENT table might contain a list of pairs (Id, pointer) where the pointer points to the row of the table containing the corresponding Id. If such an index were present, the DBMS would automatically use it to find the row that satisfies the query [query1]. If the table also had an index on the column Status, the

DBMS would use that index to find the rows that satisfy the query ([sql2]). If this second index did not exist, the DBMS would automatically use some other method to satisfy ([sql2])—for example, it might look at every row in the table in order to locate all rows having the value senior in the Status column. The programmer does not specify what method to use—just the condition the desired result table must satisfy.

In addition to selecting appropriate indices to use, the query optimizer uses the properties of the relational operations to further improve the efficiency with which a query can be processed—again, without any intervention by the programmer. Nevertheless, programmers should have some understanding of the strategies the DBMS uses to satisfy queries so they can design the database tables, indices, and SQL statements in such a way that they will be executed in an efficient manner consistent with the requirements of the application.

Changing the contents of tables

The following examples illustrate the SQL statements for modifying the contents of a table. The statement

```
UPDATE STUDENT
SET Status = 'sophomore'
WHERE Id = '111111111'
```

updates the STUDENT table to make *John Doe* a *sophomore*. The statement

```
INSERT
INTO STUDENT (Id, Name, Address, Status)
VALUES ('999999999', 'Winston␣Churchill', '10␣Downing␣St', 'senior')
```

inserts a new row for *Winston Churchill* in the *STUDENT* table.

The statement

```
DELETE
FROM STUDENT
WHERE Id = '1111111111'
```

deletes the row for *John Doe* from the *STUDENT* table. Again, the details of how these operations are to be performed need not be specified by the programmer.

Creating tables and specifying constraints

Before you can store data in a table, the table structure must be created. For instance, the STUDENT table could have been created with the SQL statement

```
CREATE TABLE STUDENT (
  Id INTEGER,
  Name CHAR(20),
  Address CHAR(50),
  Status CHAR(10),
 PRIMARY KEY (Id) )
```

where we have declared the name of each column and the domain (type) of the data that can be stored in that column. We have also declared the Id column to be a primary key to the table, which means that each row of the table must have a unique value in that column and the DBMS will (most probably) automatically construct an index on that column. The DBMS will enforce this uniqueness constraint by not allowing any *INSERT* or *UPDATE* statement to produce a row with a value in the *Id* column that duplicates a value of Id in another row. This requirement is an example of an integrity constraint (sometimes called a consistency constraint)an application-based restriction on the values that can appear as entries in the database.

We have given simple examples of each statement type to highlight the conceptual simplicity of the basic ideas underlying SQL, but be aware that the complete language has many subtleties. Each statement type has a large number of options that allow very complex queries and updates. For this reason, mastery of SQL requires significant effort.