Structured Query Language - SQL

The concept of a relational database was invented by Ted Codd in 1969, based upon the mathematical idea of a relation as a subset of a product of sets. Codd realized that a significant percentage of all data processing programming was just implementing, again and again, the same set-based operations, which he called the Relational Algebra. Databases should consist of one or more Tables, which are N-ary Relations: $T \subseteq A_1 \times A_2 \times \ldots \times A_n = \Pi A_i$. For instance, an airport departure listing is a relation among the sets: {Airlines}, {Flight#s}, {Cities}, {Gates}, {Times}, {Statuses }.

American	103	Boston	A12	02:44PM	OT
Delta	1132	Atlanta	A04	02:50PM	OT
SW	134	Chicago	B09	02:55PM	Late

Viewed as a Table, a Relation is composed of rows and columns, where each row is an element of the relation and each column corresponds to one of the sets which comprise the product set. If tables are thought of as files, then rows are records and columns are fields; but this view of data is archaic and limiting.

A column entry may be blank with two meanings:

- **Spaces (usually: " ") - column value is known to be blank;**
- **Null (usually: "") - column value is unknown.**

This introduces a 3-value logic, (True, False, Unknown) where both "Null = Null" and "Null ≠ Null" are not true.

Codd's operations, the relational algebra, are:

- **Projection - only certain columns: $\Pi_{ijk}(T) \subseteq A_i \times A_j \times A_k$;**
- **Selection - only certain rows: Selection(T) $\subseteq$ T ;**
- **Product - Cartesian product (& "compositional join") ;**
- **Quotient - summarize a table by an equivalence relation.**

Codd's relational algebra seemed awkward and his proposal did not immediately attract a large following. But some people at IBM (System R) did appreciate their significance and invented a language, SQL or Structured Query Language.

The basic SQL commands required to maintain a database are:

CREATE , DROP or MODIFY (e.g. add new column) a table (these alter the intension or definition of the table);

INSERT or DELETE a row, or UPDATE a row-column entry (these alter the extension or content of a table).

The SQL SELECT command accesses the data in the tables in ways which parallel Codd's relation algebra. The syntax of the SELECT command is the most important feature of SQL. These commands and their syntax are demonstrated in the following using a simplified database for a college.

```
CREATE TABLE students (
        student_no          integer,
        lastname            char(15),
        firstname           char(10),
        email               char(50),
        status              char(1),    /* F, P, I */
        phone               char(15)    );
UNIQUE KEY ON students (student_no);
```

1456723	Smith	Darrel	dsmith3@	F	410-798-1234
1456727	Jones	Joan	jjones6@	F	443-334-2132
1456735	Brown	Marvin	mbrown@	F	410-345-2345

```
CREATE TABLE courses (
        dept_cd             char(3),
        course_no           char(3),
        course_name         char(50),
        credit_hours        integer,
        description         text        );
UNIQUE KEY ON courses (dept_cd,course_no);
```

MAT	250	Discrete Structures	3	Math for CompSci …
CSI	135	Intro to Unix	3	Best OS …
MAT	202	Linear Algebra	4	Vectors and Matrices…

CREATE TABLE sections (

dept_cd	**char(3),**
course_no	**char(3),**
section_no	**char(3),**
semester	**char(8),**
room	**char(8),**
days	**char(5),**
time	**char(11));**

UNIQUE KEY ON sections (dept_cd,course_no,section_no, semester);

MAT	**250**	**400**	**Spr2015**	**Math204**	**MW**	**7:15**
MAT	**212**	**001**	**Spr2015**	**Math102**	**TTh**	**10:40**
MAT	**212**	**401**	**Spr2015**	**Math106**	**MW**	**5:10**

CREATE TABLE enrollments (

dept_cd	**char(3),**
course_no	**char(3),**
section_no	**char(3),**
semester	**char(8),**
student_no	**integer,**
grade	**char(1));**

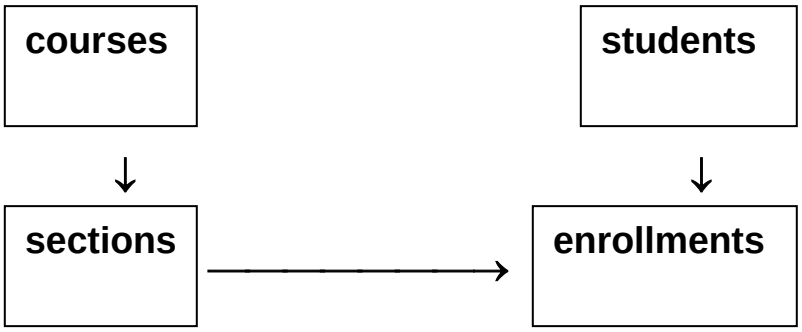
UNIQUE KEY ON enrollments (dept_cd,course_no,section_no,semester,student_no);

MAT	**250**	**400**	**Spr2015**	**1456723**	
MAT	**250**	**400**	**Spr2015**	**1456729**	
MAT	**250**	**400**	**Spr2015**	**1456735**	**W**

Notice there are no arrays, even though it might seem more natural to add an array structure (student_no, grade) to the sections table and allow it to repeat up to 100 times. Notice that each table has a primary key, which uniquely identifies each row and, in a sense, defines what the table "is about". The primary key may be a single column or multiple columns (composite key) and is usually implemented/enforced by a unique index.

Notice that the sections table primary key contains the courses table primary key. This provides a "link" or "1-to-n relationship" between these tables, in a sense defining each section as the

"child" of a specific "parent" course. Similarly, the enrollments table primary key contains the sections table primary key and the students table primary key. This logical structure is illustrated below.



The arrows indicate that each row in the "target" or "child" table must have a matching row in the "source" or "parent" table. Any child row must have a parent row but not vice versa.

Notice that "course_name", "credits_hours" and "description" are in the courses table and not the sections table. This is because they have the same value for all sections of a given course and it would be redundant to carry them in the sections table. Similarly, the enrollments table does not contain the student "lastname" or "firstname" nor the section "room" or "days". This prevents update anomalies.

A column is only placed in that table upon whose full primary key that column's value is fully, functionally dependent. This principle is called normalization: "a field is a fact about the key, the whole key, and nothing but the key, so help me Codd".

To produce a list of all full-time students:

```
SELECT  student_no, lastname, firstname
FROM students
WHERE status = "F";
```

This illustrates a: projection - "SELECT ... FROM" command,
selection - "WHERE" clause.

Some find it confusing that Codd's "projection" is implemented via the "SELECT" command.

To produce a sorted list of all courses and their locations for the current semester (2006Fall):

```
SELECT  c.dept_cd, c.course_no, course_name,
        section_no, room, days, start_time
FROM courses c, sections o
WHERE   c.dept_cd       = o.dept_cd
  AND   c.course_no     = o.course_no
  AND   semester        = "2006Fall"
ORDER BY c.dept_cd, c.course_no, section_no;
```

This illustrates a selection, projection and join - the columns "dept_cd" and "course_no" link the two tables.

Notice that SQL specifies only "what" is to be done, not "how". How to actually navigate through the tables, how to match rows from one table to those of another, is accomplished by the database software. It need not concern the programmer.

Notice that column names which appear in more than one table must be "prefixed" to identify them. The prefix is defined by placing it after the table name in the "FROM" clause.

Both the courses and sections table are listed in the "FROM" clause and linked together in the "WHERE" clause, so that sections are matched only to their appropriate "parent" courses. Without this "linking", the join would match every course to every section, producing the "full Cartesian product" of the two tables (relations). The "link" in the "WHERE" clause serves as a virtual "selection" from the huge, "full Cartesian product" table.

To produce a list of all students with "Incompletes":

```
SELECT  c.dept_cd, c.course_no, o.section_no,
        course_name,
        lastname, firstname, s.student_no
 FROM courses c, sections o, enrollments e, students s
```

```
WHERE   c.dept_cd       = o.dept_cd
  AND   c.course_no     = o.course_no
  AND   o.dept_cd       = e.dept_cd
  AND   o.course_no     = e.course_no
  AND   o.section_no    = e.section_no
  AND   o.semester      = e.semester
  AND   e.student_no    = s.student_no
  AND   o.semester      = "2006Fall"
  AND   grade           = "I";
```

This illustrates multiple joins with selection and projection. Notice that data from all four tables is required to produce this list. Imagine having to write a program to manually read and match records from four different files.

To produce a sorted list of how many students are in each course for the current semester:

```
SELECT   c.dept_cd, c.course_no,
         MIN(course_name),  count(*)
FROM courses c, sections o, enrollments e
WHERE    c.dept_cd       = o.dept_cd
  AND    c.course_no     = o.course_no
  AND    o.dept_cd       = e.dept_cd
  AND    o.course_no     = e.course_no
  AND    o.section_no    = e.section_no
  AND    o.semester      = e.semester
  AND    semester        = "2006Fall"
GROUP BY c.dept_cd, c.course_no
ORDER BY c.dept_cd, c.course_no;
```

This illustrates the "GROUP BY" clause, which is how SQL implements Codd's quotient operation to summarize data. Notice there is no need to join the Students table. The "GROUP BY" clause operators include:

```
COUNT(*),
MIN(column),
MAX(column),
TOTAL(column),
```

AVERAGE(column)

and other "summarizing" operators. Notice that in the above, "course_name" has the "MIN" operator applied since it is not in the "GROUP BY" list of columns. Every "SELECTed" column not "GROUPed BY" must have an operator. The same result would have been produced had we included "course-name" without the "MIN" and added it to the "GROUP BY" list.

The summarizing operation was called a "quotient" by Codd because it corresponds to the quotient operation of an equivalence relation. Two rows are equivalent if they share the same "GROUP BY" column values. The quotient operation defines "equivalence classes" and the operators produce facts about these "equivalence classes". The set of equivalence classes is called the quotient set.

For completeness, the syntax of the basic commands to altar the extension of a table are shown below.

A table might be populated with the insert command:

```
INSERT INTO courses values
("CSI","130","Microcomputer Operating ystems",3,,)
("CSI","135","Introduction to Unix",3,,)
("CSI","161","Programming in Java",4,)
("CSI","172","Relational Databases and SQL",3,)
("CSI","250","Data Structures in C++",3,)
("MAT","135","Statistics",3,)
("MAT","151","Pre-Calculus",4,)
("MAT","191","Calculus I",4,)
("MAT","192","Calculus II",4,)
("MAT","212","Differential Equations",4,)
("MAT","250","Discrete Structures",3,)          ;
```

To delete all inactive students:

```
DELETE FROM students
WHERE status = "I";
```

To change MAT-250 from a 3 hour to 4 hour course:

```
UPDATE courses
        SET  credit_hours = 4
WHERE dept_cd = "MAT" AND course_no = "250";
```

To change to "Inactive" the status of all students who have not taken a course since 2003:

```
UPDATE students
        SET  status = "I"
WHERE NOT EXISTS
SELECT student_no
FROM enrollments
WHERE students.student_no = enrollments.student_no
   AND semester > "2003";
```

When accessing an SQL database with a procedural language like Java or C++, one "declares a cursor" with a "SELECT ... FROM" statement. The "cursor" provides a virtual "file" of table-rows, which the program "opens" and "loops" through, just as one would when reading a simple, sequential file.

Class discussion: Is this a good design?

1. **Is the semester column a good idea?**

2. **What about faculty? Who teaches a course? Or should it be section? Add a Faculty table? What about non-faculty employees? A Persons tables?**

3. **What about addresses or phones? Multiples?**

4. **Other concerns?**

Class and Homework Exercise: The following data items are taken from the name and address book of a sales representative. Organize them into one or more "normalized" SQL tables, forming a relational database. Be sure to consider primary keys and explain why you chose the design you did.

Data organized in pre-labeled slots in the address book:

- **Name:**
  - **First**
  - **Middle**
  - **Last (sorted by this, the salesman insists, but so what?)**
  - **Title**
- **Date of Birth**
- **Address:**
  - **Street**
  - **City**
  - **State/Country**
  - **Zip/Postal Code**
- **Phone**
- **Email**

Complications with data from hand-written additions …

- **Names:**
  - **Nick Name**
  - **Maiden, Married, Pen, etc, Names?**
  - ~~**Wife**~~ **Spouse Name**
  - ~~**Secretary**~~ **Assistant Name**
  - **Business Partner Name and/or Firm Name**
  - **Children Names?**
- **Addresses**
  - **Home**
  - **Business**
  - **Vacation**
- **Phones**
  - **Home**
  - **Cell**
  - **Business**