3F6 - Software Engineering and Design

Handout 15

Database Systems II

With Markup

Ed Rosten

Contents

1. Relational Databases
2. Relational Model and SQL
3. Relational Algebra
4. Database Normalization
5. Security and SQL Injection

Copies of these notes plus additional materials relating to this course can be found at:
`http://mi.eng.cam.ac.uk/~er258/teaching`

Relational Databases

Databases are a common part of many large software systems. They store data, provide transactional safety and allow efficient searching and cross referencing of data. There are many data storage models, and one of the most popular is the relational model (Codd, 1970). This model allows for the designer to build in data safety in much the same way as data typing, in order to prevent entire classes of errors.

Many relational databases are accessed using SQL (Structured Query Language). There are many SQL databases available:

- Free: PostgreSQL, MySQL, SQLite, (BerkeleyDB), ...
- Commercial: Oracle, DB2, SQL Server, ...

SQLite is the easiest database to start using as it requires no setup, and is available on the teaching system. Type:

```
sqlite3 <db-name>
```

Then enter SQL commands followed by a '`;`'. The database will be stored in a file called `<db-name>` which will be created if it does not already exist.

Additionally, SQLite supports ACID transactions. Note that the version of SQLite on the teaching system does not support foreign key constraints. Additionally, SQLite does not support domain constraints.

The relational model

The relational model is related to set theory. A relation is a table. A relation contains a set of tuples (rows).

relation	course		
scheme	Title	Leader	Lectures
t_1	RISC Processors	Sanchez	8
t_2	QAM for modems	Sanchez	34
t_3	Introduction to Mainframes	Belford	20
t_4	Fast refresh LCDs	Richard	1
t_5	$t_5[Title]$	$t_5[Leader]$	$t_5[Lectures]$
t_6	$t_6[Title, Leader]$		

The meaning of the data is described by the scheme, which is a set of column names. Column names are known as *attributes*.

course_scheme = (Title, Leader, Lectures)

There is no ordering or grouping of attributes. The table is a relation over this scheme. A relation r over a scheme R is written as $r(R)$. Each column has a *domain*, D. So:

$D_{\text{Title}} = \text{strings}, \qquad D_{\text{Lectures}} = \mathbb{Z}^+$

So each element $t_i[j] \in D_j$ and $t_i \in D_1 \times D_2 \times \cdots \times D_n$

For example, the scheme (x, y) with $D_{\text{x}} = D_{\text{y}} = \mathbb{R}$, the domain of the tuples is the domain of all two dimensional vectors.

SQL: domain ↓ Constraint ↓

```
CREATE TABLE course (Title text, Leader text, Lectures int, CHECK(Lectures > 0))
INSERT INTO course VALUES ("RISC Processors", "Sanchez", 10)
UPDATE course SET Lectures=8 WHERE  Leader="Sanchez"
DELETE FROM course WHERE Lectures=8 AND Leader="Sanchez"
DROP TABLE course
```

SQL allows domain of tuples: $D_t \subseteq D_1 \times D_2 \times \cdots \times D_n$.

Relational algebra: Projection Π

The projection operator, Π, removes columns by listing the ones to be retained. The operator is written as:

$\Pi_{\mathrm{column1,column2,\ldots}}(\mathit{relation})$.

An example of applying projection is:

$\Pi_{\mathrm{Leader,Lectures}}(\mathrm{course}) =$

Leader	Lectures
Sanchez	8
Sanchez	34
Belford	20
Richard	1

Consider a relation, $r(R)$ where R=(x,y,z) and $x, y, z \in \mathbb{R}$. Each row represents a 3D vector. The relation $\Pi_{\mathrm{x,y}}(r)$ contains the projection of the vectors onto the x, y plane.

In SQL the SELECT statement performs all of the primitive relational algebra funcionality. The selection above is rendered as:

SELECT Leader,Lectures FROM course

The general form being:

SELECT *Col1[, Col2, [· · ·]]* FROM *table*

Note that SQL is not entirely relational and the expression:

SELECT Leader FROM course

has duplicate rows. To remove duplicates, use:

SELECT DISTINCT Leader FROM course

The there is a shorthand for the identity projection:
SELECT * FROM *table*

Relational algebra: Selection σ

The selection operator accepts a predicate, Θ and a relation. Rows matching the predicate are retained:

$\sigma_{\text{Leader="Sanchez"}}(\text{course}) =$

Title	Leader	Lectures
RISC Processors	Sanchez	8
QAM for modems	Sanchez	34

The general form of the resulting relation can be written in set builder notation

$\sigma_\Theta(r) = \{t|t \in r, \Theta(t)\}$

That is, the result consists of all tuples t such that each tuple is both in the relation r and for which the predicate applied to the tuple, i.e. $\Theta(t)$, is true.

In SQL, selection is also performed with the select statement with the predicate being specified by the WHERE clause:

SELECT * FROM course WHERE Leader="Sanchez"

Predicates can contain expressions involving any or all of the rows. SQL has more or less the same set of numeric operators as C and also AND, OR, NOT, BETWEEN:

`SELECT * FROM course WHERE Lectures BETWEEN 2 AND 10`

and IN: `WHERE Leader IN ("Belford", "Richard")`

Projection and selection can be readily composed, so in general:

$\Pi_S(\sigma_\Theta(r))$ translates to `SELECT` S `FROM` r `WHERE` Θ

Union, intersection, subtraction

In SQL, union intersection and subtraction behave much more like set theory than relational algebra. For these operations it is the order of the attributes not the names of the attributes which have significance.

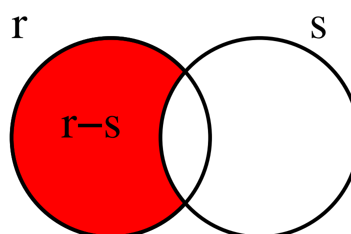
Set union, ∪ aggregates the rows of two sets together. If there are two relations, r(R) and s(R), then the union, r ∪ s can be computed:

```
SELECT * FROM r UNION SELECT * FROM s
```

Likewise, intersection can be computed using:

```
SELECT * FROM r INTERSECT SELECT * FROM s
```

Set differencing is either MINUS or EXCEPT depending on the database.




```
SELECT * FROM r EXCEPT SELECT * FROM s
```

Since ordering, not naming matters, with the schema R=(a,b), S=(b,a) and the tables r(R), s(S):

r	
a	b
1	2
3	4

s	
b	a
3	5
1	2

r - s =

a	b
3	4

Join / cartesian product $\times$

The cartesian product is the only primitive operator which combines two tables with different schemes. Joining two relations, $a \times b$ generates a new relation with every row in in a paired with every row in b. Joining is very useful for extracting related information.

students	
Student	Supervisor
Gibson	Sanchez
Murphy	Belford
Libby	Goldstein
Cook	Sanchez

labs	
Lab	Demonstrator
3F27	Cook
3F89	Libby
4F185	Margo
3F34	Ray

The table students × labs is on the next page. Note that the attributes get augmented with the table name to avoid ambiguity. The table name may be omitted if it is not ambiguous. SQL:

```
SELECT * FROM students, labs
```

Find all students of “Sanchez” who are demonstrating:

$\Pi_{\text{Student}}(\sigma_{\text{Student=Demonstrator} \wedge \text{Supervisor=``Sanchez''}}(\text{students} \times \text{labs}))$

```
SELECT Student FROM students, labs
            WHERE Student=Demonstrator AND
                  Supervisor="Sanchez"
```

The result is | Cook |. Selection is often composed with joining, so it is given the non primitive operator, the theta join:

$a \bowtie_\Theta b \equiv \sigma_\Theta(a \times b)$.

students × labs			
students.Student	students.Supervisor	labs.Lab	labs.Demonstrator
Gibson	Sanchez	3F27	Cook
Gibson	Sanchez	3F89	Libby
Gibson	Sanchez	4F185	Margo
Gibson	Sanchez	3F34	Ray
Murphy	Belford	3F27	Cook
Murphy	Belford	3F89	Libby
Murphy	Belford	4F185	Margo
Murphy	Belford	3F34	Ray
Libby	Goldstein	3F27	Cook
Libby	Goldstein	3F89	Libby
Libby	Goldstein	4F185	Margo
Libby	Goldstein	3F34	Ray
Cook	Sanchez	3F27	Cook
Cook	Sanchez	3F89	Libby
Cook	Sanchez	4F185	Margo
Cook	Sanchez	3F34	Ray

Cartesian products of real numbers

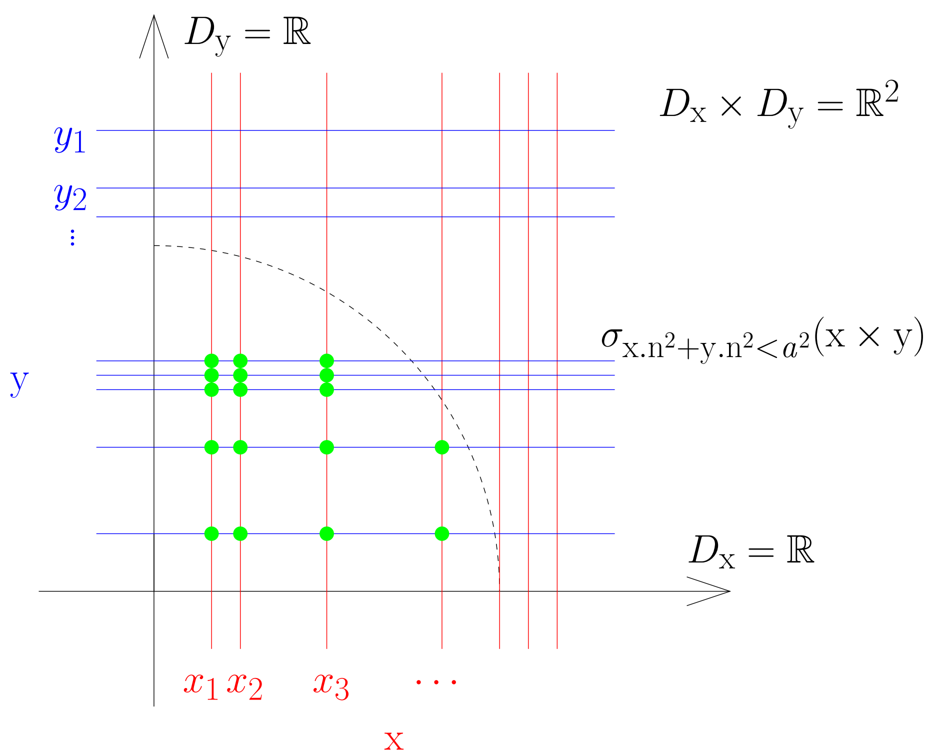
An alternative interpretation of joining is that allows the application of implicit functions to be applied to the space spanned by the data.

Consider a scheme $\mathrm{R} = (\mathrm{n}), \mathrm{n} \in \mathbb{R}$ and two relations x(R) and y(R). First consider the case where the relations contain every element of the domain, i.e. $\mathrm{x.n} = \mathrm{y.n} = \mathbb{R}$. Evaluating the expression:

$$\sigma_{\mathrm{x.n}^2+\mathrm{y.n}^2-a^2=0}(\mathrm{x} \times \mathrm{y})$$

yields all points on a circle of radius a. That is it finds all points which are the solution to the equation: $x^2 + y^2 - a^2 = 0$

Implicit equations are 'solved' by evaluating them at every point in the space. Since the data does not usually span the entire domain, the implicit function is only evaluated at the data:



Natural Join $\bowtie$

A 'natural join' is a join followed by some selection and projection:

- Perform a join.
- Perform selection so that attributes with the same name must be equal.
- Perform projection to remove duplicated attributes.

Note that there are no attribute ambiguities.

If attributes with the same name are semantically the same, then the natural join is usually the correct kind of join to use. In addition to the 'labs' table, we also have a table listing lab sessions:

sessions	
Lab	Title
3F27	Mainframe filesystems
3F27	Filesystem security
3F89	Large vehicle control
4F185	Networks for finance systems
3F34	Magnetic storage forensics

The natural join matches up the shared attributes

sessions $\bowtie$ labs =

Demonstrator	Lab	Title
Cook	3F27	Filesystem security
Cook	3F27	Mainframe filesystems
Libby	3F89	Large vehicle control
Margo	4F185	Networks for finance systems
Ray	3F34	Magnetic storage forensics

More formally:

There are two relations r(R) and s(S).

The set of shared attributes is A:

$\mathrm{A} = \{A_1, \cdots, A_n\} = \mathrm{R} \cap \mathrm{S}$

where $n = |\mathrm{A}|$. The set of all attributes with no duplicates is:

$\mathrm{R} \cup \mathrm{S}$.

The natural join is therefore:

$\mathrm{r} \bowtie \mathrm{s} \equiv \Pi_{\mathrm{R} \cup \mathrm{S}} \sigma_{\mathrm{r}.A_1=\mathrm{s}.A_1 \wedge \cdots \wedge \mathrm{r}.A_n=\mathrm{s}.A_n}(\mathrm{r} \times \mathrm{s})$

In SQL, natural joins are performed with NATURAL JOIN:

```
SELECT * FROM sessions NATURAL JOIN labs
```

In practice, you will usually design databases by considering the type of data, how it is stored in tables and how to extract the relevant information. Relation algebra will not crop up much in day-to-day design, but it is essential for understanding how the various operations in a relational database work.

Keys, and uniqueness

Rows in a relation can be uniquely identified by a key, which can consist of one or more columns. A key must be able to uniquely identify all possible rows that relation could have in the domain of tuples, not just the rows that currently exist.

Superkey: Any collection of columns which can uniquely identify a row. There may be more than one valid superkey.

Candidate key: A minimal superkey, i.e. a superkey with the minimal number of columns. I.e. there is no subset of the columns in a candidate key which will also form a candidate key. There may be more than one candidate key.

Non prime attributes: attributes of a relation which do not form part of any candidate key.

Primary key: A superkey or candidate key which has been selected to have a special status. A table can have at most one primary key.

There is an additional type which is not a key in the above sense:

Foreign key: If two relations r and s share a key k, then $r[k]$ is a foreign key if k is the primary key of s. Therefore, the foreign key k does not necessarily uniquely identify the rows of r.

Normalization

If a database has duplicated information then it is subject it update anomalies, and the information can become inconsistent. Imagine adding contact details to the 'course' table to allow lecturers to be contacted easily:

Title	Leader	Lectures	Telephone
RISC Processors	Sanchez	8	65960
QAM for modems	Sanchez	34	65960
Introduction to Mainframes	Belford	20	65536
Low latency LCD screens	Richard	1	32768

If the table is updated, for instance with the SQL command:

```
UPDATE course SET Leader="Libby" WHERE  Title="RISC Processors"
```

Then the contact details will become incorrect. The process of normalizing a database involves splitting up large tables with only weakly related information into a number of smaller tables. Normalized data is then accessed by joining tables together and performing selections on the results.

The database above is not normalized because there is duplicated data. More intuitively, the telephone number has merely been inserted as a convenience and has nothing directly to do with courses.

Much like type safety and object oriented design, database normalization allows databases to be designed such that certain errors (for instance data inconsistency) are impossible. Any error which is reduced to an impossibility can never be a bug.

Normalization is the process of movind the database comply with normal forms (1NF, 2NF, 3NF, BCNF, 4NF, 5NF and DKNF).

First Normal Form (1NF)

1. Make sure that your database really obeys the relational model:
   (a) No ordering over rows
   (b) No ordering over columns
   (c) No duplicates
2. Each row/column intersection contains exactly one datum.

Consider trying to extend the earlier design to allow for multiple phone numbers:

BAD

Title	Lectures	ID	Numbers
···	8	456	65950, 60294, 70231
···	8	456	65950, 60294, 70231
···	34	20	65536
···	1	82	32768, 16384

BAD

Title	Lectures	ID	Phone 1	Phone 2	Phone 3
···	8	456	65960	60294	70231
···	34	456	65960	60294	70231
···	20	9	65536		
···	1	82	32768		16384

Note the use of IDs to avoid duplicates as names make bad keys:

employees		
Name	ID	Phone
Sanchez	456	65960
Belford	9	65536
Richard	82	32768
Sanchez	456	60294

The list of phone numbers for the leader of a particular course can now be extracted using relational algebra: $\Pi_{\text{Phone}}(\sigma_{\text{Title}=\text{"RISC Processors"}}(\text{course} \bowtie \text{employees}))$

Second Normal Form (2NF)

A table is in second normal form if it satisfies:

1. It is in first normal form (1NF).
2. All non-prime attributes depend on the whole candidate key.

From the previous example, the complete relation, employees(E), is:

employees		
Name	ID	Phone
Sanchez	456	65960
Belford	9	65536
Richard	82	32768
Sanchez	456	60294
Sanchez	456	70231
Richard	82	16384

Lack of normalization allows buggy programs to create inconsistencies:

Inserting the record ("Belford", 10, 131072) leads to a mismatch between the name and id.

An employee name change requires updates across multiple rows, which may be done incorrectly. It also requires more locking.

The candidate key is C = (ID, Phone). The non prime attribute is therefore E − C =(Name). The employees' names do not depend on the phone number, only the ID. Therefore the table is not in 2NF. A 2NF design is:

employee_names	
Name	ID
Sanchez	456
Belford	9
Richard	82

contacts	
ID	Phone
456	65960
9	65536
82	32768
456	60294
456	70231
82	16384

Third Normal Form (3NF)

"I swear by Codd that each non-prime attribute shall depend upon the key, the whole key and nothing but the key."

More formally a table over R is in 3NF iff:

1. It is in 2NF (and therefore 1NF)
2. Every non-prime attribute is directly dependent on every candidate key of R.

Practical	Date	Demonstrator	Contact	Pay_rate
Acoustic coupling	Mon 1 Feb	Dade	45102	10
Acoustic coupling	Sat 7 Feb	Dade	45102	15
Self-propagating code	Tue 2 Mar	Joey	67822	10
Self-propagating code	Sun 9 Mar	Kate	62341	15

The candidate key is:

(Practical,Date)

Table is not fully normalized because there is repetition of data (the contact numbers and the pay rates). The table is not in 3NF because:

- Pay_rate depends on the key, but not the *whole key*. Specifically, it only depends on the date.
- Contact depends upon the whole key, but the dependence is transitive, not direct, that is:
  Contact $\rightarrow$ Demonstrator $\rightarrow$ (Practical, Date)

Updating the date requires an update of the pay rate. Updating a demonstrator requires an update of the contact number.

SQL Constraints

In addition to normal forms, which can be represented in relational algebra, SQL allows tables to be constructed with additional constraints which make the database more robust. Unlike normalization, constraints do not make it impossible to construct errors. However, constraints do make errors cause transactions to abort, rather than make inconsistent data.

NOT NULL prevents missing attributes (helpful for 1NF)

```
CREATE TABLE course (Name string NOT NULL, ...)
```

A primary key can be specified. This will ensure that ID is unique, and therefore all rows are also unique.

```
CREATE TABLE people (Name string, ID int PRIMARY KEY)
```

Known candidate keys can be marked as unique:

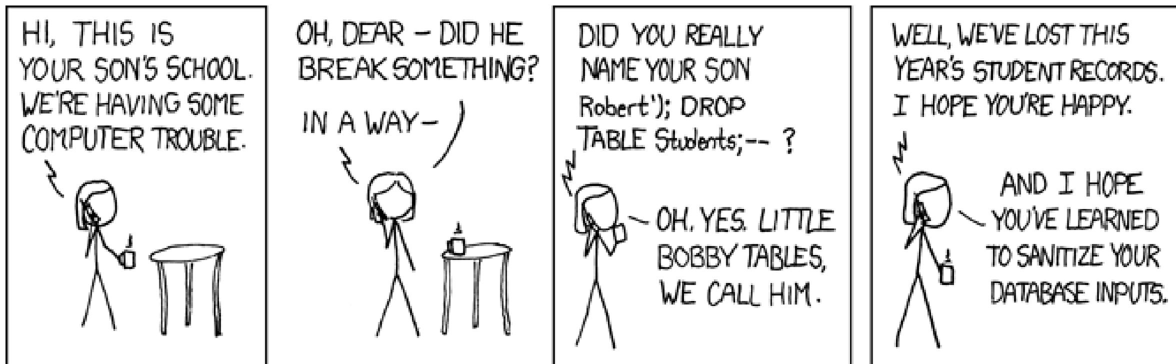
```
CREATE TABLE r (a, b, c, d, UNIQUE(a, b),
            UNIQUE(a, c, d))
```

A particularly important constraint is FOREIGN KEY which ensures that an attribute is a primary key in another table:

```
CREATE TABLE course (Title string PRIMARY KEY, ID int,
            Lectures int,
            FOREIGN KEY (ID) REFERENCES employees)
```

The ID of the course leader is now constrained to be a valid employee ID. The database will abort a transaction which attempts to add an invalid ID, or change an ID to an invalid one. Additionally the database will abort any transactions which invalidate existing ID. For example, the database will not allow erasure of employees with courses still assigned.

Security and SQL injection



Randall Munroe ©2009, `http://xkcd.com/327/`

Consider some code like this:

```
string name;
cout << "Enter product name:" << endl;
getline(cin, name);
string query = "SELECT * FROM products WHERE name=\"" + name + "\"";
do_sql(query);
```

What happens if the user enters:

```
" ; DROP TABLE products; --
```

The query becomes:

```
SELECT * FROM products WHERE name="" ; DROP TABLE products; -- "
```

The results is that the user can inject arbitrary SQL code by carefully constructing the input. This is one of the largest source of security holes.