Informatics 1: Data & Analysis

Lecture 6: SQL

Ian Stark

School of Informatics
The University of Edinburgh

Tuesday 4 February 2013
Semester 2 Week 4

Lecture Plan

Data Representation

This first course section starts by presenting two common data representation models.

- The *entity-relationship (ER)* model
- The *relational* model

Data Manipulation

This is followed by some methods for manipulating data in the relational model and using it to extract information.

- *Relational algebra*
- The *tuple-relational calculus*
- The query language *SQL*

SQL: Structured Query Language

- SQL is the standard language for interacting with relational database management systems
- Substantial parts of SQL are declarative: code states what should be done, not necessarily how to do it.
- When actually querying a large database, database systems take advantage of this to plan, rearrange, and optimize the execution of queries.
- Procedural parts of SQL do contain imperative code to make changes to the database.
- While SQL is an international standard (ISO 9075), individual implementations have notable idiosyncrasies, and code may not be entirely portable.

SQL Data Manipulation Language

In an earlier lecture we saw the SQL Data Definition Language (DDL), used to declare the schema of relations and create new tables.

This lecture introduces the Data Manipulation Language (DML) which allows us to:

- Insert, delete and update rows in existing tables;
- Query the database.

Note that "query" here covers many different scales: from extracting a single statistic or a simple list, to building large tables that combine several others, or creating *views* on existing data.

SQL is a large and complex language. Here we shall only see some of the basic and most important parts. For a much more extensive coverage of the topic, sign up for the *Database Systems* course in year 3.

Inserting Data into a Table

```
CREATE TABLE Students (
    matric VARCHAR(8),
    name  VARCHAR(20),
    age    INTEGER,
    email  VARCHAR(25),
    PRIMARY KEY (matric) )
```

The following adds a single record to this table:

```
INSERT
    INTO Students (matric, name, age, email)
    VALUES ('s0765432', 'Bob', 19, 'bob@sms.ed.ac.uk')
```

For multiple records, repeat; or consult your RDBMS manual.

Strictly, SQL allows omission of the field names; but if we include them, then the compiler will check them against the schema for us.

Update and Delete Rows in a Table

Update

This command changes the name recorded for one student:

```
UPDATE Students
    SET name = 'Bobby'
    WHERE matric = 's0765432'
```

Delete

This deletes from the table all records for students named "Bobby":

```
DELETE
    FROM Students
    WHERE name = 'Bobby'
```

Simple Query

Extract all records from the table for students older than 18:

```
SELECT *
FROM Students
WHERE age > 18
```

This will return a new table, with the same schema as Students, but containing only some rows.

Simple Query

Extract all records from the table for students older than 18:

```
SELECT *
FROM Students
WHERE age > 18
```

This will return a new table, with the same schema as Students, but containing only some rows.

Variations

We can explicitly name the selected fields.

```
SELECT matric, name, age, email
FROM Students
WHERE age > 18
```

Simple Query

Extract all records from the table for students older than 18:

```
SELECT *
FROM Students
WHERE age > 18
```

This will return a new table, with the same schema as Students, but containing only some rows.

Variations

We can identify which table the fields are from.

```
SELECT Students.matric, Students.name, Students.age, Students.email
FROM Students
WHERE Students.age > 18
```

Simple Query

Extract all records from the table for students older than 18:

```
SELECT *
FROM Students
WHERE age > 18
```

This will return a new table, with the same schema as Students, but containing only some rows.

Variations

We can locally abbreviate the table name with an *alias*.

```
SELECT S.matric, S.name, S.age, S.email
FROM Students AS S
WHERE S.age > 18
```

Simple Query

Extract all records from the table for students older than 18:

```
SELECT *
FROM Students
WHERE age > 18
```

This will return a new table, with the same schema as Students, but containing only some rows.

Variations

We can save ourselves a very small amount of typing.

```
SELECT S.matric, S.name, S.age, S.email
FROM Students S
WHERE S.age > 18
```

Anatomy of an SQL Query

SELECT *field-list*
FROM *table-list*
[**WHERE** *qualification*]

- The **SELECT** keyword starts the query.
- The list of fields specifies *projection*: what columns should be retained in the result. Using $*$ means all fields.
- The **FROM** clause lists one or more tables for cross-product.
- An optional **WHERE** clause specifies *selection*: which records to return from the tables.

Anatomy of an SQL Query

SELECT *field-list*
FROM *table-list*
[**WHERE** *qualification*]

The *table-list* in the **FROM** clause is a comma-separated list of tables to be used in the query:

...
FROM Students, Takes, Courses
...

Each table can be followed by an alias Students **AS** S, or even just Students S.

Anatomy of an SQL Query

SELECT *field-list*
FROM *table-list*
[**WHERE** *qualification*]

The *field-list* after **SELECT** is a comma-separated list of (expressions involving) names of fields from the tables in **FROM**.

SELECT name, age
...
...

Field names can be refined by using table names or aliases: Students.name, or S.age.

Anatomy of an SQL Query

SELECT *field-list*
FROM *table-list*
[**WHERE** *qualification*]

The *qualification* in the **WHERE** clause is a logical expression built from tests involving field names, constants and arithmetic expressions.

...
...
WHERE age $>$ 18 **AND** age $<$ 65

Expressions can involve a range of numeric, string and date operations.

Simple Query

Extract all records from the table for students older than 18:

```
SELECT *
FROM Students
WHERE age > 18
```

This will return a new table, with the same schema as Students, but containing only some rows.

Aside: Multisets

The relational model given in earlier lectures has tables as *sets* of rows: so the ordering doesn't matter, and there are no duplicates.

Actual SQL does allow duplicate rows, with a **SELECT DISTINCT** operation to remove duplicates on request.

Thus SQL relations are not sets but *multisets* of rows. A multiset, or *bag*, is like a set but values can appear several times. The number of repetitions of a value is its *multiplicity* in the bag.

The following are distinct multisets:

$$\{\!|2,3,5|\!\} \qquad \{\!|2,3,3,5|\!\} \qquad \{\!|2,3,3,5,5,5|\!\} \qquad \{\!|2,2,2,3,5|\!\}$$

Ordering still doesn't matter, so these are all the same multiset:

$$\{\!|2,2,3,5|\!\} \qquad \{\!|2,3,2,5|\!\} \qquad \{\!|5,2,3,2|\!\} \qquad \{\!|3,2,2,5|\!\}$$

Further Aside: Quotation Marks in SQL Syntax

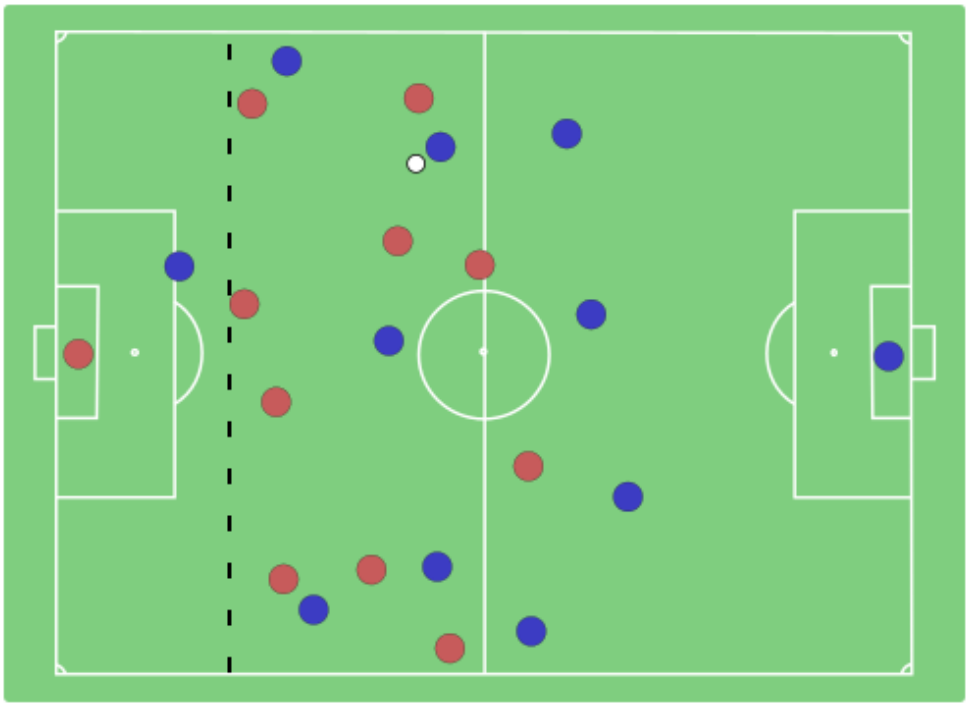
SQL uses alphanumeric tokens of three kinds:

- Keywords: **SELECT**, **FROM**, **UPDATE**, ...
- Identifiers: Students, matric, S, ...
- Strings: 'Bob', 'Informatics 1', ...

Each of these kinds of token has different rules about case sensitivity, use of quotation marks, and whether they can contain spaces.

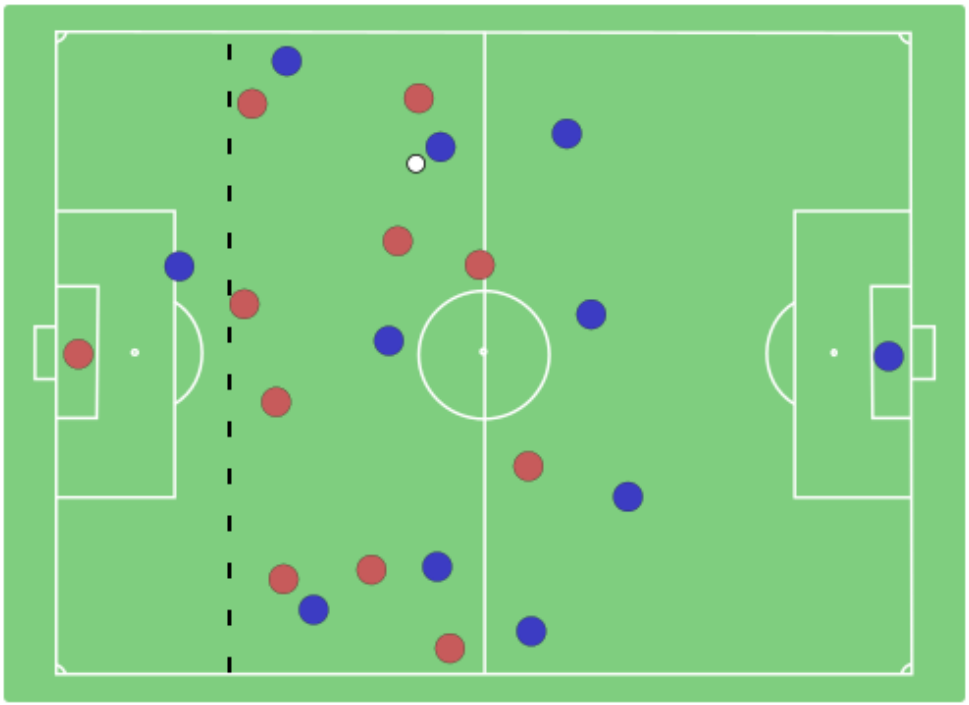
While programmers can use a variety of formats, and SQL compilers should accept them, programs that *generate* SQL code may be very formulaic in what they emit.

Anyone Recognize This?



NielsF, Wikimedia Commons

Anyone Recognize This?



NielsF, Wikimedia Commons

The blue forward on the left of the diagram is in an offside position as he is in front of both the second-to-last defender (marked by the dotted line) and the ball. Note that this does not necessarily mean he is committing an offside offence.

(New FIFA guidelines 2003; Incorporated into IFAB Law XI 2005; Clarified 2010)

Further Aside: Quotation Marks in SQL Syntax

SQL uses alphanumeric tokens of three kinds:

- Keywords: **SELECT**, **FROM**, **UPDATE**, ...
- Identifiers: Students, matric, S, ...
- Strings: 'Bob', 'Informatics 1', ...

Each of these kinds of token has different rules about case sensitivity, use of quotation marks, and whether they can contain spaces.

While programmers can use a variety of formats, and SQL compilers should accept them, programs that *generate* SQL code may be very formulaic in what they emit.

Know Your Syntax

		Case sensitive?	Spaces allowed?	Quotation character?	Quotation Required?
Keywords	**SELECT**	No	Never	None	No
Identifiers	Students	Maybe	If quoted	"Double"	If spaces
Strings	'Bob'	It depends	Yes	'Single'	Always

For example:

```
select matric
from Students as "Student Table"
where "Student Table".age > 18 and name = 'Bobby Tables'
```

It's always safe to use only uppercase keywords and put quotation marks around all identifiers. Some tools will do this automatically.

Next week: Different kinds of bracket and what they mean for you

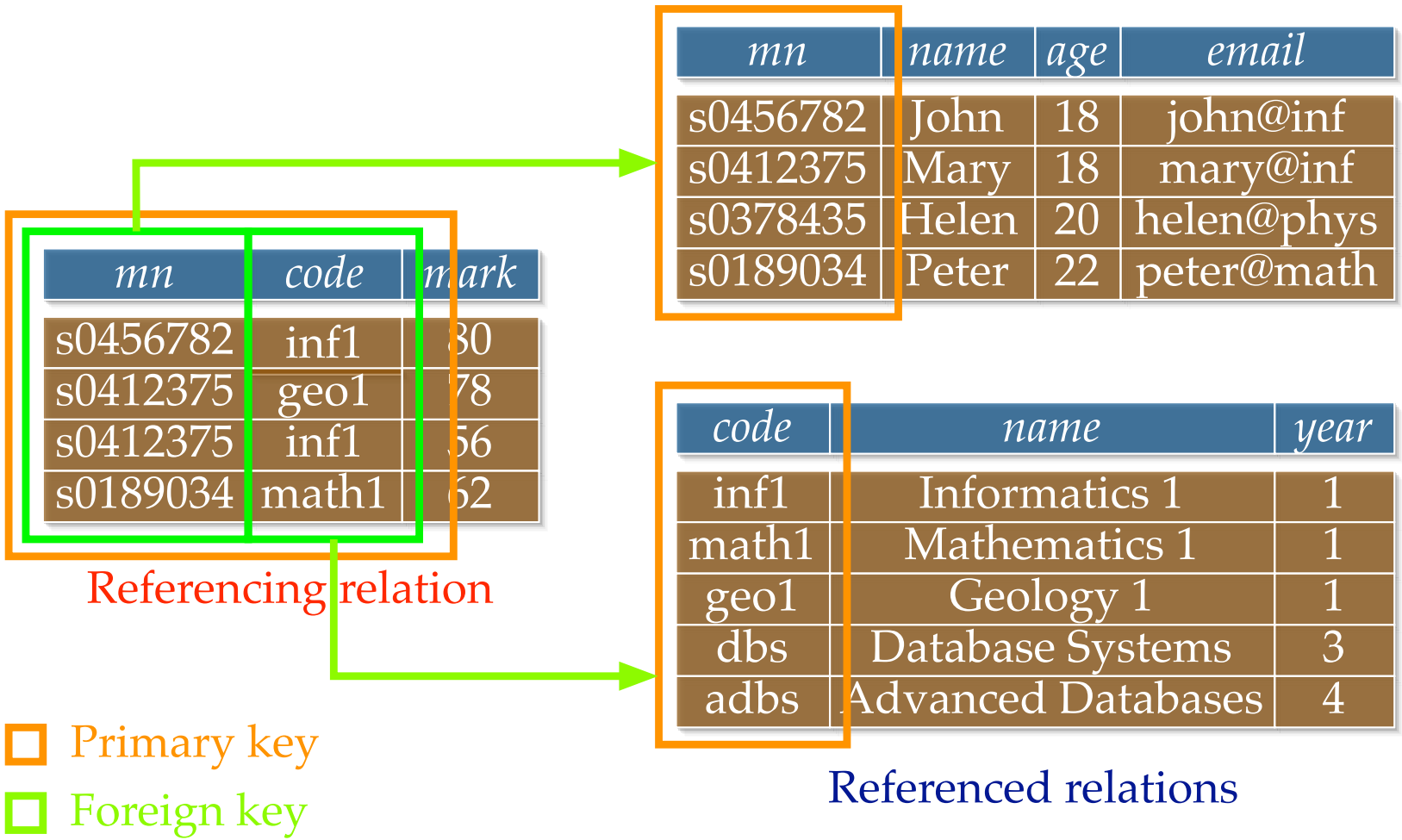
Simple Query

Extract all records from the table for students older than 18:

```
SELECT *
FROM Students
WHERE age > 18
```

This will return a new table, with the same schema as Students, but containing only some rows.

Students and Courses



Example Query

Find the names of all students who are taking Informatics 1

```
SELECT S.name
FROM Students S, Takes T, Courses C
WHERE S.matric = T.matric AND T.code = C.code
AND C.name= 'Informatics 1'
```

For the tables just given, that will return this result table.

name
John
Mary

Query Evaluation

SQL **SELECT** queries are very close to a programming-language form for the expressions of the tuple-relational calculus, describing the information desired but not dictating how it should be computed.

To do that computation, we need something more like relational algebra. A single **SELECT** statement combines the operations of join, selection and projection, which immediately suggests one strategy:

- Compute the complete cross product of all the **FROM** tables;
- Select all the rows which match the **WHERE** condition;
- Project out only the columns named on the **SELECT** line.

Query Evaluation

SQL **SELECT** queries are very close to a programming-language form for the expressions of the tuple-relational calculus, describing the information desired but not dictating how it should be computed.

To do that computation, we need something more like relational algebra. A single **SELECT** statement combines the operations of join, selection and projection, which immediately suggests one strategy:

- Compute the complete cross product of all the **FROM** tables;
- Select all the rows which match the **WHERE** condition;
- Project out only the columns named on the **SELECT** line.

Crucially, real database engines don't do that. Instead, they use relational algebra to rewrite that procedure into a range of different possible *query plans*, estimate the cost of each — looking at indexes, table sizes, selectivity, potential parallelism — and then execute one of them.