Applied Databases

Handout 1. Introduction.

22 September 2010

Lecture 1

- Course Overview
- Assessment
- Introduction to Databases
- The Relational Model
- Case studies

General information

Web page:	`http://homepages.inf.ed.ac.uk/opb/ad`
Lab page:	`http://homepages.inf.ed.ac.uk/hmueller/teaching/ad/` *provisional – will move*
Lecturer:	Peter Buneman `opb at inf dot ed dot ac dot uk` Room 5.15 Informatics Forum Office hours: Tuesdays, 1pm-2pm.
Demonstrator:	Nan Tang `ntang at inf dot ed dot ac dot uk` Room 5.38 Informatics Forum Office hours: TBA
Other support	DB admin support: `pgsql-admin@inf.ed.ac.uk`

Please consult the web page for updates, course material, etc.

Times and places

Lectures: Wednesdays, 0900–1050, Room S1, 7 George Square

Labs: Wednesdays 1200-1300 and 1300-1400 AT 4.12

Other important times (please check):

First assignment due	Friday, 22 October
Second assignment due	Friday, 12 November
Final Assignment due	Friday, 26 November

Who is this course aimed at?

- Entry level start - no prior DBMS experience is assumed.
  - Will cover basics at a fast pace
  - Research orientated
- Practical use, design and implementation of DBMSs.
- Preparation to use DBMS systems in summer projects and beyond.
- Will require some basic programming. Labs are there to help.

Course Outcomes

- Demonstrate the ability to use and apply DBMS systems.
- Understand the underlying principles.
- Compare and contrast various relational and XML based solutions.
- Appreciate the roles and limitations of DBMS in commercial and research scenarios.

Course Design

- Lectures cover essential background. Will generally last 100 minutes with an optional mid session break.
- Labs to demonstrate essential code in supervised situation
- Later labs will have no set structure and are provided as drop-in support sessions.
- Self-study and assignment designed to cover practical implementation

Assessment

- Coursework for a total of 30%:
  - Basic SQL and relational algebra (5%)
  - Database design and implementation (10%)
  - Build a "complete" system (15%): Choose a situation that requires or would benefit from using a DBMS Design and implement the DBMS Develop and test the required queries. Build the appropriate middle-ware and user interface systems.
- Exam (essays and short questions) 70%
  Please note that the exam for this course is in December, at the end of term

Plagiarism will be refereed externally

Late submissions will be penalised

Databases at Edinburgh

- e-Science centre
- Digital Curation Centre
- Strongest DB research group in the UK
- New DB courses:
  - Applied Databases
  - Advanced Databases
  - Querying and Storing XML
  - Distributed Databases
- Scottish Database Group email list (seminars)
- Lots of consumers of DB technology (esp. bio/neuro-informatics)

Let's get to work: introduction to databases

What is a Database?

- A *database* (DB) is a large, integrated collection of data.
- A DB models a real-world "enterprise" or collection of knowledge/data.
- A *database management system* (DBMS) is a software package designed to store and manage databases.

Why study databases?

- Everybody needs them, i.e. $$$ (or even £££).
- They are connected to most other areas of computer science:
  - programming languages and software engineering (obviously)
  - algorithms (obviously)
  - logic, discrete math, and theory of comp. (essential for data organization and query languages).
  - "Systems" issues: concurrency, operating systems, file organization and networks.
- There are lots of interesting problems, both in database research and in implementation. Good design is always a challenge.

Why not "program" databases when we need them?

For simple and small databases this is often the best solution. Flat files and grep get us a long way.

We run into problems when

- The structure is complicated (more than a simple table)
- The database gets large
- Many people want to use it simultaneously

Example: A personal calendar

Of course, such things are easy to find, but let's consider designing the "database" component from scratch. We might start by building a file with the following structure:

What	When	Who	Where
Lunch	24/10 1pm	Fred	Curry House
CS123	25/10 9am	Dr. Egghead	Room 234
Biking	26/10 9am	Jane	Start at Jane's
Dinner	26/10 6pm	Jane	Cafe le Boeuf
...	...	...	...

This text file is an easy structure to deal with (though it would be nice to have some software for parsing dates etc.) So there's no need for a DBMS.

Problem 1. Data Organization

So far so good. But what about the "who" field? We don't just want a person's name, we want also to keep e-mail addresses, telephone numbers etc. Should we expand the file?

What	When	Who	Who-email	Who-tel	Where
Lunch	24/10 1pm	Fred	fred@abc.com	1234	Curry House
CS123	25/10 9am	Egghead	eggy@boonies.edu	7862	Room 234
Biking	26/10 9am	Jane	janew@xyz.org	4532	Start at Jane's
Dinner	26/10 6pm	Jane	janew@xyz.org	4532	Cafe le Boeuf
...	...	...	...	...	...

But this is unsatisfactory. It appears to be keeping our address book in our calendar and doing so *redundantly*.

So maybe we want to link our calendar to our address book. But how?

Problem 2. Efficiency

Probably a personal address book would never contain more than a few hundred entries, but there are things we'd like to do quickly and efficiently – even with our simple file. Examples:

- "Give me all appointments on 10/28"
- "When am I next meeting Jane?"

We would like to "program" these as quickly as possible.

We would like these programs to be executed efficiently. What would happen if you were maintaining a "corporate" calendar with hundreds of thousands of entries?

Problem 3. Concurrency and Reliability

Suppose other people are allowed access to your calendar and are allowed to modify it? How do we stop two people changing the file at the same time and leaving it in a physical (or logical) mess?

Suppose the system crashes while someone is changing the calendar. How do we recover our work?

Example: You schedule a lunch with a friend, and your secretary *simultaneously* schedules lunch with your chairman?

You both see that the time is open, but only one will show up in the calendar. Worse, a "mixture" or corrupted version of the two appointments may appear.

Concurrency continued

Suppose you deposit a cheque for £100 by mail and sometime later withdraw £50 from a cash machine.

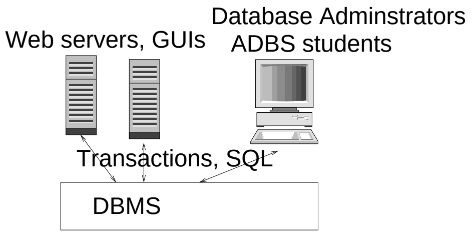
It might happen that two processes, *deposit* and *withdraw*, are simultaneously called:

1. *withdraw* reads your balance into memory location M1.
2. *deposit* reads your balance into memory location M2
3. *withdraw* subtracts £50 from M1.
4. *deposit* adds £100 to M2.
5. *deposit* writes out M2 to your balance.
6. *withdraw* writes out M1 to your balance.

Would you be happy?

Transactions

- Key concept for concurrency is that of a *transaction* – a sequence of database actions (read or write) that is considered as one indivisible action.
- Key concept for recoverability is that of a *log* – a record of the sequence of actions that changed the database.
- DBMSs are usually constructed with a client/server architecture.


Database Adminstrators
ADBS students
Web servers, GUIs
Transactions, SQL
DBMS

Database architecture – the traditional view

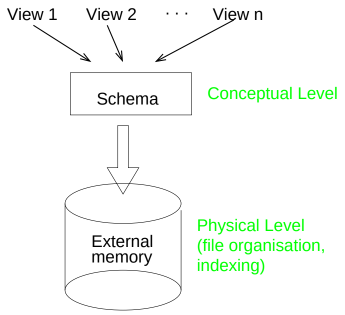
It is common to describe databases in two ways:

- **The logical structure**. What users see. The program or query language interface.
- **The physical structure**. How files are organized. What indexing mechanisms are used.

Further it is traditional to split the "logical" level into two components. The overall database design and the *views* that various users get to see.

This led to the term "three-level architecture"

Three-Level Architecture


View 1
View 2
· · ·
View n
Schema
Conceptual Level
External memory
Physical Level (file organisation, indexing)

The Relational Model

This 30-year old model is by far the most popular, but not the first, “logical” approach to databases.

In this lecture we are going to discuss relational query languages.

We’ll discuss *SQL*, the widely used language for querying, updating and creating relational databases.

We’ll also discuss a “implementation language”: *relational algebra* into which SQL is translated. We need this to understand how optimisation works.

What is a relational database?

As you probably guessed, it is a collection of *relations* or *tables*.

Munros:

MId	MName	Lat	Long	Height	Rating
1	The Saddle	57.167	5.384	1010	4
2	Ladhar Bheinn	57.067	5.750	1020	4
3	Schiehallion	56.667	4.098	1083	2.5
4	Ben Nevis	56.780	5.002	1343	1.5

Hikers:

HId	HName	Skill	Age
123	Edmund	EXP	80
214	Arnold	BEG	25
313	Bridget	EXP	33
212	James	MED	27

Climbs:

HId	MId	Date	Time
123	1	10/10/88	5
123	3	11/08/87	2.5
313	1	12/08/89	4
214	2	08/07/92	7
313	2	06/07/94	5

Munros

- Sir Hugh Thomas Munro (1856—1919)
- Scottish mountaineer
- List of mountains in Scotland over 3,000 feet (914.4 m), known as the Munros.
- 283 Munros in total (in 2009)

Why is the database like this?

Each peak has an an id, a height, a latitude, a longitude, and a rating (how difficult it is.)

Each hiker has an id, a name, a skill level and an age.

A climb records who climbed what peak on what date and how long it took (time).

We will deal with how we arrive at such a design later. Right now observe that the data values in these tables are all “simple”. None of them is a complex structure – like a tuple or another table.

Some Terminology

The column names of a relation/table are often called *attributes* or *fields*

The rows of a table are called *tuples*

Each attribute has values taken from a *domain*.

For example, the domain of `HName` is `string` and that for `Rating` is `real`

Describing Tables

Tables are described by a *schema* which can be expressed in various ways, but to a DBMS is usually expressed in a *data definition language* – something like a type system of a programming language. We'll discuss data definition languages along with database design in lecture 2

```
Munros(MId:int, MName:string, Lat:real, Long:real, Height:int,
       Rating:real)
```

`Hikers(HId:int, HName:string, Skill:string, Age:int)`

`Climbs(HId:int, MId:int, Date:date, Time:int)`

Given a relation schema, we often refer to a table that conforms to that schema as an *instance* of that schema.

Similarly, a set of relation schemas describes a database, and a set of conforming instances is an *instance* of the database.

A Note on Domains

Relational DBMSs have fixed set of "built-in" domains, such as `int`, `string` etc. that are familiar in programming languages.

The built-in domains often include other useful domains like `date` but probably not, for example, `degrees:minutes:seconds` which in this case would have been be useful. (The minutes and seconds were converted to fractions of a degree)

One of the advantages of object-oriented and object-relational systems is that new domains can be added, sometimes by the programmer/user, and sometimes they are "sold" by the vendor.

Database people, when they are discussing design, often get sloppy and forget domains. They write, for example,
`Munros(MId, MName, Lat, Long, Height, Rating)`

Keys

A *key* is a set of attributes that uniquely identify a tuple in a table. `HId` is a key for `Hikers`; `MId` is a key for `Munros`.

Keys are indicated by underlining the attribute(s):
`Hikers(HId, Hname, Skill, Age)`

What is the key for `Climbs`?

A key is a *constraint* on the instances of a schema: given values of the key attributes, there can be at most one tuple with those attributes.

In the "pure" relational model an instance is a *set* of tuples. SQL databases allow multisets, and the definition of a key needs to be changed.

We'll discuss keys in more detail when we do database design.

SQL

Reading: R&G Chapter 5

Claimed to be the most widely used programming language, SQL can be divided into three parts:

- A *Data Manipulation Language* (DML) that enables us to query and update the database.
- A *Data Definition Language* (DDL) that defines the structure of the database and imposes constraints on it.
- Other stuff. Features for triggers, security, transactions . . .

SQL has been standardised (SQL-92, SQL:99)

Things to remember about SQL

- Although it has been standardised, few DBMSs support the full standard (SQL-92), and most DBMSs support some “un-standardised” features, e.g. asserting indexes, a programming language extension of the DML.
- SQL is *large*. When last I looked, the SQL-92 standard amounted to 1400 pages. Two reasons:
  - There is a lot of “other stuff”.
  - SQL has evolved in an unprincipled fashion from a simple core language.
- Most SQL is generated by other programs — not by people.

Basic Query

```
SELECT   [DISTINCT] target-list
FROM     relation-list
WHERE    condition
```

- *relation-list*: A list of table names. A table name may be followed by a “range variable” (an alias)
- *target-list*: A list of attributes of the tables in *relation-list*: or expressions built on these.
- *condition*: Usually equality or comparisons. Some more elaborate predicates (e.g. string matching using regular expressions) are available.
- `DISTINCT`: This optional keyword indicates that duplicates should be eliminated from the result. Default is that duplicates are *not* eliminated.

Conceptual Evaluation Strategy

- Compute the product of relation-list
- Discard tuples that fail qualification
- Project over attributes in target-list
- If DISTINCT then eliminate duplicates

This is probably a very bad way of executing the query, and a good query optimizer will use all sorts of tricks to find efficient strategies to compute the same answer.

Select-Project Queries

```
SELECT   *
FROM     Munros
WHERE    Lat > 57;
```

gives

MId	MName	Lat	Long	Height	Rating
1	The Saddle	57.167	5.384	1010	4
2	Ladhar Bheinn	57.067	5.750	1020	4

```
SELECT   Rating, Height
FROM     Munros;
```

gives

Rating	Height
4	1010
4	1020
2.5	1083
1.5	1343

Product

```
SELECT   *
FROM     Hikers, Climbs
```

gives

HId	HName	Skill	Age	HId	MId	Date	Time
123	Edmund	EXP	80	123	1	10/10/88	5
214	Arnold	BEG	25	123	1	10/10/88	5
313	Bridget	EXP	33	123	1	10/10/88	5
212	James	MED	27	123	1	10/10/88	5
123	Edmund	EXP	80	123	3	11/08/87	2.5
214	Arnold	BEG	25	123	3	11/08/87	2.5
...	...	...	...	...	...	...	...

Note that column names get duplicated. (One tries not to let this happen.)

Product with selection (join)

```
SELECT   HName, MId
FROM     Hikers, Climbs
WHERE    Hikers.HId = Climbs.HId
AND      Climbs.Time >= 5
```

gives

HName	MId
Edmund	1
Arnold	2
Bridget	2

Note that `HName` and `Mid` are abbreviations for `Hikers.HName` and `Climbs.Mid`. They are unambiguous.

Aliases

The previous query could have been written:

```
SELECT   H.HName, C.MId
FROM     Hikers H, Climbs C
WHERE    H.HId = C.HId
AND      C.Time >= 5
```

Note the use of aliases (a.k.a. local variables) `H` and `C`. The are here only for convenience (they make the query a bit shorter.)

When we want to join a table to itself, they are essential.

Duplicate Elimination

```
SELECT   Rating
FROM     Routes;
```

gives

Rating
4
4
2.5
1.5

```
SELECT   DISTINCT Rating
FROM     Routes;
```

gives

Rating
4
2.5
1.5

String Matching

LIKE is a predicate that can be used in where clause. _ is a wild card – it denotes any character. % stands for 0 or more characters.

```
SELECT   *
FROM     Munros
WHERE    MName LIKE 'S_%on'
```

gives

MId	MName	Lat	Long	Height	Rating
3	Schiehallion	56.667	4.098	1083	2.5

Arithmetic

Arithmetic can be used in the SELECT part of the query as well as in the WHERE part.

Columns can be relabelled using AS.

```
SELECT   MName, Height * 3.28 AS HeightInFeet
FROM     Munros
WHERE    Lat + Long > 61;
```

gives

MName	HeightInFeet
The Saddle	3313
Ladhar Bheinn	3346

Question: How would you compute $2 + 2$ in SQL?

Set Operations – Union

```
SELECT   HId
FROM     Hikers
WHERE    Skill = 'EXP'
UNION
SELECT   HId
FROM     Climbs
WHERE    MId = 1;
```

gives

HId
123
313

The default is to *eliminate* duplicates from the union.

To preserve duplicates, use UNION ALL

What Does “Union Compatible” Mean?

```
SELECT    HId
FROM      Hikers
UNION
SELECT    MId
FROM      Climbs;
```

gives

```
HId
---
123
214
313
212
1
3
2
```

```
SELECT    HName
FROM      Hikers
UNION
SELECT    MId
FROM      Munros;
```

gives

Error!!!

- “Union-compatible” means the types as determined by the *order* of the columns must agree
- The column names are taken from the first operand.

Intersection and difference

The operator names are `INTERSECT` for $\cap$, and `MINUS` (sometimes `EXCEPT` or `DIFFERENCE`) for $-$ (set difference).

These are set operations (they eliminate duplicates).

Should `MINUS ALL` and `INTERSECT ALL` exist? If so, what should they mean?

Using bag semantics (not eliminating duplicates) for `SELECT ...FROM ...WHERE ...` is presumably done partly for efficiency.

For `MINUS` and `INTERSECT` it usually doesn’t cost any more to eliminate duplicates (why?) so one might as well do it.

`UNION` is it treated like `MINUS` and `INTERSECT`.

Nested Queries

The predicate x `IN` S tests for set membership. Consider:

```
SELECT HId
FROM   Climbs
WHERE  HId IN (SELECT HId
               FROM   Hikers
               WHERE  Age < 40) ;
```

and

```
SELECT      HId
FROM        Climbs
INTERSECT
SELECT      HId
FROM        Hikers
WHERE       Age < 40
```

Do these give the same result?

A “difference” can be written as:

```
SELECT HId
FROM   Climbs
WHERE  HId NOT IN (SELECT HId
                   FROM   Hikers
                   WHERE  Age < 40) ;
```

Correlated Nested Queries

“Correlated” means using a variable in an inner scope.

```
SELECT HId FROM Hikers h
WHERE  EXISTS (SELECT * FROM Climbs c
       WHERE   h.HId=c.HId AND c.MId = 1);
```

```
SELECT CId FROM Hikers h
WHERE  NOT EXISTS (SELECT * FROM Climbs c
                   WHERE  h.CId=c.CId);
```

```
SELECT CId FROM Hikers h
WHERE  EXISTS UNIQUE (SELECT * FROM Climbs c
                      WHERE  h.CId=c.CId);
```

`EXISTS` = non-empty, `NOT EXISTS` = empty, `EXISTS UNIQUE` = singleton set.

Comparisons with sets

x *op* ANY S means x *op* s for some $s \in S$

x *op* ALL S means x *op* s for all $s \in S$

```
SELECT HName, Age
FROM   Hikers
WHERE  Age >= ALL (SELECT Age
                   FROM   Hikers)
```

```
SELECT HName, Age
FROM   Hikers
WHERE  Age > ANY (SELECT Age
                  FROM   Hikers
                  WHERE  HName='Arnold')
```

What do these mean?

SQL is compositional – sometimes!

You can use a `SELECT ...` expression wherever you can use a table name.

Consider the query: “Find the names of hikers who have not climbed any peak.”

```
SELECT HName
FROM   ( SELECT HId
         FROM   Hikers
         MINUS
         SELECT HId
         FROM   Climbs) Temp,
       Hikers
WHERE  Temp.HId = Hikers.HId;
```

Views

[R&G 3.6]

To make complex queries understandable, we should decompse them into understandable pieces. E.g. We want to say something like:

```
NC :=    SELECT    HId
         FROM      Hikers
         MINUS
         SELECT    HId
         FROM      Climbs ;
```

and then

```
SELECT    HName
FROM      NC, Hikers
WHERE     NC.HId = Hikers.HId;
```

Instead we write

```
CREATE VIEW NC
       AS   SELECT HId
            FROM   Hikers
            MINUS
            SELECT HId
            FROM   Climbs ;
```

and then

```
SELECT HName
FROM   NC, Hikers
WHERE  NC.HId = Hikers.HId;
```

Views – cont.

The difference between a view and a value (in the programming language sense) is that we expect the database to change.

When the DB changes, the view should change. That is, we should think of a view as a *niladic function*, which gets re-evaluated each time it is used.

In fact, SQL extends views to functions:

```
CREATE VIEW ClosePeaks(MyLat, MyLong)
       AS   SELECT *
            FROM   Munros
            WHERE  MyLat-0.5 < Lat AND Lat < MyLat+0.5
            AND    MyLong-0.5 < Long AND Long < MyLong+0.5
```

Evaluating Queries on Views

```
CREATE VIEW MyPeaks
       AS   SELECT MName, Height
       FROM Munros
```

and

```
SELECT *
FROM   MyPeaks
WHERE  MName = 'Ben Nevis'
```

get rewritten to:

```
SELECT   MName, Height
FROM     Munros
WHERE    MName = 'Ben Nevis'
```

Is this always a good idea?

Sometimes it is better to *materialise* a view.

Universal Quantification

This term describes queries that ask about "all" the things in a database with certain properties. They are tricky to write.

"The names of hikers who have climbed all Munros"

```
CREATE VIEW NotClimbed            ← HId has not climbed MId
       AS   SELECT HId, MId FROM Hikers, Munros
            MINUS
            SELECT HId, MId FROM Climbs

CREATE VIEW ClimbedAll           ← HIds of climbers who have climbed all peaks
       AS   SELECT HId FROM Hikers
            MINUS
            SELECT HId FROM NotClimbed

 SELECT HName
 FROM   Hikers, ClimbedAll
 WHERE  Hikers.HId = ClimbedAll.Hid
```

Univ. Quantification – an Alternative

The `HId`s of hikers who have climbed all peaks.

```
SELECT HId
FROM   Hikers h
WHERE  NOT EXISTS
       ( SELECT RId          ← Routes not climbed by h.
         FROM   Munros m
         WHERE  NOT EXISTS
                ( SELECT *
                  FROM   Climbs c
                  WHERE  h.HId=c.HId
                  AND    c.MId=m.MId ) )
```

It's not clear whether this version is any more comprehensible!

Aggregation

These are queries that compute over columns and "aggregate" data in one or more columns. A simple example is counting:

```
SELECT    COUNT(MId)
FROM      Munros;
```

and

```
SELECT    COUNT(Rating)
FROM      Munros;
```

both give the same answer (to within attribute labels):

COUNT(Rating)
4

Why?

To fix the answer to the second, use `SELECT COUNT(DISTINCT Rating)`

GROUP BY

```
SELECT      Rating, COUNT(*)
FROM        Munros
GROUP BY    Rating;
```

gives

Rating	COUNT(*)
1.5	1
2.5	1
4	2

The effect of GROUP BY is to partition the relation according to the GROUP BY field(s). Aggregate operations can be performed on the other fields. The result is always a "flat" (1NF) relation.

GROUP BY – cont.

Note that only the columns that appear in the GROUP BY statement and "aggregated" columns can appear in the output.

```
SELECT      Rating, MName, COUNT(*)
FROM        Munros
GROUP BY    Rating;
```

gives

```
ERROR at line 1:
ORA-00979:  not a
GROUP BY expression
```

GROUP BY – cont.

```
SELECT      Rating, AVG(Height)
FROM        Munros
GROUP BY    Rating
HAVING      COUNT(*) > 1;
```

gives

Rating	AVG(Height)
4	1015

`HAVING` acts like a `WHERE` condition on the "output fields" of the `GROUP BY`. I.e., on the `GROUP BY` attributes and on any aggregate results.

In this case the output will only have tuples for the `Rating` groups with more than 1 tuple.

SQL has many more bells and whistles. E.g., one can order the output for display purposes (but this does not mean that SQL can handle ordered data.)

Null Values

The value of an attribute can be unknown (e.g., a rating has not been assigned) or inapplicable (e.g., does not have a telephone).

SQL provides a special value *null* for such situations.

The presence of null complicates many issues. E.g.:

Special operators needed to check if value is/is not null.

Is `Rating` >3 true or false when `Rating` is null? How do `AND`, `OR` and `NOT` work on null? (C.f. lazy evaluation of `AND` and `OR` in programming languages.

Operations that generate null values

An example:

```
SELECT   *
FROM     Hikers NATURAL LEFT OUTER JOIN Climbs
```

gives

HId	HName	Skill	Age	MId	Date	Time
123	Edmund	EXP	80	1	10/10/88	5
123	Edmund	EXP	80	3	11/08/87	2.5
313	Bridget	EXP	33	1	12/08/89	4
214	Arnold	BEG	25	2	08/07/92	7
313	Bridget	EXP	33	2	06/07/94	5
212	James	MED	27	⊥	⊥	⊥

Updates

There are three kinds of update: *insertions*, *deletions* and *modifications*.

Examples:

```
INSERT INTO R(a_1, ..., a_n) VALUES (v_1, ..., v_n);
DELETE FROM R WHERE ⟨condition⟩;
UPDATE R SET ⟨new-value assignments⟩ WHERE ⟨condition⟩;
```

Note: an update is typically a transaction, and an update may fail because it violates some integrity constraint.

Tuple Insertion

```
INSERT INTO Munros(MId, Mname, Lat, Long, Height, Rating)
VALUES (5, ‘Slioch’, 57.671, 5.341 981,3.5);
```

One can also insert sets. E.g., given `MyPeaks(Name, Height)`

```
INSERT INTO   MyPeaks(Name, Height)
SELECT        MName, Height
FROM          Munros
WHERE         Rating > 3
```

Note positional correspondence of attributes.

Deletion

This is governed by a condition:

```
DELETE FROM Munros WHERE MName = ‘Snowdon’
```

In general one deletes a *set*. Use a key to be sure you are deleting at most one tuple

---

Modifying Tuples

Non-key values of a relation can be changed using `UPDATE`.

Example (global warming):

```
UPDATE Munros SET Height = Height - 1 WHERE Lat < 5;
```

Old Value Semantics. Given

Emp	Manager	Salary
1	2	32,000
2	3	31,000
3	3	33,000

What is the effect of “Give a 2,000 raise to every employee earning less than their manager”?

Updating Views

This is a thorny topic. Since most applications see a view rather than a base table, we need some policy for updating views, but if the view is anything less than a “base” table, we always run into problems.

```
CREATE   VIEW   MyPeaks
         AS     SELECT   MId, MName, Height
                FROM     Munros
                WHERE    Height > 1100
```

Now suppose we `INSERT INTO MyPeaks (7, ‘Ben Thingy’, 1050)`. What is the effect on `Munros`? We can add nulls for the fields that are not in the view. But note that, if we do the insertion, the inserted tuple fails the selection criterion and does not appear in our view!!

SQL-92 allows this kind of view update. With queries involving joins, things only get worse. [R&G 3.6]

Relational Algebra

R&S 4.1, 4.2

Roughly speaking SQL is optimised by translating queries into relational algebra.

This is a set of operations (functions) each of which takes a one or more tables as input and produces a table as output.

There are six basic operations which can be combined to give us a reasonably expressive database query language.

- Projection
- Selection
- Union
- Difference
- Rename
- Join

Projection

Given a set of column names A and a table R, $\pi_A(R)$ extracts the columns in A from the table. Example, given `Munros` =

MId	MName	Lat	Long	Height	Rating
1	The Saddle	57.167	5.384	1010	4
2	Ladhar Bheinn	57.067	5.750	1020	4
3	Schiehallion	56.667	4.098	1083	2.5
4	Ben Nevis	56.780	5.002	1343	1.5

$\pi_{\texttt{MId,Rating}}(\texttt{Munros})$ is

MId	Rating
1	4
2	4
3	2.5
4	1.5

Projection – continued

Suppose the result of a projection has a repeated value, how do we treat it?

$\pi_{\texttt{Rating}}(\texttt{Munros})$ is

Rating
4
4
2.5
1.5

or

Rating
4
2.5
1.5

?

In "pure" relational algebra the answer is always a *set* (the second answer). However SQL and some other languages return a multiset for some operations from which duplicates may be eliminated by a further operation.

Selection

Selection $\sigma_C(R)$ takes a table R and extracts those rows from it that satisfy the condition C. For example,

$\sigma_{\texttt{Height > 1050}}(\texttt{Munros}) =$

MId	MName	Lat	Long	Height	Rating
3	Schiehallion	56.667	4.098	1083	2.5
4	Ben Nevis	56.780	5.002	1343	1.5

What can go into a condition?

Conditions are built up from:

- *Values*, consisting of field names (`Height`, `Age`, . . .), constants (`23`, `17.23`, `"The Saddle"`, ...)
- *Comparisons on values*. E.g., `Height > 1000`, `MName = "Ben Nevis"`.
- *Predicates* constructed from these using $\vee$ (or), $\wedge$ (and), $\neg$ (not).
  E.g. `Lat > 57` $\wedge$ `Height > 1000`.

It turns out that we don't lose any expressive power if we don't have compound predicates in the language, but they are convenient and useful in practice.

Set operations – union

If two tables have the same structure (Database terminology: are union-compatible. Programming language terminology: have the same type) we can perform set operations. Example:

`Hikers =`

HId	HName	Skill	Age
123	Edmund	EXP	80
214	Arnold	BEG	25
313	Bridget	EXP	33
212	James	MED	27

`Climbers =`

HId	HName	Skill	Age
214	Arnold	BEG	25
898	Jane	MED	39

`Hikers` $\cup$ `Climbers =`

HId	HName	Skill	Age
123	Edmund	EXP	80
214	Arnold	BEG	25
313	Bridget	EXP	33
212	James	MED	27
898	Jane	MED	39

Set operations – set difference

We can also take the *difference* of two union-compatible tables:

`Hikers` $-$ `Climbers =`

HId	HName	Skill	Age
123	Edmund	EXP	80
313	Bridget	EXP	33
212	James	MED	27

N.B. In relational algebra "union-compatible" means the tables should have the same column names with the same domains. Remember that in SQL, union compatibility is determined by the *order* of the columns. The column names in $R \cup S$ and $R - S$ are taken from the first operand, R.

Set operations – other

It turns out we can implement the other set operations using those we already have. For example, for any tables (sets) R, S

$$R \cap S = R - (R - S)$$

We have to be careful. Although it is mathematically nice to have fewer operators, this may not be an efficient way to implement intersection. Intersection is a special case of a join, which we'll shortly discuss.

Optimization – a hint of things to come

We mentioned earlier that compound predicates in selections were not "essential" to relational algebra. This is because we can translate selections with compound predicates into set operations. Example:

$$\sigma_{C \wedge D}(R) = \sigma_C(R) \cap \sigma_D(R)$$

However, which do you think is more efficient?

Also, how would you translate $R - \sigma_C(R)$?

Database Queries

Queries are formed by building up expressions with the operations of the relational algebra. Even with the operations we have defined so far we can do something useful. For example, select-project expressions are very common:

$$\pi_{\texttt{HName},\texttt{Age}}(\sigma_{\texttt{Age}>30}(\texttt{Hikers}))$$

What is this in SQL?

Also, could we interchange the order of the σ and π? Can we always do this?

As another example, how would you "delete" the hiker named `James` from the database?

Joins

Join is a generic term for a variety of operations that connect two tables that are not union compatible. The basic operation is the *product*, $R \times S$, which concatenates every tuple in R with every tuple in S. Example:

A	B
a_1	b_1
a_2	b_2

$\times$

C	D
c_1	d_1
c_2	d_2
c_3	d_3

$=$

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_1	c_2	d_2
a_1	b_1	c_3	d_3
a_2	b_2	c_1	d_1
a_2	b_2	c_2	d_2
a_2	b_2	c_3	d_3

Product – continued

What happens when we form a product of two tables with columns with the same name?

Recall the schemas: `Hikers(HId, HName, Skill, Age)` and `Climbs(HId, MId, Date,Time)`. What is the schema of `Hikers` $\times$ `Climbs`?

Various possibilities including:

- Forget the conflicting name (as in R&G) (`__, HName,Skill, Age, __, MId, Date, Time`). Allow positional references to columns.
- Label the conflicting colums with 1,2... (`HId.1, HName,Skill, Age, HId.2, MId, Date, Time`).

Neither of these is satisfactory. The product operation is no longer commutative (a property that is useful in optimization.)

Natural join

For obvious reasons of efficiency we rarely use unconstrained products in practice.

A *natural join* ($\bowtie$) produces the set of all merges of tuples that agree on their commonly named fields.

HId	MId	Date	Time
123	1	10/10/88	5
123	3	11/08/87	2.5
313	1	12/08/89	4
214	2	08/07/92	7
313	2	06/07/94	5

$\bowtie$

HId	HName	Skill	Age
123	Edmund	EXP	80
214	Arnold	BEG	25
313	Bridget	EXP	33
212	James	MED	27

$=$

HId	MId	Date	Time	HName	Skill	Age
123	1	10/10/88	5	Edmund	EXP	80
123	3	11/08/87	2.5	Edmund	EXP	80
313	1	12/08/89	4	Bridget	EXP	33
214	2	08/07/92	7	Arnold	BEG	25
313	2	06/07/94	5	Bridget	EXP	33

Natural Join – cont.

Natural join has interesting relationships with other operations. What is $R \bowtie S$ when

- $R = S$
- R and S have no column names in common
- R and S are union compatible

R&S also uses $R \bowtie_C S$ for $\sigma_C(R \bowtie_C S)$

In these notes we shall only use natural join. When we want a product (rather than a natural join) we'll use renaming . . .

Renaming

To avoid using any positional information in relational algebra, we rename columns to avoid clashes $\rho_{A \to A', B \to B', \ldots}(R)$ produces a table with column A relabelled to A', B to B', etc.

In practice we have to be aware of when we are expected to use a positional notation and when we use a labelled notation.

Labelled notation is in practice very important for *subtyping*. A query typically does not need to know the complete schema of a table.

It will be convenient to roll renaming into projection (not in R&G) $\pi_{A \to A', B \to B', \ldots}(R)$ extracts the $A, B, \ldots$ columns from R and relabels them to $A', B', \ldots$.

That is, $\pi_{A_1 \to A'_1, \ldots, A_n \to A'_n}(R) = \rho_{A_1 \to A'_1, \ldots, A_n \to A'_n}(\pi_{A_1, \ldots, A_n}(R))$

Examples

The names of people who have climbed `The Saddle`.

$$\pi_{\texttt{HName}}(\sigma_{\texttt{MName="The Saddle"}}(\texttt{Munros} \bowtie \texttt{Hikers} \bowtie \texttt{Climbs}))$$

Note the optimization to:

$$\pi_{\texttt{HName}}(\sigma_{\texttt{MName="The Saddle"}}(\texttt{Munros}) \bowtie \texttt{Hikers} \bowtie \texttt{Climbs})$$

In what order would you perform the joins?

Examples – cont

The highest Munro(s)

This is more tricky. We first find the peaks (their `MIds`) that are lower than some other peak.

$$\texttt{LowerIds} = \pi_{\texttt{MId}}(\sigma_{\texttt{Height<Height'}}(\texttt{Munros} \bowtie \pi_{\texttt{Height} \to \texttt{Height'}}(\texttt{Munros})))$$

Now we find the `MIds` of peaks that are not in this set (they must be the peaks with maximum height)

$$\texttt{MaxIds} = \pi_{\texttt{MId}}(\texttt{Munros}) - \texttt{LowerIds}$$

Finally we get the names:

$$\pi_{\texttt{MName}}(\texttt{MaxIds} \bowtie \texttt{Munros})$$

Important note: The use of intermediate tables, such as `LowerIds` and `MaxIds` improves readability and sometimes, when implemented as views, efficiency.

Examples – cont

The names of hikers who have climbed all Munros

We start by finding the set of `HId`,`MId` pairs for which the hiker has *not climbed* that peak. We do this by subtracting part of the `Climbs` table from the set of all `HId`,`MId` pairs.

$$\texttt{NotClimbed} = \pi_{\texttt{HId}}(\texttt{Hikers}) \bowtie \pi_{\texttt{MId}}(\texttt{Munros}) - \pi_{\texttt{HId},\texttt{MId}}(\texttt{Climbs})$$

The `HId`s in this table identify the hikers who have not climed *some* peak. By subtraction we get the `HId`s of hikers who have climbed all peaks:

$$\texttt{ClimbedAll} = \pi_{\texttt{HId}}(\texttt{Hikers}) - \pi_{\texttt{HId}}(\texttt{NotClimbed})$$

A join gets us the desired information:

$$\pi_{\texttt{HName}}(\texttt{Hikers} \bowtie \texttt{ClimbedAll})$$

What we cannot compute with relational algebra

There are things that we cannot compute with relational algebra.

Aggregate operations. E.g. “The number of hikers who have climbed Schiehallion” or “The average age of hikers”. These are possible in SQL which has numerous extensions to relational algebra.

Recursive queries. Given a table `Parent(Parent, Child)` compute the `Ancestor` table. This appears to call for an arbitrary number of joins. It is known that it cannot be expressed in first-order logic, hence it cannot be expressed in relational algebra.

Computing with structures that are not (1NF) relations. For example, lists, arrays, multisets (bags); or relations that are nested. These are ruled out by the relational data model, but they are important and are the province of object-oriented databases and “complex-object”/XML query languages.

Of course, we can always compute such things if we can talk to a database from a full-blown (Turing complete) programming language. We'll see how to do this later. But communicating with the database in this way may well be inefficient, and adding computational power to a query language remains an important research topic.

Review – Lecture 1

Readings: R&G Chapters 1 and 3

- Introduction. Why DBs are needed. What a DBMS does.
- 3-level architecture: separation of “logical” and “physical” layers.
- The relational model.
- Terminology: domains, attributes/column names, tables/relations, relational schema, instance, keys.
- SQL: basic forms and aggregation.
- Relational algebra: the 6 basic operations.
- Using labels vs. positions.
- Query rewriting for optimization.
- Limitations of relational algebra.