Introductory Databases

R.J.Hendley@cs.bham.ac.uk
Office: 236
Classes

Lectures:
- Tuesday 3:00 G15 Muirhead
- Friday 2:00 G15 Muirhead
Assessment

- 80% Examination
- 20% coursework
  - ‘Class test’
  - Java Exercise
<table>
<thead>
<tr>
<th>Exercise</th>
<th>Title</th>
<th>Date</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SQL Class test</td>
<td>7/11/17</td>
<td>10%</td>
</tr>
<tr>
<td>2</td>
<td>Java &amp; DB</td>
<td>28/11/17</td>
<td>10%</td>
</tr>
</tbody>
</table>

Exercise 1 is an unseen class test
Exercise 2 is assessed by viva

All work must be submitted through Canvas
Introductory Databases
Textbooks

- Databases: Ramakrishnan & Gehrke, Data Base Management Systems
- Java & DBs
  - Horstman & Cornell, Core Java, Vol 2
  - Horstman, Big Java

[Note: not all of these books!]
DataBase Systems

In the School:
- PostgreSQL:
  - See: http://supportweb.cs.bham.ac.uk/documentation/postgres
  - A ‘proper’ DBMS which is freely available

On your own machines:
- Postgres can be installed or you can use your favourite system
  - ... but be careful with non-standard features!
- Or you can use ssh or vpn to connect remotely

Coursework must be demonstrated in the lab.!!!!!!
- .... So it must work with Postgres
- [Also: note that some implementations include non-standard features – so take care!]
Course Content

An **Introduction** to *design* and *use* of Database systems

- Background, alternatives and justification of DBMS
- Relational Databases:
  - Relational model
  - Introduction to SQL
    - creating & manipulating DBs
    - Introduction to Transactions and concurrency
  - Database Design – ER diagrams and mapping to DB
  - Java & SQL – using a DB through JDBC

[Focus on *using* DBMS]
Database Management Systems - Background
How can we store (and share) data?

- Store?
- Share?
How can we store (and communicate) data?

<table>
<thead>
<tr>
<th>Store</th>
<th>Share</th>
</tr>
</thead>
<tbody>
<tr>
<td>◆ In memory:</td>
<td>◆ Shared memory</td>
</tr>
<tr>
<td>■ Persistently?</td>
<td>◆ Share files locally</td>
</tr>
<tr>
<td>◆ Java objects</td>
<td>◆ Share files remotely</td>
</tr>
<tr>
<td>◆ Flat/Structured data files:</td>
<td>◆ Document Repository</td>
</tr>
<tr>
<td>■ XML</td>
<td>◆ Data Server – fetch data as required</td>
</tr>
<tr>
<td>■ Interchange formats</td>
<td>◆ ……</td>
</tr>
<tr>
<td>◆ ……</td>
<td>◆ ……</td>
</tr>
</tbody>
</table>
All of these have their applications:

- Small, volatile datasets
- Documents, data files
- Diaries, address books ….
- Email
- Media libraries

Typically:
- Small
- Limited shared access
- Low→medium data security requirements
- Low→medium consistency
- Low→medium availability requirements
- …

Light weight solutions are fast, adequate, cheap, agile ….
... but what if we need:

- Integration with existing (DBMS-based) systems?
  - Add new functions
  - Add web clients
  - Add new functionality eg eCommerce
  - ....

- Very large amounts of data
- High availability
- High levels of data integrity
- Concurrent access from a large number of users
- High performance
Lecture 2

- What is a Database?
- What properties should they have?
- What models are there?
  - Relational Databases

- How does this fit into the bigger picture?
- A bad example
So – what is a DBMS?

<table>
<thead>
<tr>
<th>Core data base</th>
<th>Provides:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation of data</td>
<td>An abstraction from implementation</td>
</tr>
<tr>
<td>Retrieval and maintenance of data</td>
<td>Efficient implementation</td>
</tr>
<tr>
<td>Collection of tools</td>
<td>Integrity, fault tolerance, security …</td>
</tr>
<tr>
<td>Design and build and maintenance</td>
<td>Standard (and bespoke) interfaces</td>
</tr>
<tr>
<td>E.g. standard applications</td>
<td>For instance SQL for querying DB</td>
</tr>
</tbody>
</table>
DB models

There are different models available:
- Network/graph model
- Hierarchical model
- Relational model
- … and many more

Relational databases are the pre-eminent choice for general purpose data base systems:
- Especially for commercial/enterprise systems
  - Consistency, reliability etc. are critical
DB models

- We don’t always need all the benefits of a relational database
  - Consistency, integrity, reliability, fault tolerance etc.
- We may not even write to or update our database
- We may need other things:
  - Performance:
    - Fast
    - Low power consumption
  - Scaleability
    - Very large number of users
  - Flexibility
- So other technologies may be better

- Data-warehousing and data-mining
- Geographical systems
- Network infrastructure:
  - E.g. routers, nameservers
- Mobile devices/IoT
- Search engines
- Financial trading
- ……

- But:
  - These often look like proper RDBMS (on the surface)
    - SQL
Databases & Desiderata

An organised collection of data

- For a purpose
- To facilitate some set of activities
- Organised so that it can be accessed and maintained efficiently

Desirable properties

- Data independence
  - From internal or physical representation
- Minimise redundancy
  - Store data once
- Maximise consistency
  - One underlying representation
- Enable integration and sharing
- Facilitate change
- Logical organisation

[of course these apply to most things!]
ACID

Necessary properties:
- Atomicity – a transaction happens as a whole (or not at all)
- Consistency – a DB is always in a consistent state (according to the rules defined)
- Isolation – the effect of two operations happening in parallel is the same as if they had happened sequentially
- Durability – once something is stored it won’t disappear

All this, even if the plug is pulled!
ANSI/SPARC DB Architecture

- **View or external data model**
  - A view onto (a subset of) conceptual model
- **Conceptual data model**
  - An abstract model independent of implementation
- **Physical (internal) data model**
DB Applications

Users are usually shielded from the underlying DB:

- Application programs
- Web interfaces

For convenience and security ...

Actually more complex than this, but aim is to restrict access to critical components

Mechanisms are general – not just http and browsers
Actually more complex than this, but aim is to restrict access to critical components.

Mechanisms are general – not just http and browsers.
Relational Databases

- Main model for database systems
- Data in form of sets and relations
- Data connected using basic set theory
  - Selecting, combining etc.

- Normally viewed as tables
- Queries specify result not how it is computed
  - declarative
### Example – just for illustration!

#### Data Definition Language

```sql
CREATE TABLE Student (  
  sid INTEGER,  
  dob CHAR(10),  
  login CHAR(20),  
  course CHAR(10)  
)
```

#### Query Language

```sql
SELECT sid, login  
FROM Student  
WHERE course = 'Se'
```

<table>
<thead>
<tr>
<th>Student</th>
<th>sid</th>
<th>dob</th>
<th>login</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>2/1989</td>
<td>ttt</td>
<td>Cs</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>3/1989</td>
<td>rfd</td>
<td>Se</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>7/1998</td>
<td>ggg</td>
<td>Se</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>11/1998</td>
<td>hjk</td>
<td>M/Cs</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>7/1989</td>
<td>edf</td>
<td>Se</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Marks</th>
<th>sid</th>
<th>mid</th>
<th>mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>277</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>277</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>277</td>
<td>99</td>
<td></td>
</tr>
</tbody>
</table>

Tuple (or row or record)

Table name

Field or attribute names

Column (or field or attribute)

Data Definition Language

Defines Database

Table name

Field or attribute names

Tuple (or row or record)

Column (or field or attribute)

Query Language

Manipulates Data

Table name

Field or attribute names

Tuple (or row or record)

Column (or field or attribute)

Query Language

Manipulates Data
### Data Definition Language

```sql
CREATE TABLE Student (  
    sid INTEGER,  
    dob CHAR(10),  
    login CHAR(20),  
    course CHAR(10)  
)  
```

### Query Language

```sql
SELECT sid, login  
FROM Student  
WHERE course = 'Se'
```

### Table: Student

<table>
<thead>
<tr>
<th>sid</th>
<th>dob</th>
<th>login</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23/2/1989</td>
<td>ttt</td>
<td>Cs</td>
</tr>
<tr>
<td>2</td>
<td>11/3/1989</td>
<td>rfd</td>
<td>Se</td>
</tr>
<tr>
<td>3</td>
<td>22/7/1998</td>
<td>ggg</td>
<td>Se</td>
</tr>
<tr>
<td>4</td>
<td>7/11/1998</td>
<td>hjk</td>
<td>M/Cs</td>
</tr>
<tr>
<td>5</td>
<td>8/7/1989</td>
<td>edf</td>
<td>Se</td>
</tr>
</tbody>
</table>

### Table: Marks

<table>
<thead>
<tr>
<th>sid</th>
<th>mid</th>
<th>mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>277</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>277</td>
<td>65</td>
</tr>
<tr>
<td>1</td>
<td>264</td>
<td>78</td>
</tr>
<tr>
<td>2</td>
<td>266</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>277</td>
<td>99</td>
</tr>
</tbody>
</table>

What's bad about this?
### Example – just for illustration!

#### Data Definition Language

```sql
CREATE TABLE Student (  
    sid INTEGER,  
    dob CHAR(10),  
    login CHAR(20),  
    course CHAR(10)  
)
```

#### Query Language

```sql
SELECT sid, login  
FROM Student  
WHERE course = 'Se'
```

<table>
<thead>
<tr>
<th>Marks</th>
<th>sid</th>
<th>mid</th>
<th>mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>277</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>277</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>264</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>266</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>277</td>
<td>99</td>
<td></td>
</tr>
</tbody>
</table>
Example – just for illustration!

Data Definition Language
CREATE TABLE Student (
    sid INTEGER,
    dob CHAR(10),
    login CHAR(20),
    course CHAR(10)
)

Query Language
SELECT sid, login
FROM Student
WHERE course='Se'

Actually, we need a much more sophisticated model than this!
Lecture 3

Some basics
- To help you read the texts

SQL
- A start ....
A digression - Sets

Sets are collections of items
- Usually we are interested in sets where the items are of the same type
  - Numbers, dates etc.

Sets are unordered
- [Actually, here, we use ordered sets]

Sets do not contain duplicates

Written as:
- \{2, 4, 6, 8, 10\}
- \{2n \mid 1 \leq n \leq 5\}
Sets (contd.)

- The items in a set are its *elements*
  - $3 \in \{3, 5, 7, 9, 11\}$
  - $4 \notin \{3, 5, 7, 9, 11\}$

- If all the elements of a set are also elements of a second set then the first set is a *subset* of the second (and the second is a *superset* of the first).
  - $\{3, 5\} \subseteq \{3, 5, 7, 9\}$

- If the second set has extra elements then the first is a *proper subset* of the second.
  - $\{3, 5\} \subset \{3, 5, 7, 9\}$
Sets (contd)

- **Union** – the set made up of all elements of the two sets:
  - \( \{1,2\} \cup \{2,3\} = \{1,2,3\} \)

- **Intersection** – The set whose elements are in both operand sets:
  - \( \{1,2\} \cap \{2,3\} = \{2\} \)

- **Difference** – the sets whose elements are in the first but not the second:
  - \( \{1,2\} - \{2,3\} = \{1\} \)

- **The empty set**
  - \( \{\} \)
Why is this important?

- You must be clear about the logic you are trying to express in a query
- Then you can map it to an actual query
- E.g:
  - Give the phone number of everyone who lives in Birmingham who has not made a call to London:
    - ?????????
Relational Definitions
(to help follow texts)

- **Domain – D**
  - An arbitrary (non-empty) set of atomic values

- **Attribute name – A**
  - A symbol with an associated domain – \(\text{dom}(A)\)

- **Relational Schema – R**
  - A finite set of attribute names

- **Tuple – t of a Relational Schema R**
  - A mapping from the attributes A of a relational schema R to the union of their domains \(\text{dom}(A)\) s.t \(t(A) \in \text{dom}(A)\)

- **Relation – r – of a relational schema R**
  - A finite set of tuples of Relational Schema R
... and more

- **Degree (Arity) of a relation schema**
  - Number of attributes

- **Cardinality of a relation**
  - Number of tuples

That’s the formal definition – the next side should make it clear what this means!
Example - revisited

<table>
<thead>
<tr>
<th>Student</th>
<th>sid</th>
<th>dob</th>
<th>login</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23/2/1989</td>
<td>ttt</td>
<td>Cs</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11/3/1989</td>
<td>rfd</td>
<td>Se</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>22/7/1998</td>
<td>ggg</td>
<td>Se</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7/11/1998</td>
<td>hjk</td>
<td>M/Cs</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8/7/1989</td>
<td>edf</td>
<td>Se</td>
<td></td>
</tr>
</tbody>
</table>

Relation

Arity = 4
Cardinality = 5
Keys

- **Superkey** – a set of attributes that can *always* be used to differentiate one tuple from another (within a relation)
- **Key** – a minimal superkey
- **Concatenated key** – a key with more than one attribute
- **Candidate key** – any key
- **Primary key** – one of the candidate keys
- **Foreign key** – an attribute of the relation which is a key for another relation
SQL provides the mechanism that is normally used to manipulate a (relational) database. There are several components to it, but we will only look at:

- Data Definition Language – used to create, modify and delete parts of the definition of the database
- Data Manipulation Language (DML) used to manipulate the data
Conventions

- Case and whitespace are not significant
- … but there are conventions that you should follow
  - SQL keywords in upper case
  - Table names capitalised
  - Field names lower case
  - Layout
**DDL - CREATE**

- Creates a table
  - Name
  - Fields – name and type
  - Constraints on values
  - ...

```
CREATE TABLE Student (  
sid INTEGER,
dob CHAR(10),
login CHAR(20),
course CHAR(10)  
)
```
Constraints on the data can (should) be included:

- NOT NULL
- UNIQUE
  - One field or across several

It's usual to define constraints when you define the table, but:

- They can be added later
- [Sometimes this is necessary]
DDL – DROP & ALTER

- **DROP** deletes the table
  - DROP TABLE Student

- **ALTER** modifies the table definition
  - Possibly, adding nulls to the table
  - ALTER TABLE Student
    ADD COLUMN year_of_study INTEGER

Create, drop, alter can be applied to other things
- Database, constraints ...

See documentation for full details
DDL – DROP&ALTER

**DROP** deletes the table

- **DROP TABLE Student**

Create, drop, alter can be applied to other things

**ALTER** modifies the table definition

- Adding nulls to the table
  ```
  ALTER TABLE Student
  ADD COLUMN year_of_study INTEGER
  ```

See documentation for full details

Remember data bases are persistent:

- They may be business critical
  - So, we cannot rebuild them
- They may be ‘live’ for a very long time
  - So: we ‘evolve’ them ‘on the fly’

- They may be ‘live’ for a very long time
  - So: we ‘evolve’ them ‘on the fly’
Types

- **BOOLEAN** – TRUE, FALSE or NULL!
- **CHAR(size)** or **CHARACTER(size)**

**Strings**
- E.g. VARCHAR
- **INTEGER** or **INT**
  - Several variations
- **REAL, FLOAT** or **DOUBLE**
- **NUMERIC** or **DECIMAL**

- **DATE**
- **TIME** - A time of day value.

... and many more
Lecture 4
More on Constraints

Constraints place restrictions on the values (or set of values) that can be inserted into the database.

- They are **hard** constraints
  - The DBMS will check and enforce them

They are metadata used to:

- Make explicit domain constraints
  - An age must be $\geq 0$

- Maintain the integrity of the database
  - Not just locally to a table:
    - i.e. Only record marks for students who are registered
Constraints – a note

- This is similar to the checks made in Java code to maintain consistency & correctness
  - i.e. (trivially) in set methods
  - But …
    - *declarative* - so no need to write code
    - *Guaranteed* to be enforced

- But:
  - Not an excuse for laziness ….
  - Important to distinguish between:
    - Hard constraints
      - Every student has a unique sid
    - Desirable constraints
      - Every student has a login
Constraints

We have already seen some examples:
- Domain e.g. INTEGER
- NOT NULL
- UNIQUE

We can also add
- Restrictions on the range of values
- Information about keys
  - These are especially important to ensure integrity across tables
- Arbitrary checks

If constraints are violated then the operation fails
Keys

A primary key constraint

- **PRIMARY KEY** `sid`
- Relations written:
  - `Student(sid, dob, login, course)`
- Enforces:
  - **UNIQUE**
  - **NOT NULL**
- Usually a single column but can be any *key*
- Often (actually, usually) a synthetic UID

DBMS may be able to optimise

Note: signals importance to DB structure rather than *just* a data constraint
Primary Key example

CREATE TABLE Student (  
sid INTEGER,  
dob CHAR(10),  
login CHAR(20),  
course CHAR(10)  
)  
PRIMARY KEY (sid)

Or

CREATE TABLE Student (  
sid INTEGER,  
dob CHAR(10),  
login CHAR(20),  
course CHAR(10),  
CONSTRAINT StudentsKey PRIMARY KEY (sid)
)

If we name a constraint then we can manipulate it later:

• Remember, we don’t want to rebuild our database!
Keys

Foreign Key

- Defines a link to another table
  - Usually specifies the primary key in that other table (default)
- Any row inserted into this table must satisfy the constraint that:
  - The foreign key in this table must be matched by a key (usually the primary key) in the other table

CREATE TABLE Student (  
  sid INTEGER,  
  dob CHAR(10),  
  login CHAR(20),  
  course CHAR(10)  
  PRIMARY KEY (sid)  
)

CREATE TABLE Marks (  
  sid INTEGER,  
  mid INTEGER,  
  mark INTEGER,  
  FOREIGN KEY (sid) REFERENCES Student(sid),  
  FOREIGN KEY (mid) REFERENCES Course(cid)  
)
Keys

Foreign Key

- Defines a link to another table
  - Usually specifies the primary key in that other table (default)
- Any row inserted into this table must satisfy the constraint that:
  - The foreign key in this table must be matched by a key (usually the primary key) in the other table.

```
CREATE TABLE Student (
    sid  INTEGER,
    dob  CHAR(10),
    login CHAR(20),
    course CHAR(10)
    PRIMARY KEY (sid)
)

CREATE TABLE Marks (
    sid  INTEGER,
    mid  INTEGER,
    mark INTEGER,
    FOREIGN KEY (sid) REFERENCES Student(sid),
    FOREIGN KEY (mid) REFERENCES Modules(mid)
)
```

This isn’t just checked on insertion:
- It is guaranteed to always be true!

So we can have multiple rows in Marks referencing a single row in Student.
What happens if we modify the `Student` table (so that a referenced `sid` is removed?)

- ON DELETE RESTRICT
- ON DELETE NO ACTION
- ON DELETE CASCADE
- ON DELETE SET DEFAULT/NULL
Keys

CREATE TABLE Marks (  
sid INTEGER,  
cid INTEGER,  
mark INTEGER,  
FOREIGN KEY (sid) REFERENCES Student(sid)  
ON DELETE CASCADE  
ON UPDATE CASCADE,  
FOREIGN KEY (mid) REFERENCES Modules(mid)  
)

So, the integrity of the DB is maintained:

- We typically either:
  - Forbid the operation (RESTRICT or NO ACTION)
  - Delete rows that reference the deleted key (CASCADE)
- NO ACTION is the default: so, we cannot delete/change cid in Courses (IFF it is referenced!)
- Setting a default/NULL is rarely a good idea!

Of course, we probably want to be cleverer than this!
CREATE TABLE Marks (  
  sid INTEGER,  
  mid INTEGER,  
  mark INTEGER,  
  FOREIGN KEY (sid) REFERENCES Student(sid)  
  ON DELETE CASCADE  
  ON UPDATE CASCADE,  
  FOREIGN KEY (mid) REFERENCES Modules(mid)  
)

This means that:
• Changes to sid in the Students table propagate through to the Marks table
• Changes to cid in the Courses table are forbidden
  • The default is NOACTION

So, the integrity of the DB is maintained:
- We typically either:
  • Forbid the operation (RESTRICT or NO ACTION)
    • Delete rows that reference the deleted key (CASCADE)
    • NO ACTION is the default: so, we cannot delete/change cid in Courses
  • Setting a default/NULL is rarely a good idea!

Of course, we probably want to be cleverer than this!

It's a bit more complicated than we've described
• It doesn’t just apply to primary keys
• There are various syntactic forms for expressing this

What to do is a design decision!
Notes:

- Constraints are specified in the ‘child’ table and can affect operations on both ‘parent’ and ‘child’
- Operations are atomic
  - See transactions, later
- The constraints are guaranteed not to be violated
  - Operations succeed or fail
Notes:

- Not all databases fully support all of this properly:
  - E.g. some versions of MySQL
- Typically the DBMS will create an index to do the checks efficiently

- The efficiency of this varies across implementations
- One practice is:
  - For major operations (e.g. to archive graduated students)
    - Remove constraints
    - Perform operation
    - Add constraints
  - That’s why naming constraints is useful
Domain Constraints

We can also enforce additional constraints over values:

- Local constraints
- Constraints over more than one table
- These can be arbitrarily complex (to be revisited)
- See *Triggers*

```sql
CHECK (age >= 0 AND age <= 120)

CREATE TABLE Marks (
    sid INTEGER,
    mid INTEGER,
    mark INTEGER,

    FOREIGN KEY (sid) REFERENCES Student(sid)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
    FOREIGN KEY (mid) REFERENCES Modules(mid),
    CHECK (mark >= 0 AND mark <= 100)
);```
Constraints: Summary

Constraints express metadata about our domain:

- They are declarative:
  - So no need to write code
- They are guaranteed to be enforced:
  - Not just checks on insertion!

When designing:

- Capture all *hard* domain constraints:
  - ... and express them in the DB
- Do not include constraints which are desirable but not inviolable.

Note: This is really hard to do in Java.
Lecture 5 & 6

- Setting up Postgres
- SQL

  - INSERT
  - UPDATE
  - DELETE
  - SELECT
    - Selection criteria
    - Aggregate functions
    - ..... and more
Using PostgreSQL

```shell
> psql -h mod-intro-databases -d <username>

<username=> SELECT * FROM Books;
```

See module page for a cribsheet
Manipulating Data: INSERT

- INSERT
  - Inserts data into a table

  INSERT INTO Student VALUES
  (28, '28/1/09', 'reh', 'CS')

  INSERT INTO
  Student (sid, dob, login, course)
  VALUES (28, '28/1/09', 'reh', 'CS')

- In general we can specify columns, rows and values to insert
Manipulating Data: UPDATE

- **UPDATE**
  - Modify values in rows:
    - UPDATE Student
      SET course='CS'
      WHERE sid=23

- Specify table and rows and attributes and their *new* values
- All rows that meet the condition will be updated
Manipulating Data: DELETE

DELETE

- Deletes rows from the table

```
DELETE FROM Student
WHERE sid=23
```

So, we specify rows to be deleted.

All rows meeting the condition will be deleted
Data Manipulation

In general:
- We can use any SQL command within these statements
  - Arbitrary operations

All these statements are subject to constraint checking:
- If the constraints would be violated then:
  - The operation fails
  - An error is raised
Queries: SELECT

- SELECT is used to retrieve data from tables:
  - One table or (more usually) combining data from several
  - Result is a *table*:
    - that meets the *specification* in the SELECT statement

- SELECT allows the specification of:
  - Columns in the result
  - Conditions on rows
    - Which can be arbitrarily complex
  - Ordering of results
  - Aggregate functions
  - … and much more
Simple form: selecting from one table

SELECT *  
FROM    Student

SELECT sid, login  
FROM    Student  
WHERE   course='cs'

Note: can have expressions rather than just column names

SELECT Student.sid, Student.login  
FROM    Student  
WHERE   Student.course='cs'
Additionally

- DISTINCT will remove duplicate rows:
  
  ```sql
  SELECT DISTINCT course
  FROM Student
  ```

- AS attaches a label to a column

- Aggregate functions can be applied to the table: COUNT, SUM, AVG, MAX, MIN …

```sql
SELECT COUNT (DISTINCT course) AS NoC
FROM Student
```

Or we might include selection criteria
.. And more

ORDER BY allows the order of rows to be specified:
- Uses the natural order of the column
- ASC/DESC
- Can specify multiple columns

SELECT *
FROM Student
ORDER BY course, login

Remember: Tables represent ordered sets. The default order is a vestige of the retrieval algorithm.
.. And yet more

Can also specify a window onto result
- Start row
- End row

... and a few other things, which we’ll ignore
Retrieval Criteria

The selection criteria in the WHERE clause can involve arbitrarily complex expressions:

- AND/OR
- Comparison operators:
  - <, <=, =, <>

- IS [NOT] NULL
- IS [NOT] TRUE
- [NOT] IN
  - Course IN (‘Cs’, ‘Se’)

- Arithmetic
- [NOT] BETWEEN
  - Age BETWEEN 18 AND 65

Note: There are some nasties with some of this!
LIKE

LIKE provides pattern matching on strings

- _ matches any character
- % matches 0 or more characters

Name LIKE ‘_ob%’

[most implementations also allow regexps]
Other operators

There are, typically, a large number of functions and operators:
- The set (and semantics) may vary with the implementation

- Mathematical functions
- String operations
- Date manipulation
- Bitwise operations
- Formatting
- And so on
Lecture 7, 8 & 9

Joins

- INNER JOIN
- Correlation names
- Other JOINs
- Nested queries
- GROUP BY & HAVING
- Example
Joins

- Selecting data from a single table is obviously limiting
- In general we need to join data from several tables
- A join returns a table:
  - The columns are specified in the head of the SELECT clause
  - The rows are a function of the data in the joined rows

```sql
SELECT *
FROM Student, Marks
```

Or more sensibly:
```sql
SELECT * 
FROM Student, Marks
WHERE Student.sid=Marks.sid
```

Or:
```sql
SELECT Student.sid, Marks.mid, Marks.mark
FROM Student, Marks
WHERE Student.sid=Marks.sid
```
Joins

There are several different flavours of join:

- First we will look at the inner join – which is the most common
- Others follow the same pattern

```sql
SELECT Student.sid, Marks.cid, Marks.mark
FROM Student, Marks
WHERE Student.sid=Marks.sid
```

Can also be written:

```sql
SELECT Student.sid, Marks.cid, Marks.mark
FROM Student
INNER JOIN Marks
ON Student.sid=Marks.sid
```

The examples that follow will use the first style:

- It’s compact
- Referred to as ‘implicit notation’
- House styles may prefer one or the other
Join – inner join

Conceptually:

- Form cross product
  - Of the set of tables
- Select columns
  - Listed in the select
- Select rows
  - Only those that meet the WHERE condition
- [if DISTINCT then remove duplicates]
Notes

- The result of a *join* is a table
- The WHERE clause is an arbitrary expression:
  - If it evaluates to true then the row is included
- The columns are specified in the head of the SELECT:
  - Often these are just selected from the columns of the joined tables
  - ... but we can specify arbitrary expressions
Notes

If there is ambiguity in a field name (even if they are known to be equal) they must be fully qualified

One style is to always fully qualify

```
SELECT  Student.sid, Marks.cid, Marks.mark
FROM    Student, Marks
Where   Student.sid=Marks.sid
```
Correlation names

A name can be given to the table
  - Sometimes as a style
    - Briefer and clearer
  - Sometimes it is necessary
    - [Also see later example]

SELECT S.sid, M.cid, M.mark
FROM Student S, Marks M
WHERE S.sid=M.sid

SELECT S1.sid, S2.sid, S1.dob
FROM Student S1, Student S2
WHERE S1.dob=S2.dob

What does this try to do?
What is wrong with it?
Correlation names

A name can be given to the table

- Sometimes as a style
  - Briefer and clearer
- Sometimes it is necessary
  - [Also see later example]

- Every student shares a birthday with himself
- If student A shares a birthday with student B then we will also be told student B shares a birthday with student A
- How can we fix this?

```
SELECT S.sid, M.cid, M.mark
FROM Student S, Marks M
WHERE S.sid=M.sid
```

```
SELECT S1.sid, S2.sid, S1.dob
FROM Student S1, Student S2
WHERE S1.dob=S2.dob
```

What does this try to do?
What is wrong with it?
Other joins

- LEFT JOIN
  - This is just like an INNER JOIN, but
    - Includes rows in the left table that are not matched
    - NULLs are added

```
SELECT sid, marks
FROM Students
LEFT JOIN Marks
ON Student.sid = Marks.sid
```
Other joins

**RIGHT JOIN**

- This is just like an INNER JOIN, but
  - Includes rows in the right table that are not matched
  - NULLs are added

```sql
SELECT sid, marks
FROM Students
RIGHT JOIN Marks
ON Student.sid = Marks.sid
```
Other joins

- **FULL JOIN**
  - This is just like an INNER JOIN, but
    - Includes rows in the left & right tables that are not matched
    - NULLs are added

```
SELECT sid, marks
FROM Students
FULL JOIN Marks
ON Student.sid=Marks.sid
```
Other joins

- **EQUI JOIN**
  - This is just an INNER JOIN where the condition only includes equality
- **NATURAL JOIN**
  - Here the equality test is implicit based on the names of attributes
  - Don’t use it!
- **CROSS JOIN**
  - This is just the default without any selection
  - It gives every permutation of the rows in each table
Another syntactic form is:

```
SELECT Student.sid, Marks.cid, Marks.mark
FROM Student
INNER JOIN Marks
USING (sid)
```

This is equivalent to:

```
SELECT Student.sid, Marks.cid, Marks.mark
FROM Student
INNER JOIN Marks
ON Student.sid=Marks.sid
```
UNION, INTERSECT, EXCEPT

A query can be formulated as a set operation on two results:

- This can be easier and clearer than a complex condition
- It can be more efficient
- Note: the only constraint is that the tables must be compatible in number and type of attributes

```
SELECT S1.sid, S1.course
FROM Student S1
WHERE S1.course='Cs'
UNION
SELECT S2.sid, S2.course
FROM Student S2
WHERE S2.course='Se'
```

This is not obviously very useful in this example (although it might be faster) – but there are many cases where a query is much simpler, clearer and more efficient.
Nested Queries/Subqueries

- It is possible to have subqueries whose results are then used in the outer query:
  - Easier to formulate
  - Clearer
  - More efficient

- There are many operators that can be used with the sub-result:
  - [NOT] IN
  - [NOT] EXISTS
  - <op> ANY
  - <op> ALL
Example

SELECT S1.sid
FROM Student S1
WHERE S1.dob > ALL (SELECT S2.dob
FROM Student S2
WHERE S2.course='Cs')

Notes:

- We can reference the outer row within the nested query ie. reference S1 within the nested query
- Notice, that we don’t need to use different names for Student in the two queries – but if the inner query references the outer, then we must.
- Nesting can be arbitrarily deep
Exercise

Give the name of everyone who lives in Birmingham who has not made a call to London
Exercise

Give the name of everyone who lives in Birmingham who has not made a call to London

```sql
SELECT Subscribers.name 
FROM Subscribers 
WHERE 
    Subscribers.city='Birmingham'
AND 
    Subscribers.phoneno NOT IN 
(SELECT caller 
FROM Calls, Subscribers 
WHERE Calls.receiver=Subscribers.phoneno 
AND 
Subscribers.city='London');
```
GROUP BY/HAVING

- Allows operations over groups of rows
  - GROUP BY specifies grouping of rows
    - SELECT customerName, SUM(orderQuantity)
      FROM Sales
      GROUP BY customerName
  - HAVING specifies selection criteria for groups
    - SELECT customerName, SUM(orderQuantity)
      FROM Sales
      GROUP BY customerName
      HAVING SUM(orderQuantity) > 5
  - Can specify fields and aggregate operations over the groups

- So, what this does is:
  - Join tables
  - Filter rows using WHERE clause
  - Group the table using the attribute(s) specified in the GROUP BY clause
  - Returning one row per group
  - Remove rows where the HAVING clause does not return TRUE
  - Not a problem!
GROUP BY/HAVING

Notes:

- The columns must either be:
  - Specified in the GROUP BY clause
  - An aggregate applied over the group
  - [Guaranteed to be unique (because of a functional dependency)]

- This is useful:
  - In its own right
  - As a nested query

Exercise:

- Return a table which lists each student registered on course ‘CS’ with their average mark over all modules:
  - Extend to only include students with average >= 70
  - and all marks >= 40
Remember:

- We can usually use an arbitrary expression where a value is required
- This can be:
  - Useful
  - Or obscure

```sql
select b.isbn,
       (select count(borrowerid)
        from loans
        where loans.isbn = b.isbn
       ) as NoLoans
from books b;
```

This is not the right way to do this!

How should it be done?
Views

Views provide a mechanism whereby a user/programmer can be given an alternative view onto the underlying database.

For
- Convenience
- Security
Views

A *view* is specified in terms of the underlying tables

- *Derived tables vs Base tables*
- Operations are still (conceptually) performed on the base tables
  - This may have implications for some operations

The specification of a view can be arbitrarily complex
Views - Motivation

For convenience
- Rather than working with a large and complex system the data can be repackaged to provide a simpler and more appropriate view of the data
  - Filtering
  - Re-organising
  - Abstracting

Security
- Hide data that is sensitive
- Apply security policies to views

Whilst this seems like a good idea sometimes we may prefer to do this in a ‘batch’ mode.
SQL - summary

We have covered the core of SQL
- There are many more features
  - Triggers
  - Transactions – to be revisited
  - Security
  - Implementation & efficiency
  - And more ….

The features we have covered can be combined to build arbitrarily complex statements
- But as always there are techniques for managing complexity

Think!
- Get the logic right.
- Encode in SQL!
Student Records Database:
Done right!
Lecture 10, 11, 12

Java and SQL

- Basics
- Data Manipulation
- How to do it – patterns etc.
- Transactions
- Summary
JDBC

JDBC provides

- A mechanism for connection to database systems
- An API for:
  - Managing this connection
  - Sending statements to the DB
  - Receiving results from the DB

As usual, the API is:

- easy to use
- .. but general and complex if you need it

Although we focus on relational databases this is not a restriction
- And other languages provide equivalent features.
Mechanism

- Make connection
- Issue requests/process results
- Close connection

• Architecture will be more complex

Java <-> JDBC <-> DB

• Connections to multiple databases
• Connections from multiple programs
Some Warnings

Java is fairly strongly typed but …:

- The interaction with the data base is in terms of strings and resultsets:
  - It is the programmers responsibility to check they are well formed and type compatible

There are more sophisticated (and ultimately easier) ways to manage the mapping between java objects and data base relations:

- E.g Hibernate
Style

Vehicle & DBMSs have different strengths:

- Use each for the appropriate tasks:
  - Java for interaction, complex algorithms, etc.
  - DB for data manipulation

Don’t fetch all the data and then search it!
Notes

Whilst you can have a Java application which:

- Creates tables
- Populates DB
- Performs operations
- Drops tables
- [and this is a good pattern for development!]

In general, the DB will be persistent.

So:

- Java application should connect to an existing DB
- For development/debugging convenience, either:
  - Use DBMS separately to setup
  - Use separate java programs to create and destroy your DB
Notes - testing

- The usual techniques apply to testing.
- Both:
  - Java code
  - DB code

In general the point of using a DB is that there is a lot of data.
- You can use a Java program to:
  - Read data files and insert into DB
  - Generate synthetic data
- Or, you can use other languages
Using jdbc

One pattern:
- Setup JDBC
- Establish DB connection
- Repeat
  - Execute query
  - Process result
- Close DB connection

Note:
- Establishing the connection can be slow:
  - Inefficient to make/close a connection per statement
  - Connection pooling is a more elegant solution
Preparing for jdbc

**Packages**

import java.sql.*
import javax.sql.*

In practice your IDE will do most of this for you

**Drivers**

System.setProperty("jdbc.drivers", "org.postgresql.Driver")

Note: see notes on jdbc at:

http://supportweb.cs.bham.ac.uk/docs/tutorials/dosystem/build/tutorials/jdbc/jdbc.html
DB server

Define location of server:

String dbName = "jdbc:postgresql://mod-intro-databases/<name>"

Establish a connection:

Connection dbConn = DriverManager.getConnection(dbName, <User>, <Password>"

Note: There are variations on this. In particular, it is common to use a property file (especially for user and password)
String dbName = "jdbc:postgresql://dbteach/<name>

Establish a connection:
Connection dbConn = DriverManager.getConnection(dbName, <User>, <Password>

Note: There are variations on this. In particular, it is common to use a property file (especially for user and password)

try {
    Class.forName("org.postgresql.Driver");
} catch (ClassNotFoundException ex) {
    System.out.println("Driver not found");
}

System.out.println("PostgreSQL driver registered.");

Connection conn = null;
try {
    conn = DriverManager.getConnection(url, username, pass);
} catch (SQLException ex) {
    ex.printStackTrace();
}

if (conn != null) {
    System.out.println("Database accessed!");
} else {
    System.out.println("Failed to make connection");
}
Connection

- It is usual to embed statements in a try-catch-finally block
  - Where the finally closes the connection
- SQLException - later

```java
Connection dbConn ......
try
{
    <code to use DB>
}
finally
{
    dbConn.close()
}
```
Executing Statements

To execute a statement you create a
- `Statement object`

Or
- `PreparedStatement` object

PreparedStatements are preferred

These provide methods which allow you to manipulate and query the data base.

Query is specified:
- `Statement: when executed`
- `PreparedStatement: when object created`
createStatement

Statement
stmt=dbConn.createStatement()  

This creates a new Statement object

- The *SQL statement* is provided as a string parameter later (in the call)

- It is common to re-use the Statement object for successive SQL statements
  - But you obviously lose the result of the previous request

- Can have multiple Statement objects
Executing statements

- `execute(String)`
  - Executes any SQL statement
- `executeUpdate(String)`
  - For insert, update, delete etc.
- `executeQuery(String)`
  - For queries

It is more convenient to use `executeUpdate` or `executeQuery` as appropriate.

See API for details.
UPDATE, DELETE, INSERT + DDL

- Result is the number of rows affected

- Of course, we would build this string dynamically, in practice.
  - Using values read or computed

```java
Statement stmt=dbConn.createStatement();

n = stmt.executeUpdate("INSERT INTO Students VALUES (28,'28/1/09', 'reh', 'CS')");

Or (better)

n = stmt.executeUpdate("INSERT INTO Students" + " VALUES (28,'28/1/09', 'reh', 'CS')");

Or (even better)

String SQLcommand = "INSERT INTO Students" + " VALUES (28,'28/1/09', 'reh', 'CS')";

n=stmt.executeUpdate(SQLCommand);
```
executeQuery

- Takes the query as a string parameter
- Returns a `ResultSet` object

- Table
- Iterate through rows using `next()`
  - Note: usage of `next`
- Access field using position or name
  - Note: position starts at 1
  - Note: `<Type>`

```java
ResultSet rs = stmt.executeQuery("SELECT * FROM Students");
while (rs.next()) {
    sid=rs.getInt(1);
    sid=rs.getInt("sid")
}
```
executeQuery

- Takes the query as a string parameter
- Returns a `ResultSet` object

- Table
- Iterate through rows using `next()`
  - Note: usage of `next`
- Access field using position or name
  - Note: position starts at 1
  - Note: `<Type>`

```java
ResultSet rs = stmt.executeQuery("SELECT * FROM Students WHERE sid = " + selectedSID);
while rs.next()
{ sid=rs.getInt(1);
  sid=rs.getInt("sid")
}
```
PreparedStatements

- Allows a SQL statement to be parameterised:
  - More efficient
    - If it’s run many times
  - More secure
  - May be more convenient:
    - I.e. generic command with parameters supplied by user
    - quotes

- Provide the SQL statement when the object is created
- Use ‘?’ to mark parameter
- Use set methods to set parameter
  - Type defined in name
  - Specify by position
    - setInt (1, 123)
Example

PreparedStatement studentQuery = 
    dbConn.prepareStatement(
        "SELECT *"
        + " FROM Students"
        + " WHERE sid = ?");
studentQuery.setInt (1, 123);
ResultSet rs = 
    studentQuery.executeQuery();

- In all respects this is the same as a statement
  - Except:
    - SQL statement defined in constructor
    - So, method calls do not specify
    - Analysed only once, when created

- Nb: SQL injection
**SQL Injection**

User types in their sid:
- 12345
  - Executes:
    - SELECT * FROM Students WHERE sid = '12345'

What if they type:
- 12345' OR '1'='1
- 12345'; DROP TABLE Students; SELECT * ......
- 12345''--

Estimates:
- ~90% data breaches due to SQL injection
- 60-80% of websites found to be vulnerable

There are similar problems with other tools e.g. Javascript

```java
ResultSet rs = stmt.executeQuery("SELECT * FROM Students WHERE sid = " + stringReadFromUser + "");
```
SQL Injection & PreparedStatements

If you use PreparedStatements then you are safe from SQL injection

- Unless you contrive to make it vulnerable
  - E.g. by constructing the command string using user input

PreparedStatements are a win-win:

- Safer
- Faster
- Clearer
SQL Injection & PreparedStatements

If you use PreparedStatements then you are safe from SQL injection

- Unless you contrive to make it vulnerable
  - E.g. by constructing the command string using user input

PreparedStatements are a win-win:

- Safer
- Faster
- Clearer
Metadata

- It is possible to obtain metadata about:
  - The DBMS e.g. its capabilities
  - The DB e.g. the tables used
  - A resultSet e.g. the number of rows and columns, their types or names
- Cf. reflection in Java

This can be useful for many reasons:
- To build applications that are robust to differences between DBMS
- To build general purpose tools
- To allow for changes to the DB e.g.
  - Changes to attribute names
  - Changes to the number of attributes
  - To adapt to changes or to detect and report an error
Scrollable, updateable, dynamic & cached resultSets

The default resultSet only allows forward movement through a static table:

- This may be inconvenient
- It *may* be possible to have more convenient access
  - But it may not:
    - Depends on DBMS/drivers and query
  - And it may be inefficient!

Scrollable – move to any position
Updateable – modify resultSet and write changes back into DB
Dynamic – have resultSet updated to reflect changes in DB
CachedRowSets – allows work offline
SQLException

- Errors are signalled as Exceptions as you would expect:
  - And should be caught and handled appropriately

- You should have multiple catch clauses for different exception classes

SQLExceptions are unusual (compared to most java Exceptions):

- Ideally, you use the information in the exception object to decide what action to take.
Close

- You can (should) explicitly close `resultSets`, `statements` and `connections` when they are no longer needed.

- Closing a `connection` closes all its `statements`.
- Closing a `statement` closes all its `resultsets`
  - cached rowsets
Transactions

There are some compound operations which **must** be performed in their entirety (as though they were atomic)

- E.g. move money from one account to another
- If they are **partially** completed then:
  - The DB is inconsistent
  - There are real-world consequences

But:

- What if the application crashes part way
  - Or a communications failure occurs
- The DB server crashes
- Another user interferes
  - e.g. concurrently accesses these accounts
- We cannot complete the operation
- etc
The problem

1. Find Balance Acc1
2. Subtract transfer
3. Store new value
4. Find Balance Acc2
5. Add transfer
6. Store new value

Server Crash
So, if we can have multiple concurrent accesses then we can have a problem
Transactions

- The DBMS will provide support through transactions
  - Guarantees correct behaviour
    - ACID properties
  - Implements mechanisms that ensure correct behaviour and robustness
    - *Possibly* through locks and logs
    - ... there are other ways
  - No need (usually) to worry about implementation!

Note: this abstraction from the underlying mechanism has *some* similarity to the way Java provides synchronisation of threads

In JDBC:
- By default each operation is an independent transaction:
  - Conn.setAutoCommit (false)
    - The program must now explicitly handle transactions
- To commit a transaction:
  - Conn.commit()
    - The operations are now permanent
- To abort a transaction:
  - Conn.rollback()
    - The operations since the last *commit* are undone
- (As usual) there is the option for finer grain control e.g. named checkpoints
Transactions – why?

Why do we need transactions?
- Concurrent access
  - Across multiple applications
  - Maintain correct behaviour
- Robustness against failure
  - Hardware
  - Software
  - Communications

Queries (updates etc) may fail part way through:
- E.g. Violated constraints

An application may abort a transaction part way through:
- E.g. payment failure
Transactions – how?

Typically:

- Each transaction maintains its own copy of the database.
  - (conceptually)

- If the transaction is successful:
  - Write the changes through:
    - They are committed in the permanent database

- If the transaction is unsuccessful:
  - These changes are rolled back

Failure can be for several reasons:

- Explicit or implicit failure
- Conflict with other transactions
- Deadlock detection
Classically, the handling of transactions is divided as:

- **Optimistic**
  - We assume there won’t be a problem:
    - If there is we detect & fix it
    - E.g. we detect dirty reads (that data has changed) and repeat the transaction

- **Pessimistic**
  - We assume there is likely to be a conflict:
    - So we prevent it happening
    - E.g. we lock parts of our database until the transaction is committed

[And there are stages in between]
Transactions & performance (2)

The best DBMSs use very sophisticated techniques to:
- Deliver correct behaviour
- High performance

[The implication of this is that some use naïve methods which perform poorly or incorrectly]

Usually, the programmer can use their knowledge to achieve finer grain control:
- To improve performance
Transactions & Performance (3)

Transactions inevitably come with a performance overhead:
- If you don’t need them turn them off:
  - Read only databases
  - Maintenance operations

E.g. batching updates will be much quicker than individual transactions for each update
- [Actually, this is for several reasons not just transaction overhead]
Transactions (finally)

Transactions provide an abstraction from how the DBMSs implement the correct behaviour.

The efficiency with which different systems implement this varies.

The programmer can have fine grain control over how this works.

- But implementations vary.
An aside ....

Different DBMS implementations have different goals:
- So you need to choose appropriately

You also need to consider what is important:
- Cost?, Speed? Robustness? ....

One critical factor is the profile of your data & operations:
- Large amount of data?
- Frequent writes?
- Large amount of concurrency?
  - Of what type?
- Availability requirements etc.
And another ...

For reliability:
- Ups, multiple power/network connections ...
- Hot backups

For performance:
- Multiple servers
  - Distributed DB?
    - How?
  - Partitioned database?
Summary

- JDBC provides a convenient and efficient mechanism to access databases from Java programs
- Use the strengths of Java & DBMS appropriately
- JDBC programs are (fairly) portable across DBMSs
  - So stick to standards
- Java programs can be used to populate databases with real or synthetic data for testing etc.
- Connection can be expensive and connections are precious

- Prepared statements are preferred to statements
  - security
- There is limited type checking so greater care is needed
- Metadata is accessible
- There may be more convenient ways to perform some operations
- There are better ways of doing this than raw JDBC
  - hibernate
- Care should be taken to make code readable
- See online documentation for much more detail.
Lecture 13, 14 & 15
DataBase Design

Background

What is involved?
- Phases
- Methodologies and notations

Data Base Design

What is involved?
- The bigger picture
- ER Diagrams
  - For large data bases
- Mapping to DB Schema
- Example
Background

- Data bases are part of many information systems:
  - Existing data bases
  - New data bases

- They may serve one purpose:
  - ... but often they serve several
  - ... and often their *integrating* role is as important

It is necessary to describe data bases:

- As part of the design process
- To document an existing system
- As part of the modification and extension of existing systems
What is Involved?

There are many aspects e.g.:

- Requirements analysis
- Design
  - Conceptual/architectural/detailed ...
- Coding
- Testing
- Maintenance
- Documentation
- Project management

These may be:

- Undertaken at different levels of abstraction/detail
- Iterative

The emphasis will vary depending on the application

There is little that is special about DB design

So lessons & techniques carry across
Methodologies and Notations

Methodologies:
- Approaches to the task of:
  - Analysing the problem
  - Creating solutions
  - Testing the result
  - Etc.
- Usually include some heuristics and some metrics
- Different methodologies for different phases/classes of problem etc

Notations:
- A means of describing:
  - The problem
  - The solution
  - The process
  - Etc.
- Provide a means to:
  - Record
  - Communicate
Database Design

A database is usually part of some larger system:
- Capture data requirements
- Capture constraints

Non-functional requirements:
- Reliability, consistency, extensibility, security, performance …
- These drive design, analysis, testing etc. of DB component and integrated systems

Need to generate a description of the data, its organisation and its mapping onto the data base system

There are, obviously, many different ways to meet a set of requirements. Aim is to optimise the soft constraints on the design:
- Maximise generality/extensibility
- Maximise maintainability
- Minimise redundancy
- … And many more
- The balance is problem specific
There are many different methodologies that can drive this process and notations that can be used. Almost universally a data design is described using:

- **Entity Relationship (ER) diagrams**
  - Capture *conceptual* design
  - Data model

These can be easily *mapped* to a database schema. They may be partitioned for large complex designs.
Entity Relationship Diagrams

- Components of ER diagrams:
  - Oval – attributes
  - Rectangle – entity set
  - Diamond – relation set
  - Lines – showing links
  - Also:
    - Labels on lines indicating role
    - Cardinality
    - Double outlined boxes show weak entity set
Notes

There are a lot of detail variations
- Cardinality and optionality
  - Explicitly
  - As annotations to the diagram
- Some different conventions about details
  - E.g. primary key

There are additional features which can be used:
- Colour coding
- Type information
- Derived values
- Super/subclasses
- Constraints

…. But keep it simple for now
Attributes & Entity (sets)

- Attributes written in ovals
  - Primary key underlined
    - Dashed in weak entity sets
  - Can have composite attributes e.g. address
  - May be derived e.g. age
    - Written with dashed oval
  - May be multivalued
    - Drawn with double outline

- Entity set written in rectangle

[Notice: we are not interested in some details at this stage!]
Relationships

- Association between entities
- Name inside diamond
- Lines linking to:
  - Entity sets
  - Attributes
  - Relations

Roles can label the links if necessary:
Cardinality Restrictions

We can describe constraints on the participation of entities in a relation. For instance:

- A student must have at least 1 and possibly several tutors
- A tutor can tutor 0 or several students
- A car must have exactly 1 keeper (but that keeper may have >1 cars)
- Every driver must have exactly one licence
- A customer can have 0 or more accounts

What you are trying to express is:

- Whether an entity must participate (total) or whether it is optional (partial)
- If it participates, what are the limits on the number of times it can participate – max
- So we normally end up specifying:
  - Minimum – 0 or 1 (typically)
  - Maximum – 1 or many (typically)
Cardinality in ER Diagrams

There are three main ways these are typically used:

**Cardinality Representation:**

- Connectivity Ratio Representation:
  - PUBLISHER: (0,N)
  - PUBLISHED: (1,1)
  - BOOK

- **Crow’s Feet:**
  - Same interpretation as connectivity ratio, different syntax
  - PUBLISHER
  - PUBLISHED
  - BOOK
There are also (many) other ways to describe this. The clearest way is to explicitly label the maximum and minimum participation of the entity in the relation.
Weak entity sets

A weak entity set is one where the entities cannot (necessarily) be distinguished - there is no key.

Drawn using double lined boxes (for entity set and relationship).

They can only be identified uniquely in the context of an owner:
- E.g. dependents of an employee
- Emergency contacts for a student
Note:

- The critical point is an overview of the data design
  - Additional detail tends to obscure this:
    - And can be stored separately e.g. in the design tool or separate documentation
  - But, more importantly, we want to describe the conceptual design!
    - Which is later mapped to tables

- There will be several ways to solve the problem
  - Consider alternatives
  - Choose the most appropriate

- There is more than we have covered.

Remember

- Don’t duplicate data:
  - Space etc
  - Consistency
- Unpack data:
  - E.g. address
- Drive by needs of:
  - operations/users
- Design for the future
Mapping from ER diagrams to database design

The mapping from ER diagrams to a DB schema can be straightforward:
- Almost mechanical
- Some judgement & refinement is needed

The rules that follow should be seen as guidelines – you may want to preserve more of the structure at a cost in terms of space & computation

As usual there are different ways to do this
The design is mapped to:
- Tables
- Constraints within those tables
- Reflect modifications back into the ER diagram
Entity sets

These map straight into tables
- We know
  - The table name
  - Attributes and their names
  - Primary key
- For composite include the simple components
  - Or, maybe create a table

We should also know:
- Domain of each attribute
- Any other constraints
For weak entity sets

As for the normal case but:

- Include the key of the owner entity

Note:

Since the weak entity cannot exist without the owner:

- Deletion/update of the owner must be propagated to the weak entity:
  - ON DELETE CASCADE
Relationship sets

- Here the obvious approach is to map to a table:
  - This is necessary in some cases
  - This does have the virtues of:
    - Maintaining the structure
    - Avoiding unnecessary attributes in other tables
  - But there can be a cost in the operations – extra joins

- The ‘standard’ approach is to avoid creating an unnecessary table if possible
**1:1 relationship**

- For each 1:1 relationship
  - Add key of one entity as **foreign key** of other.
    - If possible, choose the recipient to be totally participating in the relationship.
  - Move attributes of the relationship to that entity
1:N relationship

- For each 1:N relationship,
  - add key of the entity on the N side as **foreign key** of the other.
  - Move attributes of the relationship to that entity.
M:N relationship

For each M:N relationship

- add a new relation containing
  - the keys of the participating entities as foreign keys
  - The attributes of the relationship

- The key of the new relation is the combination of the foreign keys
N-ary relations

For each n-ary relationship
  ■ as M:N relationships
    ♦ The key is all foreign keys.
    ♦ If one of the values is 1
      ■ the key can be reduced to just that foreign key
Mapping multi-valued attributes

For each multivalued attribute
  - add a new table containing:
    - an attribute representing the attribute itself
    - The containing entity’s primary key as a foreign key
  - The key is the foreign key and one or more of the attributes
Summary

ER diagrams are the standard way to describe the conceptual structure of a database.
- There are many detail variations on the representation
- There is more than we have covered
  - But this is the core

Mapping from ER diagrams to a database design is straightforward
- There are rules that can drive it
- Refinement of the design may occur during this mapping

Remember: this is only part of a larger process
Lecture 16

- Feedback on Exercise 1
- Discussion of Exercise 2
Lecture 17

Case Study

- A database for a car hire system
- Example - revisited
A car hire company needs to build a data base to manage its rentals:

Every rental must record the date and time at which the rental started and finished, the details of the driver and the details of the car rented. Each car has its registration number recorded, the date it was purchased and a record of all maintenance work undertaken. For a driver, the company needs to have their name, driver’s licence number, address and at least one telephone number.
A solution
A solution

- Is this okay? Shall we generate a unique ID?
- There could be Others!
- Is this a weak entity or should we have an ID: Mid?
- Is it sufficient to specify driver & car? Do we need a unique ID?
- There could be Others!
Map this to tables
Example – Let’s do it better!

Table name

Field or attribute names

Tuple (or row or record)

Column (or field or attribute)

Data Definition Language

CREATE TABLE Student (  
sid INTEGER,  
dob CHAR(10),  
login CHAR(20),  
course CHAR(10)  )

Query Language

SELECT sid, login  
FROM Student  
WHERE course='Se'

Marks

<table>
<thead>
<tr>
<th>sid</th>
<th>mid</th>
<th>mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>277</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>277</td>
<td>65</td>
</tr>
<tr>
<td>1</td>
<td>264</td>
<td>78</td>
</tr>
<tr>
<td>2</td>
<td>266</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>277</td>
<td>99</td>
</tr>
</tbody>
</table>
Summary

This was an *introduction* to database design (really only ER diagrams and mapping to DB), SQL and JDBC. See:
- Database course
- Documentation etc
- SE methodologies

There are a lot of details which cannot be covered
- But ....

Data bases are just a tool for data representation
- They should *serve* the larger system rather than drive it

Databases are part of many ISs
- Because they are already there
- Because they provide a useful tool

DBMSs provide a lot of essential services
- so using them is sensible

SQL is declarative and efficiently implemented
- Use SQL rather than Java

JDBC is well designed
- Simple to use
- Powerful and flexible
- … but there are probably better, more abstract, approaches which can be used on large problems