Databases

Dr. Richard E. Turner (ret26@cam.ac.uk)

Big-data: Databases

- Database = structured collection of data
- Everywhere: Facebook, MySpace, Google, Android (sqlite3), Amazon, ...
- AWS S3 - storage as a service (Dropbox, Tumblr, etc.)

Amazon S3 growth

Total number of objects stored vs. time (2006-2012)
Motivational example: what's wrong with this table?

<table>
<thead>
<tr>
<th>Item</th>
<th>Price</th>
<th>Supplier</th>
<th>Customer</th>
<th>Employee</th>
<th>EmployeeName</th>
<th>OrderNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lamp</td>
<td>50</td>
<td>Henderson</td>
<td>Smith</td>
<td>12/3/1990</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Chair</td>
<td>200</td>
<td>Ford</td>
<td>Jones</td>
<td>25/12/1995</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Table</td>
<td>350</td>
<td>Henderson</td>
<td>Jones</td>
<td>3/3/1993</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Chair</td>
<td>200</td>
<td>Ford</td>
<td>Jones</td>
<td>25/12/1995</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Desk</td>
<td>100</td>
<td>Henderson</td>
<td>Smith</td>
<td>12/3/1990</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

What are we going to cover in these lectures:

- Database design
  - Entity relationship modelling
  - Normalisation to ensure consistency, remove redundancy, avoid anomalies
- Database queries
  - Relational algebra
  - Manipulating data (relational database managers, SQL)
  - Relational algebra
- Distributed processing
  - Transaction processing
  - Concurrency control
  - Concurrency control (reordering relational operators, indices)
  - Reordering data (relational database managers, SQL)
  - Relationship modelling
  - Redundancy & slowdown query & slow to enter data

Database design
Conceptual Design - Entity Relationship Diagrams
Conceputal Design - Entity Relationship Diagrams
Semantics:

- Find the relationships between them
- Analogous to the object oriented approach: identify conceptual abstractions
- Notation suggested by Peter Chen in "The Entity Relationship Model: Toward a Unified View of Data", 1976 (UML can also be used)

Summary of ER diagrams
• **key**: uniquely identifies all possible rows that a relation could have.

• **superkey**: collection of attributes that uniquely identifies a row (can be many).

• **candidate key**: minimal superkey i.e. no subset of attributes in a candidate key may form a candidate key. There may be more than one candidate key.

example super-key = \{(StudentNo, CourseNo, SupNo)\}

example candidate-key = \{StudentNo, CourseNo\}

---

More about keys

---

example candidate-key = \{(StudentNo, CourseNo)\}

More about keys
More about keys

• **Primary key**: A superkey or candidate selected to have special status. Relations can have at most one primary key. Should be small and constant.

• **Foreign key**: If two relations R and S share an attribute k, then R[k] is a foreign key if k is the primary key of S. The foreign key does not necessarily uniquely identify the rows of R. Can comprise multiple attributes.

Examples:

- Three foreign keys = StudentNo, CourseNo, and SupNo.
**Normalization**

Formalise the process whereby we redesigned the attributes and relations to avoid logical inconsistencies arising when the database is used.

**Goal:** Minimise redundancy and dependency, thereby improving efficiency and extensibility.

Avoid logical inconsistencies arising when the database is used.

**Involves splitting data up into smaller relations**

- Efficiency (since redundant data is not stored),
- Extensibility (since changes to the database structure will only affect parts of the database),
- Consistency (since inconsistencies are prevented by the structure of the database).

**First Normal Form (1NF)**

- Each row/column intersection contains exactly one datum
- No duplicates
- No ordering over columns (permutations don't matter)
- No ordering over rows (permutations don't matter)

<table>
<thead>
<tr>
<th>Name</th>
<th>Course</th>
<th>Mark</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>40</td>
<td>Maths</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>80</td>
<td>Maths</td>
</tr>
<tr>
<td>Chris</td>
<td>3</td>
<td>70</td>
<td>Maths</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Course</th>
<th>Mark</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>40</td>
<td>Maths</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>80</td>
<td>Maths, Science</td>
</tr>
<tr>
<td>Chris</td>
<td>3</td>
<td>70</td>
<td>Maths, Science</td>
</tr>
</tbody>
</table>

**Bad:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Course</th>
<th>Mark</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>40</td>
<td>Maths</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>80</td>
<td>Maths</td>
</tr>
<tr>
<td>Chris</td>
<td>3</td>
<td>70</td>
<td>Maths</td>
</tr>
</tbody>
</table>

**Bad:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Course</th>
<th>Mark</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1</td>
<td>40</td>
<td>Maths</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>80</td>
<td>Maths</td>
</tr>
<tr>
<td>Chris</td>
<td>3</td>
<td>70</td>
<td>Maths</td>
</tr>
</tbody>
</table>

Make sure that your database really obeys the relational model.
First Normal Form (1NF)

- Make sure that your database really obeys the Relational Model:
  - No ordering over rows (permutations don't matter)
  - No duplicates
  - Make sure that your database really obeys the Relational Model:

Second Normal Form (2NF)

- A relation is in second normal form if:
  - All non-prime attributes depend on the whole candidate key
  - It is in first normal form

Each row/column intersection contains exactly one datum

- No duplicates
- No ordering over columns (perms of cols. and scheme doesn't matter)
- No ordering over rows (permutations don't matter)
Second Normal Form (2NF)

- A relation is in second normal form if:
  - It is in first normal form
  - All non-prime attributes depend on the whole candidate key

Sign that data should be split off into another relation

all non-prime attributes depend on the whole candidate key

A relation is in second normal form if:

- A relation is in second normal form if:

Second Normal Form (2NF)
Clarification:

A relation is in Third Normal Form (3NF) if it is in 2NF and every attribute is non-transitively dependent on the primary key (no dependency between non-primed attributes).

**Example:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Practice</th>
<th>Date</th>
<th>Demonstrator</th>
<th>Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom</td>
<td>9/2/13</td>
<td>2</td>
<td>Simon</td>
<td>2/2/13</td>
</tr>
<tr>
<td>Simon</td>
<td>9/2/13</td>
<td>2</td>
<td>Tom</td>
<td>2/2/13</td>
</tr>
<tr>
<td>Tom</td>
<td>11/2/13</td>
<td>5</td>
<td>Tom</td>
<td>4/2/13</td>
</tr>
</tbody>
</table>

**Table Description:**

- **Candidate Key:** {Name}
- **Non-prime attributes:** {Practice, Date, Contact}
- **Dependence:** Contact depends upon Demonstrator

**Validation:**

- The relation is in 2NF and in 3NF.
- Every attribute is non-transitively dependent on the primary key (Name).

**Conclusion:**

A relation is in Third Normal Form (3NF) if it satisfies the above conditions.
Other dependencies

The normal forms we've seen do not remove all redundancy.

Solution is to split into two tables:

<table>
<thead>
<tr>
<th>Treatment</th>
<th>ItemNo</th>
<th>Material</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warmings</td>
<td>35</td>
<td>Steel</td>
</tr>
<tr>
<td>Low-test</td>
<td>35</td>
<td>Steel</td>
</tr>
<tr>
<td>Oil-proof</td>
<td>27</td>
<td>Steel</td>
</tr>
<tr>
<td>Pressure test</td>
<td>27</td>
<td>Steel</td>
</tr>
<tr>
<td>Warmings</td>
<td>27</td>
<td>Plastic or oil-proof</td>
</tr>
</tbody>
</table>

Each row/column intersection contains one datum - 1NF.

Primary key = all three attributes.

- 2NF, 3NF

The normal forms we've seen do not remove all redundancy.

Continued example:

<table>
<thead>
<tr>
<th>Treatment</th>
<th>ItemNo</th>
<th>Material</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warmings</td>
<td>35</td>
<td>Steel</td>
</tr>
<tr>
<td>Low-test</td>
<td>35</td>
<td>Steel</td>
</tr>
<tr>
<td>Oil-proof</td>
<td>27</td>
<td>Steel</td>
</tr>
<tr>
<td>Pressure test</td>
<td>27</td>
<td>Steel</td>
</tr>
<tr>
<td>Warmings</td>
<td>27</td>
<td>Plastic or oil-proof</td>
</tr>
</tbody>
</table>

Each row/column intersection contains one datum - 1NF.

Primary key = all three attributes.

- 2NF, 3NF

The normal forms we've seen do not remove all redundancy.
Summary of Normalisation

1. Normalisation forms part of the process of database design. Higher normal forms, such as BCNF, 4NF, and 5NF, are not commonly used in practice due to their complexity.

2. You should always think about the information you want out of a database when designing it.

3. Selective denormalisation may help for performance reasons (see later).

4. Normalisation formalises part of the process of database design.

Relational Databases: Going Under the Hood

By late 1960s, "Software Crisis" was already declared.

Data storage was not doing much better.

1970: Ted Codd, an English mathematician working for IBM, published a paper

"A Relational Model of Data for Large Shared Data Banks," a paper that forever changed the way databases are designed.

Edgar 'Ted' Codd, 1923-2003

Image copyright IBM

...
Codd's Rules (inexhaustive flavour)

- **Relational Features**
  - Data represented purely as relations
  - To return data require: relation-name, primary key & attributes
  - Move away from hierarchical or navigational structure of early databases to simple tables
  - To return data require: relation-name, primary key & attributes

- **Closure**
  - Simple tables

- **Recap: Recurrence Database Terminology**
  - **Relation** = set of tuples
  - **Scheme** = set of attribute names
  - **Tuple** = list of components (not quite a set: ordering & repeats)
  - **Domain**
    - $D(EmployeeName) = \text{strings}$
  - **Relation = set of tuples**
  - **Scheme = set of attribute names**
  - **Tuple = list of components** (not quite a set: ordering & repeats)
  - **Domain**
    - $D(EmployeeName) = \text{strings}$
  - **Collection of relations = database**
    - Any possible row unique => key
  - **EmployeeName**

- **Closure**
  - Simple tables

- **Data represented purely as relations**
  - Must have relational features (and only these features)
Many relational database management systems (RDBMS) are accessed using SQL (Structured Query Language), which is defined by industry standards and has been developed over many revisions from SQL-87 to SQL 2008. SQL is designed by industry standards and has been developed over many revisions. SQLite is the easiest database to start using; requires no setup & available on the teaching system. SQL (Structured Query Language)
Projection

In relational algebra, repeated tuples will be deleted.

Example

\[
\Pi_{D}^{(L)} = \Pi_{D}^{(L)}
\]

\[
\Pi_{D}^{\alpha(D)}(\Pi_{E}^{(L)}L) = (\Pi_{D}^{\alpha(D)})L
\]
In SQL the `SELECT` operator also carries out SELECTION

\[
\text{SELECT} \ a \ \text{FROM} \ T \ \text{WHERE} \ C
\]

\[\Pi_a (\sigma_C (T)) \rightarrow \text{SELECT} \ a \ \text{FROM} \ T \ \text{WHERE} \ C\]

Selection

In relational algebra, repeated tuples will be deleted

Example

\[
\begin{pmatrix}
\text{Student} & \text{Course} \\
\text{Bob} & \text{Biology} \\
\text{Bob} & \text{Computer Science} \\
\text{Alice} & \text{Biology} \\
\text{Alice} & \text{Computer Science} \\
\end{pmatrix}
\]

\[\sigma_{\text{Student} = \text{Bob}}(T) \]

Selection eliminates tuples (rows)
- Union
  - Example: combine all customer information
    `SELECT * FROM CUSTOMERS_USA UNION SELECT * FROM CUSTOMERS_EUROPE`

- Intersection
  - Example: return all European customers outside of the UK
    `SELECT * FROM CUSTOMERS_EUROPE UNION SELECT * FROM CUSTOMERS_UK`

- Difference
  - Example: return all European customers outside of the UK
    `SELECT * FROM CUSTOMERS_EUROPE EXCEPT SELECT * FROM CUSTOMERS_UK`
Cartesian product

\[ |S| \times |H| = |L| \]

Also called the outer-product.

Example

\[ \{ S \supset w, s \} \times \{ H \supset v, u \} = S \times H = L \]

Relations with different schema

Cartesian product = only primitive operator for Joining.
Join

• Cartesian product \((R \times S)\) followed by selection on attributes \(a\) and \(b\) = "join"

• An non-primitive operator

\[
\begin{array}{c|c|c|c}
\hline
\text{Column} & \text{Column} & \text{Column} \\
\hline
\text{Lecturer} & \text{Lecturer} & \text{Lecturer} \\
\hline
\text{Name} & \text{No} & \text{Course} \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\hline
\text{Column} & \text{Column} & \text{Column} \\
\hline
\text{Course} & \text{Lecturer} & \text{Lecturer} \\
\hline
\text{Name} & \text{No} & \text{Course} \\
\hline
\end{array}
\]

Example 1:

\[
\text{SELECT * FROM LECTURER, LECTURER_COURSE}
\text{WHERE LECTURER.LecturerNo = LECTURER_COURSE.LecturerNo;}
\]

Example 2:

\[
\text{SELECT * FROM LECTURER JOIN LECTURER_COURSE}
\text{ON LECTURER.LecturerNo = LECTURER_COURSE.LecturerNo;}
\]

Natural Join

• Join results in repeated attributes =⇒ can perform projection to remove the repeated attributes

\[
((S \times R)_{uq=\upsilon p}^{\cap q}) \Pi = S \nabla R
\]

Example:

\[
\text{SELECT LECTURER_COURSE.CourseNo, LECTURER.Name}
\text{FROM LECTURER NATURAL JOIN LECTURER_COURSE}
\]

\[
\text{Example: SELECT FROM LECTURER, LECTURER_COURSE WHERE LECTURER.LecturerNo = LECTURER_COURSE.LecturerNo;}
\]

• Many joins in SQL (many for compatibility with other versions of SOL)

\[
(S \times R)_{uq=\upsilon p}^{\cap q} = S \nabla R
\]

• A non-primitive operator

Cartesian product \((S \times R)\) followed by selection on attribute \(a\) and \(b\) = "join"
Using operators to construct queries: SQLite3 Demo

1. Return unique student names who take courses mechanics or structures:
   - SELECT DISTINCT STUDENT.Name FROM STUDENT JOIN COURSE_STUDENT_SUPERVISOR ON STUDENT.StuNo = COURSE_STUDENT_SUPERVISOR.StuNo JOIN COURSE ON COURSE_STUDENT_SUPERVISOR.CourseNo = COURSE.CourseNo WHERE COURSE.Name = 'mechanics' OR COURSE.Name = 'structures';

2. Return all students who share supervisors with Eve:
   - SELECT DISTINCT STUDENT.Name FROM STUDENT JOIN COURSE_STUDENT_SUPERVISOR ON STUDENT.StuNo = COURSE_STUDENT_SUPERVISOR.StuNo JOIN COURSE ON COURSE_STUDENT_SUPERVISOR.CourseNo = COURSE.CourseNo WHERE COURSE_STUDENT_SUPERVISOR.SupNo IN (SELECT COURSE_STUDENT_SUPERVISOR.SupNo FROM STUDENT JOIN COURSE_STUDENT_SUPERVISOR ON STUDENT.StuNo = COURSE_STUDENT_SUPERVISOR.StuNo WHERE STUDENT.Name = 'Eve') AND STUDENT.Name != 'Eve';

3. Return supervisor on Brunel's courses:
   - SELECT SUPERVISOR.SupName FROM SUPERVISOR JOIN COURSE_STUDENT_SUPERVISOR ON SUPERVISOR.SupNo = COURSE_STUDENT_SUPERVISOR.SupNo JOIN LECTURER_COURSE ON COURSE_STUDENT_SUPERVISOR.CourseNo = LECTURER_COURSE.CourseNo JOIN LECTURER ON LECTURER.LecturerNo = LECTURER_COURSE.LecturerNo WHERE LECTURER.Name = 'Brunel' GROUP BY SUPERVISOR.SupName;

4. Return all courses for all of Alice's supervisions:
   - SELECT CourseNo FROM COURSE_STUDENT_SUPERVISOR JOIN STUDENT ON COURSE_STUDENT_SUPERVISOR.StuNo = STUDENT.StuNo JOIN SUPERVISOR ON COURSE_STUDENT_SUPERVISOR.SupNo = SUPERVISOR.SupNo WHERE STUDENT.Name = 'Alice';

5. Return coursenumbers for all of Alice's supervisions with supervisor Brown:
   - SELECT CourseNo FROM COURSE_STUDENT_SUPERVISOR JOIN STUDENT ON COURSE_STUDENT_SUPERVISOR.StuNo = STUDENT.StuNo JOIN SUPERVISOR ON COURSE_STUDENT_SUPERVISOR.SupNo = SUPERVISOR.SupNo WHERE STUDENT.Name = 'Alice' AND SUPERVISOR.SupName = 'Brown';
Find the pre-requisites of the pre-requisites for the structures course:

Rough idea

problem with attributes

Joining a table to itself
Joining a table to itself

Notice that we cannot use a traditional join like:

```
SELECT * FROM course_pa JOIN course_pa ON course_pa.course_no = course_pa.course_no
```

SQL version (renaming handled by "alias" - useful for long names too)

```
SELECT * FROM course_pa AS X JOIN course_pa AS Y ON X.course_no = Y.course_no
```

Find the pre-requisites of the pre-requisites for the structures course:
Speeding up queries

- **Expression Trees**
  - Selection and projection throw out data
  - Do this as soon as possible to make queries faster
  - Expression trees enable us to do this

- **Indices**
  - Secondary indices improve query performance but slow down data modification/addition/deletion
  - Relations are sorted by their primary index to accelerate queries

**Example**

\[
(S \times H)^{\mathcal{O}} = (S \times H)^{\mathcal{O}}
\]

Want to do selection as early as possible (e.g., before joins)

Selection pushing

Expression trees enable us to do this

- Do this ASAP to make queries faster
- Selection and projection throw out data
Expression tree example

\[(S)_{N} \Join (H)_{N} \Join (T)_{N} = (S \Join (H \Join T))_{N}\]

Projection pushing

\[\pi_{q}(S \Join (H \Join T))_{N} = (\pi_{q}(S) \Join (\pi_{q}(H) \Join T))_{N}\]

Want to do projection as early as possible (e.g. before joins)
Separate the selections

Push the selections below the first join

Return course numbers for all of Alice's supervisions with supervisor Brown
Return course numbers for all of Alice's supervisions with supervisor Brown

Push the projection through the first join

Push the selection below the second join
Push the projection through the second join.

Remove redundant projections.

Return course numbers for all of Alice's supervisions with supervisor Brown.

COURSE-STUDENT-SUPERVISOR

STUDENT

Supervisor name = "Alice"

SUPERVISOR

Supervisor name = "Brown"

COURSE

Return course numbers for all of Alice's supervisions with supervisor Brown.

SUPERVISOR

Supervisor name = "Brown"

COURSE

Return course numbers for all of Alice's supervisions with supervisor Brown.
Removeredundantprojections

SUPERVISOR
STUDENT
COURSE-STUDENT-SUPERVISOR
Return course numbers for all of
Alice's supervisions with supervisor
Brown
new table size
no. attributes:
SupNo,
CourseNo,
= 2
no. tuples:
selected for
"Alice"&"Brown"
= 2

Speeding up queries: indices

– trade-off between query speed and data-entry speed
– needs to be updated
– slows down data addition/modification/deletion since the data structure
  – speed queries on that new attribute

Secondary indices can also be added to speed queries
– data structure: binary trees or hash-tables typically used
– speed queries that use that attribute
– normally Primary Index = Primary Key
– Relations are normally stored in a sorted form via a Primary Index

Speeding up queries: indices

- primaryindex = primarykey
- datastructure: binarytreesorthash-tablestypicallyused
- speedqueriesonthatnewattribute
- slowdown data addition/modification/deletion since the data structure
  - trade-off between queryspeed and data-entry speed
What we have covered so far:

- Database design
  - Entity-relationship modelling
  - Normalisation to ensure consistency, remove redundancy, avoid anomalies
- Database queries
  - Relational algebra
  - Relational databases
  - Expression trees, indices
  - Manipulating data (relational database managers, SQL)