Data

What is data?

- A datum is a single fact about some universe of discourse.
  
  Jack Johnson has student number 20723081

- Data is the plural of datum, hence refers to a collection of facts.
  
  By itself data has little or no meaning.

- Information is the interpretation of data — attributing meaning to the data.
  
  As one writer has said:
  
  *Information is what you want; data is what you get.*

Databases

A database is an organized collection of data, usually representing a model of the activity of some business or other organization.

- Students, Courses, Units and Grades
- Customers, Products, Orders and Deliveries
- Doctors, Patients, Prescriptions, Medications
- Students, Books, Periodicals, Loans

The fundamental role of database technology is to allow users, typically organizational users, to extract information from their data.

Database Management Systems

A database management system (DBMS) is any system that allows users to manage their data, although nowadays the term is used almost exclusively for software rather than manual systems.

The amount of data that can be collected automatically is growing exponentially, and so there is a strong demand for database professionals, particularly database administrators.

Many different types of DBMS, but relational database systems are by far the most important in practice.
Advantages of a DBMS

Why use a DBMS

What are the benefits of using specialized DBMS tools rather than simply having data stored as a collection of files on a file system?

- Data Independence
- Efficiency
- Data Integrity
- Data Administration
- Concurrency Control
- Application Development

(Data independence provides analogous benefits to the encapsulation found in object-oriented programming languages:)

- Applications use a logical model of the underlying data, rather than directly manipulating the physical files storing the data
- Implementation of physical storage can be altered or improved without affecting client code
- Physical storage can be remote, or distributed, or both with no alteration in client code.

Data Independence

A critical role of a DBMS is to ensure that the entire collection of data is maintained in a consistent state.

At its simplest, this means that data should be stored in a single place — if several parts of an organization use personal data about an employee, and each stores that data separately, then it is difficult to enforce integrity.

More significantly, a change in one data item often has a “ripple effect” of consequences for data in other areas: a DBMS can ensure that all of these consequences occur and disallow an operation that would leave the database corrupt.

This is often described as the DBMS enforcing integrity constraints.

Efficiency

A DBMS can implement a number of storage strategies and optimizations to make the most common operations as fast as possible.

In particular, the DBMS can maintain various indexes to the data to make querying the database quick; the user can control which indexes are present, but need not know how they are implemented.

Database storage and indexing strategies are extremely sophisticated applications of data structures techniques.

(Data storage and indexing strategies are extremely sophisticated applications of data structures techniques.)
Data Administration

A DBMS allows the organization a fine degree of control over who is permitted various levels of access to the database.

In most operating systems (Unix, Windows etc) users can either read an entire file or none of it, but a DBMS can present different views of the same data to different groups of users.

For example, a lecturer may be able to look up a student’s academic record, but not their personal or financial details, while only certain staff will be able to alter their academic record.

Concurrency Control

In a large organization, there will often be several people accessing the same data item at the same time.

While this is not a problem if all users are simply viewing the data, it becomes a major problem if some of the users need to update the data.

For example, an airline reservation system may have several travel agents viewing availability at the same time, but the DBMS must prevent two agents from booking the same seat at the same time.

Application Development

Analysing data may require more sophisticated and application-dependent programs than a general-purpose DBMS can provide.

This can be accomplished by having a general purpose programming language such as Java and C accessing the data through the DBMS and then performing additional processing with the results.

This combination permits the developer to focus on the “business logic” of the application.

The power and success of this form of application development can be seen by the fact that essentially every large dynamic website is a database-backed application.

A large subject

Each of the topics listed above has enough theory, practice and technology associated with it to form an entire unit that could legitimately be called Databases.

Ramakrishnan & Gehrke identify two major approaches:

- Systems Emphasis

  Building database systems — the nuts and bolts of storage, indexing, query optimization, transaction management.

- Applications Emphasis

  Using database systems — data modelling, data query languages, database-backed applications.
This unit

We will take the applications emphasis covering
- The relational data model
- Relational algebra and calculus
- SQL, mySQL and Oracle
- Database-backed applications with Java (or C#)

Databases - Relational Data Model

The three-level architecture

An abstract view

The abstract structure of a modern DBMS is a 3-level architecture as follows:

An external schema is essentially a user's view of the database.

A single external schema is designed for one particular group of users, and presents to them a particular view of the database tailored to their requirements — it provides a fairly high-level logical view of the data.

For example, an external schema may be used to
- Hide sensitive or irrelevant data from certain users
- Combine existing data in specialized ways
A database may have many external schemata (schemas).
Conceptual Schema

The conceptual schema is a logical description of the data that is actually contained in the database. The conceptual schema describes what is contained in the database — it is a logical view of all the underlying data from which the external schemata can be created.

Constructing a conceptual schema is often a complicated and highly-skilled task, and in a large organization is one of the main roles of the database administrator (DBA).

A database has just one conceptual schema.

Physical Schema

The physical schema describes how the data is actually stored in the database.

Although many of the lowest level details (file names, compression etc) are part of the functionality of the DBMS itself, there are a range of choices that the DBA must make concerning the physical storage and the metadata.

- Choice of “storage engine”
- Choice of indexes

Although these choices do not affect the logical behaviour of the database, they can have a significant effect on its performance.

Many data models

Many data models have been proposed:

- Hierarchical Model
- Network Model
- Relational Model
- Object-Relational Model
- Object-Oriented Model

The hierarchical and network models are older models that provide only limited functionality, while the object-oriented model is not yet practical.
The dominant model

Currently, the relational model, introduced by Codd in 1970 is by far the dominant data model, and the vast majority of modern DBMS use this model.

- It is a declarative model both for specification and query
  This means that – to some extent – the user specifies what data they wish to specify or query and the DBMS then works out how to satisfy that request efficiently.

- It has just one fundamental concept called a relation
  This concept is expressive enough to model a useful portion of an organization’s activities, yet it is simple enough that it can be completely analysed mathematically.

Relations

The fundamental concept of the relational data model is the relation. Later we will define a relation mathematically, but informally we can view a relation as a table consisting of rows and columns:

<table>
<thead>
<tr>
<th>Surname</th>
<th>Name</th>
<th>State</th>
<th>Born</th>
<th>Died</th>
</tr>
</thead>
<tbody>
<tr>
<td>Washington</td>
<td>George</td>
<td>VA</td>
<td>1732-02-22</td>
<td>1799-12-14</td>
</tr>
<tr>
<td>Adams</td>
<td>John</td>
<td>MA</td>
<td>1735-10-30</td>
<td>1826-07-04</td>
</tr>
<tr>
<td>Jefferson</td>
<td>Thomas</td>
<td>VA</td>
<td>1743-04-13</td>
<td>1826-07-04</td>
</tr>
<tr>
<td>Madison</td>
<td>James</td>
<td>VA</td>
<td>1751-03-16</td>
<td>1836-06-28</td>
</tr>
<tr>
<td>Monroe</td>
<td>James</td>
<td>VA</td>
<td>1758-04-28</td>
<td>1831-07-04</td>
</tr>
</tbody>
</table>

Part of a table describing American presidents.

A practical example

The description above is very theoretical so we will consider an extended example that will ground these concepts.

This uses an example derived from the following book

- MySQL
- Paul DuBois
- ISBN 0672326736

This book is a comprehensive MySQL reference and tutorial.

Structure of a relation

A relation has a fixed number of columns, sometimes called attributes each of which has a name and a type.

Each row of a relation represents a single data item and specifies actual values for each of the attributes.

Therefore the relation AmericanPresident models an American president by his surname, name, state, birth and death date.

Each row of the table represents a specific American President.
A bit like a class!

class AmericanPresident {
    String surname;
    String name;
    String state;
    Date birth;
    Date death;
}

There is a helpful correspondence

A class description \(\leftrightarrow\) A conceptual schema

An object \(\leftrightarrow\) A row in the table

SQL

Structured Query Language or SQL is the “standard” database language for specifying and querying a relational database.

Initially developed by IBM in the mid-1970s, it was widely adopted and later standardized by ANSI/ISO.

- SQL-92
- SQL:1999
- SQL:2003
- SQL:2006→7→8?

Despite the existence of a standard version of SQL, every database vendor implements only an approximation to the standard.

Creating a database

A single MySQL server maintains a number of different databases.

Each database consists of a number of tables — this explains our loose definition of a relation as a table.

Typically an administrator will create a database for a specific user or group of users, who would themselves have more limited privileges.

mysql> CREATE DATABASE sampdb;
Query OK, 1 row affected (0.12 sec)

The mysql> is called the prompt at which the user (or administrator) enters commands.
Using a database

A user can only be “using” one database at a time, and any table-names refer to tables of that name in the “currently used” database.

```sql
mysql> use sampdb;
Database changed
```

This restriction makes it possible for different databases to have tables of the same name with no risk of confusion.

Creating a table

Creating a table involves specifying the table name and then the names and types of the columns (attributes).

```sql
CREATE TABLE president(
    last_name VARCHAR(15),
    first_name VARCHAR(15),
    state VARCHAR(2),
    birth DATE,
    death DATE
);
```

This command is part of the data definition language (DDL).

MySQL types

SQL and hence MySQL have many different data types, some of them being familiar types such as INT while others are much less familiar to us.

The types used in president are

- **VARCHAR(15)**
  
  A variable-length string of up to 15 characters; when assigned a value MySQL stores both its value and its length.

- **CHAR(2)**
  
  A fixed-length string that holds exactly two characters.

- **DATE**
  
  A date value, stored in **YYYY-MM-DD** format.
Adding data

The data manipulation language (DML) consists of the commands that are used to insert, delete, modify and query the rows in a table.

```sql
mysql> INSERT INTO president VALUES ('Washington', 'George', 'VA', '1732-02-22', '1799-12-14');
Query OK, 1 row affected (0.00 sec)
```

To insert a new row into a table, we use the command `INSERT INTO` and simply list the values for each of the columns/attributes.

Querying Data

The fundamental SQL command for querying data is the `SELECT` command.

```sql
mysql> SELECT * FROM president;
+-----------+------------+-------+------------+------------+
<table>
<thead>
<tr>
<th>last_name</th>
<th>first_name</th>
<th>state</th>
<th>birth</th>
<th>death</th>
</tr>
</thead>
<tbody>
<tr>
<td>Washington</td>
<td>George</td>
<td>VA</td>
<td>1732-02-22</td>
<td>1799-12-14</td>
</tr>
<tr>
<td>Adams</td>
<td>John</td>
<td>MA</td>
<td>1735-10-30</td>
<td>1826-07-04</td>
</tr>
<tr>
<td>Jefferson</td>
<td>James</td>
<td>VA</td>
<td>1751-03-16</td>
<td>1836-06-28</td>
</tr>
<tr>
<td>Monroe</td>
<td>James</td>
<td>VA</td>
<td>1758-04-28</td>
<td>1831-07-04</td>
</tr>
</tbody>
</table>
+------------+------------+-------+------------+------------+
```

The `*` is a wildcard character that means “everything” so this statement says “Select everything from the table `president`.”

Restricting Queries I

Of course, we don’t usually want to select everything from a database, so we can specify exactly which columns we do want.

```sql
mysql> SELECT last_name, first_name FROM president;
+------------+------------+
<table>
<thead>
<tr>
<th>last_name</th>
<th>first_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Washington</td>
<td>John</td>
</tr>
<tr>
<td>Adams</td>
<td>Thomas</td>
</tr>
<tr>
<td>Jefferson</td>
<td>James</td>
</tr>
<tr>
<td>Monroe</td>
<td>James</td>
</tr>
</tbody>
</table>
+------------+------------+
```

Restricting Queries II

We can also select only certain rows:

```sql
mysql> SELECT * FROM president WHERE state = "MA";
+-----------+------------+-------+------------+------------+
<table>
<thead>
<tr>
<th>last_name</th>
<th>first_name</th>
<th>state</th>
<th>birth</th>
<th>death</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>John</td>
<td>MA</td>
<td>1735-10-30</td>
<td>1826-07-04</td>
</tr>
</tbody>
</table>
+------------+------------+-------+------------+------------+
```

This command says to select each row that satisfies the condition expressed after the `WHERE` part of the statement.
And much more . . .

A language such as SQL is not as expressive as a general purpose programming language — one reason for this is that every query must terminate.

In general this means that each new construct is just added to the language as another keyword with its own — sometimes complex — syntax.

Thus learning MySQL involves mastering the most important constructs, and always working with a copy of the documentation alongside!

Relational terminology

Much of the power of relational databases comes from the fact that they can be described and analysed mathematically.

In particular, queries can be expressed with absolute precision using either

- The relational algebra
  A procedural language expressing how to extract information from the database.
- The relational calculus
  A declarative language expressing what information should be extracted from the database.

A set is simply a collection of objects, and is usually denoted by a capital letter such as $A$.

A set can be specified just by listing all of its members explicitly

$$A = \{\text{George}, \text{John}, \text{James Thomas}\}$$
$$B = \{\text{red}, \text{green}, \text{blue}\}$$
$$C = \{2, 4, 6, 8, 10\}$$

or sometimes by specifying a pattern

$$D = \{2, 4, 6, 8, \ldots\}$$
Membership

The symbol \(\in\) is used to denote membership of the set i.e. whether an object belongs to the set or not. Therefore

George \(\in A\)

but

violet \(\notin B\)

Similarly

102 \(\in D\)

but

999 \(\notin D\)

Defining Properties

A more precise way of defining the set \(D\) is to use a defining property of the set.

\[ D = \{ x \mid x \text{ is a positive even integer} \} \]

This way of defining a set is reminiscent of the WHERE statement of SQL.

Sets can be defined in terms of other sets like this

\[ E = \{ x \mid x \in D \text{ and } x \text{ is a square} \} \]

This syntax is now almost identical to an SQL statement.

Cardinality

The cardinality of a set is its size; some sets are finite in that they have a specific number of elements while some are infinite.

The sets \(A\), \(B\) and \(C\) are all finite, while \(D\) is infinite.

There is a very special set \(\emptyset\) that is called the empty set because it contains no elements.

Clearly most of the sets that we deal with in databases are finite.

Subsets and Supersets

A set \(X\) is a subset of a set \(Y\), denoted

\[ X \subseteq Y \]

if every member of \(X\) is also a member of \(Y\).

There are two special cases:

\[ \emptyset \subseteq Y \]

and

\[ Y \subseteq Y \]

If we want to express the fact that \(X\) is a proper subset of \(Y\) then we say

\[ X \subset Y. \]
**Union**

Given two sets \( X \) and \( Y \), the *union* of \( X \) and \( Y \) is the set containing all the objects that are members of \( X \) or members of \( Y \). In symbols, we have 

\[
X \cup Y = \{ x \mid x \in X \text{ or } x \in Y \}.
\]

Therefore we have

\[
A \cup B = \{ \text{George, John, James Thomas, red, green, blue} \}
\]

and

\[
C \cup D = D.
\]

**Intersection**

Given two sets \( X \) and \( Y \), the *intersection* of \( X \) and \( Y \) is the set containing all the objects that are members of \( X \) and members of \( Y \). In symbols, we have 

\[
X \cap Y = \{ x \mid x \in X \text{ and } x \in Y \}.
\]

Therefore we have

\[
A \cap B = \emptyset
\]

and

\[
C \cap D = C.
\]

**Cartesian Product**

For database purposes, the Cartesian product is one of the most important set operations.

Given two sets \( X \) and \( Y \), the *Cartesian product* \( X \times Y \) is the set of *ordered pairs* with components from \( X \) and \( Y \) respectively:

\[
X \times Y = \{ (x,y) \mid x \in X, y \in Y \}
\]

Notice that the *elements* of \( X \times Y \) are *pairs*.

**Cartesian Product Example I**

If 

\[
A = \{ \text{George, John, James Thomas} \}
\]

and

\[
F = \{ \text{Washington, Adams, Jefferson, Madison} \}
\]

then \( A \times F \) contains all of

\[
\begin{align*}
(\text{George, Washington}) & \quad (\text{George, Adams}) & \quad (\text{George, Jefferson}) & \quad (\text{George, Madison}) \\
(\text{John, Washington}) & \quad (\text{John, Adams}) & \quad (\text{John, Jefferson}) & \quad (\text{John, Madison}) \\
(\text{James, Washington}) & \quad (\text{James, Adams}) & \quad (\text{James, Jefferson}) & \quad (\text{James, Madison}) \\
(\text{Thomas, Washington}) & \quad (\text{Thomas, Adams}) & \quad (\text{Thomas, Jefferson}) & \quad (\text{Thomas, Madison})
\end{align*}
\]

If \( X \) and \( Y \) are both finite, then \( X \times Y \) has \( |X| \times |Y| \) elements.
### Cartesian Product Extended

If $X$, $Y$ and $Z$ are three sets then the Cartesian product $X \times Y \times Z$ defined by

$$X \times Y \times Z = \{(x, y, z) \mid x \in X, y \in Y, z \in Z\}$$

has ordered triples as its elements. Sometimes we shorten $X \times X$ to $X^2$ and $X \times X \times X$ to $X^3$ and so on.

### Cartesian Products in a DB

Consider a variable like `first_name` which has type `VARCHAR(15)`. This variable can take on a finite number of legal values, and the entire collection of legal values is called the domain of the variable — suppose we call this $D$. Then the entire range of possible `(first_name, last_name)` combinations is the set $D \times D$.

The collection of `(first_name, last_name)` combinations that is actually used in the database is therefore just a subset of $D \times D$.

### Extended Cartesian Product Example

Suppose that $X = \{0, 1, \ldots, 255\}$

Then $X^3$ is the set of all triples

$$\{(x, y, z) \mid 0 \leq x, y, z \leq 255\}$$

This set could represent, for example, the set of all RGB-colours.

### Relations

Finally, we are in a position to define a (binary) relation:

**Definition**

A binary relation between two sets $X$ and $Y$ is a subset of $X \times Y$.

Given a binary relation $R \subseteq X \times Y$ we say that $x$ is related to $y$ in the relation if $(x, y) \in R$. 

Relation Example I

Consider the following subset of our earlier example product:

(George, Washington)  (George, Adams)  (George, Jefferson)  (George, Madison)
(John, Washington)    (John, Adams)    (John, Jefferson)    (John, Madison)
(James, Washington)   (James, Adams)   (James, Jefferson)   (James, Madison)
(Thomas, Washington)  (Thomas, Adams)  (Thomas, Jefferson)  (Thomas, Madison)

Under this relation, two names are related if they form the name of one of the first four American presidents.

Relation Example 2

Let $X$ be the set of all positive integers, and define a relation as follows:

$$R = \{(x, y) \mid x \text{ is a divisor of } y\}.$$ 

It follows that

$$(2, 4) \in R \quad (5, 100) \in R \quad (3, 997) \notin R$$

Relation Example 3

Let $X$ be the set of all issued UWA student numbers, and let $Y$ be the set of all units at UWA.

Then we can define a relation EnrolledIn such that

$$(s, c) \in \text{EnrolledIn}$$

if and only if the student with student number $s$ is enrolled in the unit with code $c$.

Thus a typical element of this relation might be something like

$$\text{(10423884, CITS2232)}.$$
**One-to-many**

What happens if we alter the relation to include the fifth president, James Adams?

This relation is now called *one-to-many* because it is possible for *one* element of \( x \) to be related to *many* elements of \( y \).

---

**Entity Sets**

An *entity* is an object or event in which we are interested — an entity is described using a set of *attributes*.

For example, we might use the attributes *student-id* and *name* to describe a student. Then each individual student has a particular *value* for each attribute.

An *entity set* is a collection of similar objects in that every entity in the set has the same attributes — not the same *values*, but the same attributes.

For example, the set of all students is an *entity set* with two attributes.

---

**Many-to-many**

Frequently a relation is *many-to-many* such as the “is a divisor of” relation discussed earlier.

---

**Diagrammatic Representation**

An entity set is described by using a specific type of diagram; a rectangle represents the entity set, while ovals represent the attributes.
Entity Sets and Relations

Each attribute of an entity set has a domain, which is the complete range of legal values for that attribute.

So if $D_1$ is the set of all legal student numbers, and $D_2$ the set of all legal names, then a specific entity set can be described just as a particular subset of $D_1 \times D_2$.

Therefore an entity set with two attributes with domains $D_1$ and $D_2$ is nothing more than a binary relation between $D_1$ and $D_2$.

(Gf Royle, N Spadaccini 2006-2010)

Entity Sets in an RDBMS

The whole point of all of this terminology is that in a relational database an entity set with two attributes is represented by a binary relation.

<table>
<thead>
<tr>
<th>Student-id</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>30781233</td>
<td>John Smith</td>
</tr>
<tr>
<td>30721438</td>
<td>Emily Tan</td>
</tr>
<tr>
<td>30611345</td>
<td>Mark James</td>
</tr>
<tr>
<td>30551001</td>
<td>Jennifer Lee</td>
</tr>
</tbody>
</table>

The relation schema is the names and types of the columns, while the relation instance is the collection of rows.

(Ternary Example)

Let $S$ be the set of all students, $U$ the set of all units and $L$ the set of all lecturers.

Then we could define a ternary relation

$\text{Teaches} \subseteq L \times U \times S$

where

$(l, u, s) \in \text{Teaches}$

if $l$ is the lecturer for unit $u$ being taken by student $s$.

The information contained in a ternary relation cannot be represented just by binary relations between $S$, $U$ and $L$.

Arity

Of course, most entity sets have more than two attributes so cannot be modelled by a binary relation.

More generally, if $D_1, D_2, \ldots, D_n$ are $n$ sets then a relation with arity $n$ is defined to be a subset

$R \subseteq D_1 \times D_2 \times \cdots \times D_n$.

A relation with arity 2 is binary, while a relation with arity 3 is called ternary.

(Ternary Example)

Let $S$ be the set of all students, $U$ the set of all units and $L$ the set of all lecturers.

Then we could define a ternary relation

$\text{Teaches} \subseteq L \times U \times S$

where

$(l, u, s) \in \text{Teaches}$

if $l$ is the lecturer for unit $u$ being taken by student $s$.

The information contained in a ternary relation cannot be represented just by binary relations between $S$, $U$ and $L$. 
**Databases - Relations in Databases**

**Re-capping - data model**

A data model is a *precise, conceptual* description of the data stored in a database.

The relational data model is a data model where the data takes the form of *relations*.

A relation is made up of *tuples*.

**Re-capping - What is a tuple?**

A tuple is an *ordered* list of values.

We write a tuple as comma separated values within parentheses, e.g.

(Wayne’s World, 1992, 95, comedy, Paramount)

This tuple contains five values, and the order is significant.

(Paramount, Wayne’s World, 1992, 95, comedy)

is a different tuple.

**Re-capping - What is a relation?**

Mathematically a relation is a *set* of tuples.

A relation is defined on certain *domains*, which are the sets from which the values of the tuple come.

(Wayne’s World, 1992, 95, comedy, Paramount) could be a tuple of a relation defined on the domains,

- text strings
- integers
- integers
- {'comedy', 'thriller', ...}
- text strings
Tuple attributes

Each component of a tuple is assigned a name, referred to as an attribute. (Wayne’s World, 1992, 95, comedy, Paramount) could have the attributes of title, year, length, genre and company.

The attribute genre always refers to the fourth value of all tuples of this relation, and for this particular tuple it refers to comedy.

Representing relations

We usually write relations as two-dimensional tables, with the attributes listed as a row header and the tuples (the data) as simple rows.

Example:

<table>
<thead>
<tr>
<th>title</th>
<th>year</th>
<th>length</th>
<th>genre</th>
<th>company</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star Wars</td>
<td>1977</td>
<td>124</td>
<td>‘sci fi’</td>
<td>Fox</td>
</tr>
<tr>
<td>Gone with the Wind</td>
<td>1939</td>
<td>231</td>
<td>drama</td>
<td>MGM</td>
</tr>
<tr>
<td>Wayne’s World</td>
<td>1992</td>
<td>95</td>
<td>comedy</td>
<td>Paramount</td>
</tr>
</tbody>
</table>

Schemas

A relation is described using a schema. The schema consists of:

- a name for the schema
- a tuple of its attributes, optionally with types

The relation we have been using could have the schema:

Movies(title, year, length, genre, company)

or with data types

Movies(title: string, year: integer, length: integer, genre: string, company: string)

and in SQL

Movies(title VARCHAR(3), year INT, length INT, genre VARCHAR(15), company VARCHAR(30))

Actually the order of the attributes is arbitrary, but once chosen, all tuples must adhere to that order.
Relations, tables and schemas

The word “relation” is used in a number of contexts:
- In referring to the data model, as described by a schema
- As a concrete example of data tuples

To be specific one refers to the former as the **schema** and the latter as the **relation instance**.

---

Cartesian Products are relations

From the previous slide you will note that the result of the Cartesian Product of two or more relations **is itself a relation**. Equally the result of a query on one or more tables is itself a table.

We will see in future lectures that explore SQL in greater detail, that this feature is very powerful. It enables you to query on the result of some other query, better described as SQL’s subquery semantics.

---

Cartesian Product

Consider the two (instance) relations $R$ and $S$.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

What results from $R \times S$?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>9</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

---

Cartesian Product in SQL - the **join**

In SQL the syntax for the Cartesian Product of two tables is

```
SELECT *
FROM R, S;
```

But what results from the query?

```
SELECT a, b
FROM R, S
WHERE a = 1;
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Relations in RDBMSs

The mathematical formulation of a relation as a set of tuples, and its actual realisation in a DBMS as tables gives rise to some differences.

- Terminology is mixed: Rows or records vs tuples, a table vs a relation, attributes vs fields, . . . .
- A RDBMS makes certain implementation compromises for efficiency, storing each row/tuple in a specific order, and stores them as a list and not a set.
- From the previous slide a resulting relation may have the same tuple repeated several times. The RDBMS implements bags of tuples, not sets of tuples.

Terms to understand

- data model
- relational data model
- tuple
- component of a tuple
- data type of a component
- attribute
- relation
- schema
- instance relation

Some myths

Myth 1
A relational database is a database in which the data is physically stored as tables.

Myth 2
A relation is just a flat file.

Myth 3
A relation model is just flat files.

Myth 4
Relational systems must necessarily perform poorly.

Myths cont.

Myth 5
Relational databases use more storage than other kinds of databases.

Myth 6
The relational approach is intended purely for query, not for production systems.

Myth 7
Relational databases involve a lot of redundancy.

Myth 8
The relational model is just theory.
Database Design Process

Ramakrishnan & Gehrke identify six main steps in designing a database:
- Requirements Analysis
- Conceptual Design
- Logical Design
- Schema Refinement
- Physical Design
- Application & Security Design

Requirements Analysis

Requirements Analysis is the process of determining what the database is to be used for.
It involves interviews with user groups and other stakeholders to identify what functionality they require from the database, what kinds of data they wish to process and the most frequently performed operations.
This discussion is at a non-technical level and enables the database designers to understand the business logic behind the desired database.
Conceptual & Logical Design

Using the ER data model, the conceptual design stage involves identifying the relevant entities and the relationships between them and producing an entity-relationship diagram.

The logical design stage involves translating the ER diagram into actual relational database schema.

Once a suitable ER model has been constructed, then this can be translated into a relational database fairly easily.

However ER modelling is as much an art as a science, as there are usually many choices to be made and the consequences of each choice sometimes does not become apparent until problems arise later.

Iterative Process

As with all software engineering processes, information uncovered during later phases of the project may alter some design decisions so the nice neat diagram of the 6 phases is really an iterative process where each stage feeds back to the previous stages.

One of the major causes of design alterations is incomplete requirements analysis — this is frequently attributed to users not being aware of the possibilities until they start using the system, or at least a prototype.

After the modelling

We will cover the remaining steps of the process later as they involve the effective use of the database after it’s fundamental structure has been determined.

This involves

- Mathematical analysis and refinement of the schema
- Performance-based decisions on indexing, machine capacities, required performance etc.
- Interfacing with client applications and security

Running Example

We use a modified version of Exercise 2.3 in the text as an example. This question asks the user to produce an ER model from the following requirements:

- A lecturer has a staff number, a name and a rank
- Research projects have a project id, a sponsoring organization and a budget
- Each project has one lecturer as a principal investigator
- Each project may have other lecturers as co-investigators
- Each lecturer can be principal or co-investigator on multiple projects
Entity Sets

We have already introduced the concept of an *entity-set* and explained how that can be diagrammed and then translated directly into a relational database.

One detail that we omitted is that a relation should be a *set* and therefore cannot contain any duplicates elements — the corresponding database table should not have any duplicate rows.

A *key* for an entity set is an attributed, or combination of attributes that is guaranteed to distinguish the elements.

Example key

A Student is always uniquely identified by their *student-id* so this attribute is a suitable *key* for this relation.

In an ER diagram, an entity set’s key is designated by underlining the attribute or attributes that form the key.

Why are keys important?

It is important to identify the key for an entity set and indeed — as we will see later — for any relation in a database for several reasons:

- Explicitly identify a key ensures that the data model is logically consistent.
- When implemented, the DBMS can enforce *key constraints* that ensure that the nominated key does indeed uniquely identify each row.
- The DBMS can *index* the table using the key values and manipulate that relation very efficiently.
- The DBMS can enforce *referential integrity* by ensuring that tables that refer to other tables remain in a consistent state — this comes later!

Example – Lecturer Entity

Modelling of the *entities* in the example is fairly easy:

So the actual entities would be things like

<table>
<thead>
<tr>
<th>staff-id</th>
<th>name</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>00144238</td>
<td>&quot;J Smith&quot;</td>
<td>&quot;Associate Professor&quot;</td>
</tr>
</tbody>
</table>
Example – Project Entity

![Diagram of Project Entity]

So the actual entities would be things like

```
proj-id  sponsor  budget
DH10304   ARC     5120000
```

Hmmm, what about the key

Although it is unlikely that two sponsoring organizations use the same format for project id numbers, it is possible. So maybe the key should consist of the two attributes (proj-id, sponsor)?

![Diagram of Project Entity with underlined attributes]

In this case, the attributes that form the key are all underlined.

Relationship Sets

The power of relational databases comes from the ability to model and query relationships between entity sets.

If $E_1, E_2, \ldots, E_n$ are entity sets, then a relationship set is a subset

$$R \subseteq E_1 \times E_2 \times \cdots \times E_n$$

In other words $R$ is an $n$-ary relation whose elements are entities.

As entity sets are also relations, this means that we are using relations/tables to model both entity sets and relationship sets!

Diagramming Relationships

A relationship is diagrammed by a named diamond shape that is connected by lines to the related entity sets.

Here we are modelling the relationship Principal which relates projects to their principal investigators.
Elements of the relationship set

The actual elements of the relationship set are the pairs that specify which lecturer manages which project.

Thus if Associate Professor Smith manages the project DH10304 then that pair of entities would be an element of the Principal relationship set.

Notice that we can unambiguously specify this pair just by storing the keys for this pair:

\[(00144238, DH10304) \in \text{Principal}\]

This of course is exactly how an RDBMS stores a relationship set.

A sample relationship set

Each black circle represents one element of the Principal relation.

Participation Constraints

If the participation is a constraint based on the business logic (rather than just an accident of this particular set of data), then it can be encoded into the ER diagram (and subsequently enforced by the DBMS).

Every project must have a principal investigator and so there is total participation of the entity set Projects in the relationship set Principal.

This is indicated in the ER diagram by a thick black line connecting the entity set with the relationship set.
Key constraints

The relationship \textit{Principal} is one-to-many because each project has just \textit{one} principal investigator.

This is called a \textit{key constraint} because it means that in any allowable instance of \textit{Principal} each entity from \textit{Project} appears at most once.

In particular, this means that the key for \textit{Project} can be used as a key for \textit{Principal}.

This is indicated in the ER diagram by an arrow-head on the line connecting the entity set with the relationship set.

Other conventions

There are lots of other conventions for ER diagrams that include other symbols, or possibly little numbers indicating the exact form of the relationship etc., thus making it more detailed and more expressive.

We deliberately use this very simple form of ER diagramming because the constraints that are used in this model can all be \textit{implemented} in standard SQL, and thus the database model corresponds precisely to the ER diagram.
Additional Requirements

So far we have only considered a few of the requirements for the university database — here are some more.

- Departments have a department number, a department name and a contact phone number
- Departments have a lecturer who is the head of department
- Lecturers may work in more than one department, in which case they have a percentage of their time associated with each department

Another entity set

These requirements introduce another straightforward entity set.

By now the procedure for identifying entity sets and their attributes and key should seem straightforward — at least for these very well specified small examples!
Headship – key constraints

Next we cover the *key constraints*. Each department has exactly one lecturer as head and so here we have a key constraint — each *department* can only appear in one tuple of the relationship set. This is indicated by an arrow which goes *from* the entity set that is constrained to appear only once.

The *Works-In* relationship set

At first sight the *Works-In* relationship set is straightforward:

Note the thick lines indicating that every lecturer works in some department, and every department has some lecturers in it.

What’s the problem?

Suppose Associate Professor Smith works 60% of the time for CSSE and 40% of the time for Maths.

The ER diagram gives us no mechanism for storing the percentages at the moment. We cannot extend the entity set *Lecturer* to contain a field *percent* because that field makes no sense when considering a lecturer in isolation.

<table>
<thead>
<tr>
<th>staff-id</th>
<th>name</th>
<th>rank</th>
<th>percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>00144238</td>
<td>“J Smith”</td>
<td>“Associate Professor”</td>
<td>??????</td>
</tr>
</tbody>
</table>

Similarly we can not add this field to the entity set *Department*. 
Relationship Attributes

The percentage time is not an attribute of a lecturer alone, nor of a department alone, but in fact is an attribute of a *relationship* between a lecturer and a department.

So in our example, the 60% amount is associated with the *pair of entities* (Smith, CSSE) and the 40% amount is associated with the *pair of entities* (Smith, Maths).

So percent is actually an attribute of the *relationship set*, not of any of its constituent entities.

Diagramming relationship attributes

An oval-shaped attribute is added to the relationship symbol to indicate that this attribute “belongs to” the relationship itself rather than to its components.

Example

Thus for our example, if \( L \) is the lecturer entity set, and \( D \) the department entity set and \( P = \{ p \mid 0 \leq p \leq 100 \} \) then

\[
\text{Works-In} \subseteq L \times D \times P
\]

So now we can say sensible things like

\[
(\text{Smith, CSSE, 60}) \in \text{Works-In}
\]

to capture the fact that Smith is 60% associated with CSSE.

Notice however that the ER model is *NOT* strong enough to specify various other types of “business logic” such as ensuring that the percentages for each lecturer add up to 100.

Formally

R & G sidestep this issue, but their formal definition of a relationship set needs to be modified.

If \( E_1, E_2, \ldots, E_n \) are entity sets and \( A_1, A_2, \ldots, A_m \) are sets (i.e. attribute sets) then a *relationship set* is a subset

\[
R \subseteq E_1 \times E_2 \times \cdots \times E_n \times A_1 \times \cdots A_m
\]

We think of an element of the relationship set as consisting of \( n \) entities (which of course possess their own attributes), together with \( m \) values that describe aspects of the relationship.
Relationships between a set and itself

In our definition of a relationship set, there was no requirement for the entity sets $E_1, E_2$, etc to be different.

Suppose that the university commences a formal system of peer review of teaching and specifies that:

- Each lecturer has another lecturer, called a peer reviewer who attends a few of their lectures and provides feedback.

Thus we have a binary relationship set in which both components come from the Lecturer entity set.

Roles in an ER diagram

Roles

In this context an element of this relationship set is a pair of lecturers such as

\[(Smith, Jones)\]

Although both components are from the Lecturer entity set, they occur in this relationship set with different roles.

One of the two lecturers is the reviewer and the other is the review subject.

In an ER diagram this is indicated by a single occurrence of the entity set, but connected multiple times to the relationship diamond with each line labelled with the relevant role.

Aggregation

So far we have covered relationship sets that involve a number of entity sets and that may (or may not) have relationship attributes.

However in a number of circumstances we want to model a relationship between an entity set and a relationship set.

This can be achieved by using a feature of the ER model called aggregation which provides a way of indicating that an entire relationship set is participating in another relationship set.

Essentially it allows us to view a relationship set as a “special sort” of entity set.
When is it used?

Suppose that we have additional requirements for research projects as follows.

- Graduate students have a student number, a name and a degree course.
- Each research project has one or more graduate students working on it.
- When graduate students work on a project they have one supervisor for their work on that project; they can work on more than one project in which case they may have different supervisors for each one.

First approach

This introduces a ternary relationship called Supervises that relates a graduate student, a project and a lecturer.

The elements of this relationship set would be triples containing one graduate student, one project and a lecturer. So a triple of the form 

$$ (g, p, l) $$

could be interpreted as meaning

Graduate $g$ is supervised by lecturer $l$ while working on project $p$.

This does capture some of the requirements — but what about participation and key constraints?
Key constraints

The problem is that we cannot correctly model the key constraints — that each “graduate student working on a project” should have a unique supervisor.

The problem is that we really want a binary relationship that has \((g, p)\) as one of its components and \(l\) as the other.

This is exactly the concept of aggregation in that we treat the relationship set \(\text{Works-On}\) as an entity set in its own right.

---

Aggregation in ER diagrams

A balancing act

Even with these additions, there are many aspects of a normal business that an ER diagram cannot capture.

The reason that relational databases have been so very successful is because

- The ER model is strong enough to model a useful proportion of the activity of a business, yet
- The ER model is simple enough to be implemented and mathematically analysed

More expressive models are hard to implement and optimize, while less expressive models omit important functionality.
This lecture

This lecture starts formal coverage of SQL, in particular MySQL.

Overview

- **Data Definition Language (DDL)**
  - Creating, deleting and altering tables with `CREATE TABLE`, `DROP TABLE` and `ALTER TABLE`.
  - Working with views.
- **Data Manipulation Language (DML)**
  - Inserting, updating and deleting rows with `INSERT`, `UPDATE` and `DELETE`.
  - Querying the database with `SELECT` combining one or more tables.
  - Summarizing data with aggregate functions and `GROUP BY`.
- **Advanced functions**
  - Stored procedures, functions and triggers.

Another example

For these lectures we'll use another example based on Paul Dubois' book MySQL with the following specification. The database will keep the marks of students taking a particular unit.

- Students have a first name, a sex and a unique student number
- Grade Events are either tests or quizzes and happen on a particular date
- Students take tests or quizzes and get a score for that grade event
Creating tables

We have already seen some examples of creating tables simply by listing the attributes and their types inside a `CREATE TABLE` statement.

```
CREATE TABLE student (  
    name VARCHAR(20) NOT NULL,  
    sex ENUM('F','M') NOT NULL,  
    student_id INT NOT NULL AUTO_INCREMENT,  
    PRIMARY KEY (student_id)  
) ENGINE = InnoDB;
```

This definition contains only features that we have seen before, but there are several others we will meet later.

Deleting tables

Deleting tables is easy

```
DROP TABLE student;
```

If the table does not exist, then an error message will be generated if you try to drop it; you can avoid this error message by first checking that the table exists.

```
DROP TABLE IF EXISTS student;
```

This first checks that `student` exists and then drops it if so, otherwise it does nothing.

Changing tables

Tables can be changed with the `ALTER TABLE` command; notice that this alters the relation `schema` and not an individual row.

```
CREATE TABLE grade_event (  
    date DATE NOT NULL,  
    category INT NOT NULL,  
    event_id INT NOT NULL AUTO_INCREMENT,  
    PRIMARY KEY (event_id)  
) ENGINE = InnoDB;
```

This table has been entered incorrectly because it has `INT` for the category which is either a test or a quiz.
Altering a column type

ALTER TABLE grade_event
CHANGE category
category ENUM('T','Q') NOT NULL;

The old column category is named and then an entirely new column definition (which may include a new name) is given.

Other Alterations

There are many other things that can go after ALTER TABLE

- ALTER TABLE ADD COLUMN column-definition
- ALTER TABLE ADD INDEX...
- ALTER TABLE ADD FOREIGN KEY...
- ALTER TABLE ADD PRIMARY KEY...

Anything that you can ADD with ALTER TABLE, you can also DROP in the same way.

You can also make changes that do not alter the logical structure of the table, but have some other effect:
ALTER TABLE student ENGINE = MyISAM;

The score table

CREATE TABLE score (
    student_id INT UNSIGNED NOT NULL,
    event_id INT UNSIGNED NOT NULL,
    score INT NOT NULL,
    PRIMARY KEY (event_id, student_id),
    INDEX (student_id),
    FOREIGN KEY (event_id)
        REFERENCES grade_event (event_id),
    FOREIGN KEY (student_id)
        REFERENCES student (student_id)
) ENGINE = InnoDB;

This contains one new feature - an INDEX on the student_id field to permit rapid searching.

Inserting Data

There are two main ways to insert data into a table

- INSERT INTO table-name VALUES ( values )
- LOAD DATA
Queries

The most fundamental database task is querying the database. For this purpose the most important statement is the `SELECT` statement, which can be extremely simple or very complicated due to its many optional parts.

```
SELECT columns
FROM tables
WHERE conditions
GROUP BY group columns
ORDER BY sort columns
HAVING more conditions
LIMIT number
```

Single table selections

We have already covered the fundamentals of single-table selection without aggregation, including:
- Selecting, renaming, re-ordering, transforming and combining columns
- Selecting only those rows that meet certain conditions
- Ordering and limiting the output rows

Zero table selections

In MySQL you can make selections that simply compute results “on-the-fly” and therefore you do not need the `FROM` clause.

```
mysql> SELECT NOW();
+---------------------+
<table>
<thead>
<tr>
<th>NOW()</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006-08-03 11:47:28</td>
</tr>
</tbody>
</table>
+---------------------+
1 row in set (0.00 sec)
```

```
mysql> SELECT POW(2,7);
+----------+
<table>
<thead>
<tr>
<th>POW(2,7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
</tr>
</tbody>
</table>
+----------+
1 row in set (0.00 sec)
```

Multiple table selections

The real power, and hence complexity, of `SELECT` comes from the ability to rapidly extract data from more than one table.

A multiple table `SELECT` statement can become very complex, and (unfortunately) the syntax can often seem somewhat counterintuitive — this is largely because the lack of general programming constructs in SQL.
Combining two tables

The tables `score` and `grade_event` are related by the field `event_id`.

```sql
mysql> select * from grade_event;
+------------+----------+----------+| date | category | event_id | +------------+----------+----------+| 2004-09-03 | Q | 1 | | 2004-09-06 | Q | 2 | ...
...```

```sql
mysql> select * from score;
+------------+----------+-------+| student_id | event_id | score | +------------+----------+-------+| 1 | 1 | 20 | | 3 | 1 | 20 | | 4 | 1 | 18 | ...
```

(GF Royle, N Spadaccini 2006-2010)

Connecting the tables

Suppose we want to determine the scores in the quiz (or test) held on the 9th September 2004. Conceptually this involves

- Consulting the `grade_event` table to find the `event_id` of the event of that date
- Consulting the `score` table to extract the rows that have this `event_id`

This can be performed in a single `SELECT` statement:

```sql
SELECT * FROM grade_event, score
WHERE grade_event.event_id = score.event_id
AND date = '2004-09-09';
```

Output from this query

```sql
<table>
<thead>
<tr>
<th>date</th>
<th>category</th>
<th>event_id</th>
<th>student_id</th>
<th>event_id</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004-09-09</td>
<td>T</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>88</td>
</tr>
<tr>
<td>2004-09-09</td>
<td>T</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>84</td>
</tr>
<tr>
<td>2004-09-09</td>
<td>T</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>69</td>
</tr>
<tr>
<td>2004-09-09</td>
<td>T</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>71</td>
</tr>
<tr>
<td>2004-09-09</td>
<td>T</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>97</td>
</tr>
<tr>
<td>2004-09-09</td>
<td>T</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>83</td>
</tr>
<tr>
<td>2004-09-09</td>
<td>T</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>88</td>
</tr>
<tr>
<td>2004-09-09</td>
<td>T</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>75</td>
</tr>
<tr>
<td>2004-09-09</td>
<td>T</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>83</td>
</tr>
<tr>
<td>2004-09-09</td>
<td>T</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>72</td>
</tr>
<tr>
<td>2004-09-09</td>
<td>T</td>
<td>3</td>
<td>11</td>
<td>3</td>
<td>74</td>
</tr>
</tbody>
</table>
...
```

[The data contained in all three tables can be downloaded from the Downloads section of the CITS3240 website]

Logically speaking

We can analyse this query in parts

- `SELECT * FROM grade_event, score`
  
  On its own, this says to select every combination of rows from `grade_event` and `score` — in fact, to form the Cartesian product `grade_event × score`.

- `WHERE grade_event.event_id = score.event_id`
  
  This now selects only the rows where the `event_id` in the two tables matches — so each row now refers to an individual score in a specific quiz/test.

- `AND date = '2004-09-09'`
  
  And finally this now picks out only the events from this date.
Qualifying the fields

Notice that the conditions included

\[
\text{grade\_event.event\_id = score.event\_id} \\
\text{and} \\
\text{date = '2004-09-09'}
\]

where the two occurrences of `event_id` need to be qualified with the table name because both tables contain a field of that name.

However the `date` field only occurs in one table and so it is unambiguous.

Aliasing the tables

We can also alias the table names temporarily

\[
\text{SELECT S.student\_id, S.score FROM grade\_event G, score S} \\
\text{WHERE} \\
\text{G.event\_id = S.event\_id} \\
\text{AND} \\
\text{G.date = '2004-09-09';}
\]

This is convenient but it also semantically critical in certain specific cases.

When both tables are the same

Suppose we want to find out if two presidents have the same birthday — basically we need to form the cross product of two copies of the same table `president` and then pick out the rows where the birthdates fall on the same day of the year.

We can’t start off with

\[
\text{SELECT * FROM president, president} \\
\text{WHERE} \\
\text{...}
\]

because any subsequent occurrence of a field name, like `birth` would be ambiguous.

A better way

\[
\text{SELECT P1.last\_name, P2.last\_name FROM president P1, president P2} \\
\text{WHERE} \\
\text{DAYOFYEAR(P1.birth) = DAYOFYEAR(P2.birth);} \\
\]

Now `P1` aliases the first copy of the table, while `P2` aliases the second copy of the table, and by comparing which day-of-the-year they were born on we can pick out only those pairs with the same birthday.
Whoops

<table>
<thead>
<tr>
<th>last_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Washington</td>
<td>Washington</td>
</tr>
<tr>
<td>Adams</td>
<td>Adams</td>
</tr>
<tr>
<td>Jefferson</td>
<td>Jefferson</td>
</tr>
<tr>
<td>Madison</td>
<td>Madison</td>
</tr>
<tr>
<td>Monroe</td>
<td>Monroe</td>
</tr>
<tr>
<td>Adams</td>
<td>Adams</td>
</tr>
<tr>
<td>Jackson</td>
<td>Jackson</td>
</tr>
<tr>
<td>Van Buren</td>
<td>Van Buren</td>
</tr>
</tbody>
</table>

We forgot that when the two components of the cross product refer to the same president, then obviously their birthdays will match.

Really finally

We can ensure that the pair is listed only once by adding an extra condition.

```
SELECT P1.last_name, P2.last_name
FROM president P1, president P2
WHERE
  DAYOFYEAR(P1.birth) = DAYOFYEAR(P2.birth)
  AND
  P1.birth <> P2.birth
  AND
  P1.last_name < P2.last_name;
```

Finally...

```
SELECT P1.last_name, P2.last_name
FROM president P1, president P2
WHERE
  DAYOFYEAR(P1.birth) = DAYOFYEAR(P2.birth)
  AND
  P1.birth <> P2.birth;
```

```
<table>
<thead>
<tr>
<th>last_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harding</td>
<td>Polk</td>
</tr>
<tr>
<td>Polk</td>
<td>Harding</td>
</tr>
</tbody>
</table>
```

2 rows in set (0.00 sec)
This lecture focuses on the summary or aggregate features provided in MySQL.

The summary functions are those functions that return a single value from a collection of values — for example, functions that produce counts, totals and averages.

Summary Functions

One of the main uses of a database is to summarize the data it contains, in particular to provide statistical data.

The main summary functions are:

- **COUNT** — to count rows
- **SUM** — to add the values in a column
- **MIN** — to find the minimum value in a column
- **MAX** — to find the maximum value in a column
- **AVG** — to find the average value in a column
- **STD** — to find the standard deviation of the values in a column

Counting students

How many students are in the class for the grade-keeping project?

```
SELECT COUNT(*) FROM student;
```

<table>
<thead>
<tr>
<th>COUNT( *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
</tbody>
</table>

The COUNT function says to count the number of rows that are returned by the SELECT statement — notice that this syntax is not very intuitive.

How many men and women?

If we add a WHERE clause to the statement, then the COUNT will apply only to the selected rows.

```
SELECT COUNT(*) FROM student
WHERE sex = 'M';
```

<table>
<thead>
<tr>
<th>COUNT( *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

```
SELECT COUNT(*) FROM student
WHERE sex = 'F';
```

<table>
<thead>
<tr>
<th>COUNT( *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
</tr>
</tbody>
</table>
With one statement

We can count both men and women in a single statement by using the `GROUP BY` clause.

```
SELECT COUNT(*) FROM student
GROUP BY sex;
```

<table>
<thead>
<tr>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

(GF Royle, N Spadaccini 2006-2010)  Structured Query Language II  6 / 22

But which is which

As it stands, we don’t know which value is associated with which sex!

```
SELECT sex, COUNT(*) FROM student
GROUP BY sex;
```

<table>
<thead>
<tr>
<th>sex</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>15</td>
</tr>
<tr>
<td>M</td>
<td>16</td>
</tr>
</tbody>
</table>

The `GROUP BY` clause says to first group the rows according to the distinct values of the specified attribute(s) and then do the counting.

(GF Royle, N Spadaccini 2006-2010)  Structured Query Language II  7 / 22

Statistical Data

Now let’s try and find statistical data about the quizzes and tests.

```
SELECT event_id, MIN(score), MAX(score), AVG(score)
FROM score
GROUP BY event_id;
```

<table>
<thead>
<tr>
<th>event_id</th>
<th>MIN(score)</th>
<th>MAX(score)</th>
<th>AVG(score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>20</td>
<td>15.1379</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>19</td>
<td>14.1667</td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>97</td>
<td>78.2258</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>20</td>
<td>14.0370</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>20</td>
<td>14.1852</td>
</tr>
<tr>
<td>6</td>
<td>62</td>
<td>100</td>
<td>80.1724</td>
</tr>
</tbody>
</table>

(GF Royle, N Spadaccini 2006-2010)  Structured Query Language II  8 / 22

Counting tests and quizzes

How many of the events were tests and how many were quizzes?

```
SELECT G.category, COUNT(*)
FROM grade_event G
GROUP BY G.category;
```

<table>
<thead>
<tr>
<th>category</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>Q</td>
<td>4</td>
</tr>
</tbody>
</table>

(GF Royle, N Spadaccini 2006-2010)  Structured Query Language II  9 / 22
Separating tests and quizzes

Can we get separate summary data for the quizzes and the tests? To do this we will need to do a multi-table query because `score` does not know what type each event is.

```
SELECT G.category, AVG(S.score)
FROM grade_event G, score S
WHERE G.event_id = S.event_id
GROUP BY G.category;
```

<table>
<thead>
<tr>
<th>category</th>
<th>AVG(S.score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>79.1667</td>
</tr>
<tr>
<td>Q</td>
<td>14.3894</td>
</tr>
</tbody>
</table>

*(GF Royle, N Spadaccini 2006-2010)*

Structured Query Language II

Separating males and females

Now suppose we want to find the averages for each sex separately and for tests and quizzes separately.

```
SELECT G.category, S.sex, AVG(M.score)
FROM grade_event G, student S, score M
WHERE G.event_id = M.event_id
AND M.student_id = S.student_id
GROUP BY G.category, S.sex;
```

```
<table>
<thead>
<tr>
<th>category</th>
<th>sex</th>
<th>AVG(M.score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>77.5862</td>
</tr>
<tr>
<td>T</td>
<td>M</td>
<td>80.6452</td>
</tr>
<tr>
<td>Q</td>
<td>F</td>
<td>14.6981</td>
</tr>
<tr>
<td>Q</td>
<td>M</td>
<td>14.1167</td>
</tr>
</tbody>
</table>
```

*(GF Royle, N Spadaccini 2006-2010)*

Structured Query Language II

Super-aggregate

```
SELECT G.category, S.sex, AVG(M.score)
FROM grade_event G, student S, score M
WHERE G.event_id = M.event_id
AND M.student_id = S.student_id
GROUP BY G.category, S.sex
WITH ROLLUP;
```

```
<table>
<thead>
<tr>
<th>category</th>
<th>sex</th>
<th>AVG(M.score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>F</td>
<td>14.6981</td>
</tr>
<tr>
<td>Q</td>
<td>M</td>
<td>14.1167</td>
</tr>
<tr>
<td>Q</td>
<td>NULL</td>
<td>14.3894</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>77.5862</td>
</tr>
<tr>
<td>T</td>
<td>M</td>
<td>80.6452</td>
</tr>
<tr>
<td>T</td>
<td>NULL</td>
<td>79.1667</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>36.8555</td>
</tr>
</tbody>
</table>
```

*(GF Royle, N Spadaccini 2006-2010)*

Structured Query Language II

What rollup does

The `ROLLUP` clause generates “summaries of summaries” that are inserted at appropriate places in the table.

The `GROUP BY` clauses caused the data to summarised according to the four groups $(Q,F)$, $(Q,M)$, $(T,F)$, $(T,M)$.

Rollup causes these groups to be further grouped together into $(Q, both)$ and $(T, both)$ and then finally combined into a single group.

The fields where multiple values have been counted together are displayed in the result set by using `NULL` for that field.
### Adding the names

At the end of semester, the lecturer needs to know how many marks each *person* got in their quizzes and tests.

```sql
SELECT S.name, G.category, COUNT(*), SUM(M.score)
FROM grade_event G, student S, score M
WHERE G.event_id = M.event_id AND S.student_id = M.student_id
GROUP BY S.name, G.category
WITH ROLLUP;
```

#### The output

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
<th>COUNT(*)</th>
<th>SUM(M.score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abby</td>
<td>Q</td>
<td>4</td>
<td>63</td>
</tr>
<tr>
<td>Abby</td>
<td>T</td>
<td>2</td>
<td>194</td>
</tr>
<tr>
<td>Abby</td>
<td>NULL</td>
<td>6</td>
<td>257</td>
</tr>
<tr>
<td>Aubrey</td>
<td>Q</td>
<td>4</td>
<td>58</td>
</tr>
<tr>
<td>Aubrey</td>
<td>T</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>Aubrey</td>
<td>NULL</td>
<td>6</td>
<td>195</td>
</tr>
<tr>
<td>Avery</td>
<td>Q</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>Avery</td>
<td>T</td>
<td>2</td>
<td>138</td>
</tr>
<tr>
<td>Avery</td>
<td>NULL</td>
<td>5</td>
<td>178</td>
</tr>
<tr>
<td>Becca</td>
<td>Q</td>
<td>4</td>
<td>60</td>
</tr>
<tr>
<td>Becca</td>
<td>T</td>
<td>2</td>
<td>176</td>
</tr>
</tbody>
</table>

### Filtering on aggregate values

Suppose we want to find the student who got the highest average quiz mark.

```sql
SELECT S.name, COUNT(*), AVG(M.score)
FROM grade_event G, student S, score M
WHERE G.category = 'Q'
AND G.event_id = M.event_id
AND S.student_id = M.student_id
GROUP BY S.name
ORDER BY AVG(M.score) DESC;
```

#### The output

<table>
<thead>
<tr>
<th>name</th>
<th>COUNT(*)</th>
<th>AVG(M.score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Megan</td>
<td>3</td>
<td>17.3333</td>
</tr>
<tr>
<td>Gabrielle</td>
<td>3</td>
<td>17.0000</td>
</tr>
<tr>
<td>Michael</td>
<td>4</td>
<td>16.7500</td>
</tr>
<tr>
<td>Teddy</td>
<td>4</td>
<td>16.2500</td>
</tr>
</tbody>
</table>

### Using **HAVING**

But the quiz-prize can only go to a student who sat all of the quizzes.

```sql
SELECT S.name, COUNT(*), AVG(M.score)
FROM grade_event G, student S, score M
WHERE G.category = 'Q'
AND G.event_id = M.event_id
AND S.student_id = M.student_id
GROUP BY S.name
HAVING COUNT(*) = 4
ORDER BY AVG(M.score) DESC;
```

#### The output

<table>
<thead>
<tr>
<th>name</th>
<th>COUNT(*)</th>
<th>AVG(M.score)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Michael</td>
<td>4</td>
<td>16.7500</td>
</tr>
<tr>
<td>Teddy</td>
<td>4</td>
<td>16.2500</td>
</tr>
</tbody>
</table>
Summary

The **HAVING** clause behaves exactly like a **WHERE** clause except that it operates on the *summarized* data, so the whole process is as follows:

- The named columns are extracted from the Cartesian product of all the tables listed in the **FROM** clause.
- All of these rows are then filtered according to the **WHERE** clause.
- The filtered rows are then grouped together according to the **GROUP BY** clause.
- The *aggregate* functions are applied to the rows in each *group*.
- The resulting rows are then *filtered* by the **HAVING** clause.
- The filtered, aggregated rows are then *ordered* by the **ORDER BY** clause.

Using **DISTINCT**

In order to count the number of different states from which the presidents come, we can use

```sql
SELECT COUNT (DISTINCT state) FROM president;
```

<table>
<thead>
<tr>
<th>COUNT(DISTINCT state)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

The **DISTINCT** keyword eliminates the duplicate values before counting.

### Tables with **NULL** values

Consider a table with the following data

```sql
mysql> select * from test;
```

<table>
<thead>
<tr>
<th>mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

What is the number of rows, the sum of the rows and the average value for the single field?

```sql
mysql> SELECT COUNT(*), SUM(mark), AVG(mark) FROM test;
```

<table>
<thead>
<tr>
<th>COUNT(*)</th>
<th>SUM(mark)</th>
<th>AVG(mark)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>53</td>
<td>13.2500</td>
</tr>
</tbody>
</table>

Notice that **AVG** is not equal to **SUM / COUNT**.

Sometimes **NULL** counts, sometimes not!

```sql
mysql> SELECT COUNT(*), SUM(mark), AVG(mark) FROM test;
```

<table>
<thead>
<tr>
<th>COUNT(*)</th>
<th>SUM(mark)</th>
<th>AVG(mark)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>53</td>
<td>13.2500</td>
</tr>
</tbody>
</table>

Notice that **AVG** is not equal to **SUM / COUNT**.
Learning how to summarize

Learning how to use the summary functions requires a lot of practice because you just have to learn the somewhat strange syntax, and the error messages produced by MySQL are not very informative.

For example, to someone used to a normal programming language it seems very strange to type

```
SELECT COUNT(*) FROM president WHERE death IS NULL;
```

rather than

```
COUNT (SELECT * FROM president WHERE death is NULL);
```

(GF Royle, N Spadaccini 2006-2010)

This lecture covers the process of implementing an ER diagram as an actual relational database.

This involves converting the various entity sets and relationship sets into tables (i.e. relations) in such a way that the DBMS can check and enforce the participation and key constraints.

This lecture could equally well be called “From ER diagrams to relational tables”.

Entity Sets

Implementing entity sets is usually straightforward.

Each attribute of the entity set becomes a column of a relational table, and the only real choice needed is to select a type for that column.

- staff-id
- name
- rank

Lecturer
**Basic Syntax**

This entity set will be represented by a single 3-column table called Lecturer, with one column each for the staff number, the name and the rank of the lecturer.

The basic syntax for creating the table is simple:

```sql
CREATE TABLE lecturer (
    staff_id INT,
    name VARCHAR(20),
    rank VARCHAR(20)
);
```

**The staff ID**

We have chosen the numeric type INT for the staff-id — this choice however will depend on the business logic for that particular organization.

Prior to 2005, UWA student “numbers” were allowed to contain the non-numeric character X so that something like

```
9712303X
```

would be a valid student number.

Even today, most staff numbers at UWA start with zeros, such as

```
00097382
```

In both of these situations, we would need to use one of the string types.

**Specifying the key**

When creating a table, it is possible (and highly recommended) to specify the key for the table.

```sql
CREATE TABLE lecturer (
    staff_id INT,
    name VARCHAR(20),
    rank VARCHAR(20),
    PRIMARY KEY (staff_id)
);
```

The main consequence of this specification is that the DBMS will ensure that this relation never contains duplicate rows.

**The special value NULL**

Sometimes the rows in a relational database table can only be partly specified.

For example, in the President database table we cannot enter the date-of-death for the most recent US presidents. In this situation there is a special value NULL which essentially means that “this value is not yet specified”.

However there are many situations where a field should not be permitted to have a NULL value, and in particular a key value can never be NULL.

Declaring a field to be a primary key means that the database enforces the restriction that it is both unique and not NULL.
Values that cannot be **NULL**

In many other situations, the business logic dictates that a certain attribute cannot be **NULL** and so there is a way of specifying this restriction separately from being a key.

CREATE TABLE lecturer (
  staff_id INT,
  PRIMARY KEY (staff_id),
  name VARCHAR(20) NOT NULL,
  rank VARCHAR(20)
);

**Any old numbers**

Frequently the number used for a primary key has no meaning other than to uniquely identify that row — provided it is different from the other numbers we don’t actually care what it is.

We can get the DBMS to assign the numbers for us automatically by modifying the table appropriately.

CREATE TABLE lecturer (
  staff_id INT AUTO_INCREMENT,
  PRIMARY KEY (staff_id),
  name VARCHAR(20) NOT NULL,
  rank VARCHAR(20)
);

**Using auto-increment**

If a column is auto-increment then specifying its value is *optional* when inserting data into the table.

- If the number is specified (and valid) then it is used
  
  ```sql
  INSERT INTO lecturer
  VALUES(1454,"John Smith","Associate Professor");
  ```

- If the number is not specified then a unique number is generated.
  
  ```sql
  INSERT INTO lecturer
  VALUES(NULL,"Gill Lee","Professor");
  ```

(Yes, it does seem odd to specify **NULL** for a column that is *explicitly not* permitted to be **NULL**.)

**The rank**

So far we have specified a string type, namely `VARCHAR(20)` for the *rank* of the lecturer.

Although this would work, it is unnecessarily error-prone because we know in advance that there are only five different ranks — Associate Lecturer, Lecturer, Senior Lecturer, Associate Professor and Professor. Ideally we would like a type that could only take those values, so that a typing mistake like

```sql
INSERT INTO lecturer
VALUES(1454,"John Smith","Asscoiate Porfessor");
```

could be automatically detected.
Enumerated Types

SQL has an *enumerated type* facility that allows the values that a field can take to be restricted to values from a *specified list*.

```sql
CREATE TABLE lecturer (  
    staff_id INT AUTO_INCREMENT,  
    PRIMARY KEY (staff_id),  
    name VARCHAR(20) NOT NULL,  
    rank ENUM('AL', 'L', 'SL', 'AP', 'P')  
);
```

This specifies that the value of `rank` can only be one of the five possible strings AL, L, SL, AP or P.

Encoding Relationship Sets

The simplest “plain vanilla” situation is a binary relationship with no constraints.

![ER Diagram](attachment:image.png)

This is a variant of the ER diagram previously discussed which relates lecturers to the departments in which they work.

Sample entity set data

```sql
mysql> select * from lecturer;
+----------+-----------------+------+  
<table>
<thead>
<tr>
<th>staff_id</th>
<th>name</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gill Lee</td>
<td>P</td>
</tr>
<tr>
<td>2</td>
<td>John Smith</td>
<td>AP</td>
</tr>
<tr>
<td>3</td>
<td>Adam Zoot</td>
<td>AL</td>
</tr>
<tr>
<td>4</td>
<td>Bill Jackson</td>
<td>L</td>
</tr>
<tr>
<td>5</td>
<td>Jessica Ormerod</td>
<td>SL</td>
</tr>
</tbody>
</table>
+----------+-----------------+------+
5 rows in set (0.00 sec)
```

```sql
mysql> select * from department;
+---------+------------------+-------+  
| dept_no | name | phone |  
|---------|------------------+-------|  
| 1       | Computer Science | 2716  |  
| 2       | Mathematics      | 1574  |  
| 3       | Physics          | 2746  |  
| 4       | Chemistry        | 9716  |  
| 5       | Geology          | 1458  |  
+---------+------------------+-------+
5 rows in set (0.00 sec)
```
Sample relationship set data

mysql> select * from works;
+----------+---------+------------+
| staff_id | dept_no | percentage |
|----------+---------+------------|
| 1        | 1       | 20         |
| 1        | 2       | 80         |
| 2        | 3       | 100        |
| 3        | 3       | 50         |
| 3        | 4       | 50         |
| 4        | 1       | 30         |
| 4        | 5       | 70         |
| 5        | 1       | 100        |
+----------+---------+------------+
8 rows in set (0.00 sec)

Each record (row, tuple) gives information about how a particular lecturer relates to a particular department.

Who works in physics?

How do we use this information to find out who works in Physics?

We need to

- Look in the department table to find out the department number of Physics.
- Look in the works table to find out staff numbers of the people working in departments with that particular number.
- Look in the lecturer table to find the names of the lecturers with those staff numbers.

A simpler version

First we'll just find out the staff numbers of those who work in Physics.

mysql> SELECT staff_id FROM works, department WHERE
   -> works.dept_no = department.dept_no
   -> AND
   -> department.name = "Physics";
+----------+
<table>
<thead>
<tr>
<th>staff_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
+----------+
2 rows in set (0.00 sec)

What does this mean?

The FROM clause implies that we are (notionally) considering every possible combination of rows from the two tables works and department.
Most of these rows are useless...

Of course most of these combinations of rows are useless because they contain things that are *not related* to each other.

The conditions, specified in the `WHERE` clause first specify the row-combinations that make sense in that are both talking about the same department.

```
WHERE works.dept_no = department.dept_no
```

<table>
<thead>
<tr>
<th>staff_id</th>
<th>dept_no</th>
<th>percentage</th>
<th>dept_no</th>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
<td>1</td>
<td>Computer Science</td>
<td>2716</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>30</td>
<td>1</td>
<td>Computer Science</td>
<td>2716</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>Computer Science</td>
<td>2716</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>80</td>
<td>2</td>
<td>Mathematics</td>
<td>1574</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>70</td>
<td>3</td>
<td>Physics</td>
<td>2746</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>70</td>
<td>5</td>
<td>Geology</td>
<td>1458</td>
</tr>
</tbody>
</table>

... until we select just those we want...

Then we add the additional condition that we want to know only about Physics.

```
AND
department.name = "Physics";
```

<table>
<thead>
<tr>
<th>staff_id</th>
<th>dept_no</th>
<th>percentage</th>
<th>dept_no</th>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>100</td>
<td>3</td>
<td>Physics</td>
<td>2746</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>50</td>
<td>3</td>
<td>Physics</td>
<td>2746</td>
</tr>
</tbody>
</table>

2 rows in set (0.03 sec)

But who are they?

Of course a user is immediately likely to ask *who* the lecturers are, not just what their staff numbers are, which means that we have to involve the `lecturer` table as well.

```
mysql> SELECT lecturer.name FROM lecturer, works, department
    -> WHERE
    -> lecturer.staff_id = works.staff_id
    -> AND
    -> works.dept_no = department.dept_no
    -> AND
    -> department.name = "Physics";
```

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Smith</td>
</tr>
<tr>
<td>Adam Zoot</td>
</tr>
</tbody>
</table>

2 rows in set (0.01 sec)
How are queries built?

A fundamental reason for the success of SQL and RDBMS is that queries like these can be expressed formally using either relational algebra or relational calculus and analysed to produce a reasonable evaluation strategy.

This allows the user to create complex queries involving multiple tables and have the DBMS work out the best way to actually perform that query.

This lecture continues the discussion of encoding relationship sets into tables, but this time considering how things work in the presence of participation and key constraints.

When discussing the encoding of relationship sets, it is useful to distinguish relationships that are
- One-to-One
- One-to-Many
- Many-to-Many

We will use examples based on the book "Fundamentals of Database Management Systems" by Mark. L. Gillenson.

These examples are based on a hypothetical hardware supplier General Hardware that sells hardware products in quantity to retail outlets.
Relationship Sets

One-to-Many

This ER diagram indicates that every customer of General Hardware has a unique salesperson responsible for that account.

Sample entity sets

<table>
<thead>
<tr>
<th>staffId</th>
<th>name</th>
<th>hireYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>137</td>
<td>Baker</td>
<td>1995</td>
</tr>
<tr>
<td>186</td>
<td>Adams</td>
<td>2001</td>
</tr>
<tr>
<td>204</td>
<td>Dickens</td>
<td>1998</td>
</tr>
<tr>
<td>361</td>
<td>Carlyle</td>
<td>2004</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>customerId</th>
<th>name</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1121</td>
<td>Bunnings</td>
<td>Subiaco</td>
</tr>
<tr>
<td>1122</td>
<td>Bunnings</td>
<td>Claremont</td>
</tr>
<tr>
<td>1211</td>
<td>Mitre 10</td>
<td>Myaree</td>
</tr>
<tr>
<td>1244</td>
<td>Mitre 10</td>
<td>Joondalup</td>
</tr>
<tr>
<td>1345</td>
<td>Joe's Hardware</td>
<td>Nedlands</td>
</tr>
<tr>
<td>1399</td>
<td>NailsRUs</td>
<td>Jolimont</td>
</tr>
</tbody>
</table>

What is the key?

In the works relation, we needed the combination of the staff number and department number to form a key because the relationship was many-to-many.

However here the relationship is one-to-many and so each customerId can only appear once in the table, and so we can use customerId as the key.

But what this suggests to us is that this table is unnecessary because we could incorporate this information into the Customer relation.

This permits us to eliminate one table with the resulting simplification of queries.

The relationship set

Now suppose that Baker is the account manager for both Bunnings outlets, Adams for both Mitre 10 outlets, Dickens for Joe's Hardware and Carlyle for NailsRUs.

Then we could have a separate table SellsTo, similar to the works table from last lecture.

<table>
<thead>
<tr>
<th>staffId</th>
<th>customerId</th>
</tr>
</thead>
<tbody>
<tr>
<td>137</td>
<td>1121</td>
</tr>
<tr>
<td>137</td>
<td>1122</td>
</tr>
<tr>
<td>186</td>
<td>1211</td>
</tr>
<tr>
<td>186</td>
<td>1244</td>
</tr>
<tr>
<td>204</td>
<td>1345</td>
</tr>
<tr>
<td>361</td>
<td>1399</td>
</tr>
</tbody>
</table>
The one-to-many approach

CREATE TABLE customer (
    customerId INT,
    PRIMARY KEY (customerId),
    name VARCHAR(20),
    address VARCHAR(40),
    accountMgr INT
);

We incorporate an additional field into `customer` which contains the `staffId` for the salesperson responsible for that company's account. This field is called a foreign key because it is the key value for a different table.

Participation Constraints

The advantage of this second approach to the one-to-many relationship is that it allows us to enforce the participation constraints that every customer must have an account manager — we simply have to add NOT NULL to that field.

CREATE TABLE customer (
    customerId INT,
    PRIMARY KEY (customerId),
    name VARCHAR(20),
    address VARCHAR(40),
    accountMgr INT NOT NULL
);

The new entity set

<table>
<thead>
<tr>
<th>customerId</th>
<th>name</th>
<th>address</th>
<th>accountMgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1121</td>
<td>Bunnings</td>
<td>Subiaco</td>
<td>137</td>
</tr>
<tr>
<td>1122</td>
<td>Bunnings</td>
<td>Claremont</td>
<td>137</td>
</tr>
<tr>
<td>1211</td>
<td>Mitre 10</td>
<td>Myaree</td>
<td>186</td>
</tr>
<tr>
<td>1244</td>
<td>Mitre 10</td>
<td>Joondalup</td>
<td>186</td>
</tr>
<tr>
<td>1345</td>
<td>Joe's Hardware</td>
<td>Nedlands</td>
<td>204</td>
</tr>
<tr>
<td>1399</td>
<td>NailsRUs</td>
<td>Jolimont</td>
<td>361</td>
</tr>
</tbody>
</table>

One-to-One

Suppose that every salesperson has an office and every office houses at most one salesperson.
### Choices

We have a number of different choices
- A single table combining the salesperson and office information
- Incorporating the office information as a foreign key in the salesperson entity set
- Incorporating the salesperson information as a foreign key in the office entity set

What are the pros and cons of these options?

### Is there total participation?

The crucial factor here is whether there is total participation of the entities in the relationship set — the diagram does not show total participation.

If the relationship is one-to-one with total participation, so that every salesperson has an office and every office has a salesperson, then we can simply incorporate all of the “office” information into the SalesPerson relation.

In this situation, there would be two candidate keys, namely the staffId and the roomId.

### The table

```sql
CREATE TABLE salesPerson (
    staffId INT,
    PRIMARY KEY (staffId),
    name VARCHAR(20),
    hireYear DATE,
    roomId INT,
    phone INT,
    location VARCHAR(20)
);
```

### A bad design

In general, this is a bad design because the relationship is not intrinsically a one-to-one relationship with total participation.

What happens if a salesperson resigns? Then their office becomes vacant, but there is no way of recording this with the current structure.

If a relationship really is 1-1 with total participation, then it could (would?) be modelled as the attributes of an entity set in the ER diagram, rather than as a relationship set.
Other choices

The other two choices are to incorporate the office number as a foreign key in the SalesPerson entity set, or the incorporate the staff number as a foreign key in the Office entity set.

The correct decision here follows from the business logic — we should include the office number as a NOT NULL foreign key in the SalesPerson entity set because each salesperson must be assigned an office.

Empty offices are then simply entries in the Office table that do not appear in the SalesPerson table, and cause no problems.

Referential Integrity

Although we know that these fields are meant to refer to the other tables, the DBMS does not.

This means that there is nothing stopping us from entering data into the works table that has a staff-id that does not actually correspond to any actual staff member, or a dept-no that does not correspond to a department.

INSERT INTO works VALUES(-1,12,50);

This is called a violation of referential integrity — in other words, the references from one table to another are corrupt.

Foreign Keys

Consider again the works table which relates lecturers and departments.

CREATE TABLE works (  
  staff_id INT,  
  dept_no INT,  
  percentage INT,  
  PRIMARY KEY (staff_id, dept_no)  
);  
The fields staff_id and dept_no are foreign keys that refer to the fields of the same name in the tables lecturer and department.

Foreign key support

SQL permits the user to declare that a certain field is a foreign key that references another table.

CREATE TABLE works (  
  staff_id INT,  
  dept_no INT,  
  percentage INT,  
  PRIMARY KEY (staff_id, dept_no),  
  FOREIGN KEY (staff_id) REFERENCES lecturer (staff_id),  
  FOREIGN KEY (dept_no) REFERENCES department (dept_no)  
);
Foreign key support in MySQL

In MySQL, only the InnoDB storage engine provides foreign key support and this must be declared when the table is created.

```sql
CREATE TABLE works (
    staff_id INT,
    dept_no INT,
    percentage INT,
    PRIMARY KEY (staff_id, dept_no),
    FOREIGN KEY (staff_id) REFERENCES lecturer (staff_id),
    FOREIGN KEY (dept_no) REFERENCES department (dept_no)
) ENGINE = InnoDB;
```

Referential Integrity

Once the DBMS knows that the fields really are foreign keys it can maintain referential integrity by forbidding operations that would violate referential integrity.

```sql
mysql> INSERT INTO works
        -> VALUES(6,1,100);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails (`university/works`, CONSTRAINT 'works_ibfk_1' FOREIGN KEY ('staff_id') REFERENCES 'lecturer' ('staff_id'))
```

This operation is forbidden because the value “6” is not a legitimate staff number according to the lecturer table.

Deletion

What should happen to a row containing a foreign key if the thing that it refers to is deleted?

Suppose for example that Adam Zoot leaves the university and we want to remove his records from the database.

```sql
mysql> DELETE FROM lecturer WHERE name = "Adam Zoot";
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails (`university/works`, CONSTRAINT 'works_ibfk_1' FOREIGN KEY ('staff_id') REFERENCES 'lecturer' ('staff_id'))
```

By default this action is **forbidden** because it would leave the works table with a dangling reference — a foreign key that referred to a non-existent lecturer.

Cascade

In this situation, it is clear that we want to delete the associated records in the works table because they are no longer relevant.

```sql
CREATE TABLE works (
    staff_id INT,
    dept_no INT,
    percentage INT,
    PRIMARY KEY (staff_id, dept_no),
    FOREIGN KEY (staff_id) REFERENCES lecturer (staff_id)
    ON DELETE CASCADE,
    FOREIGN KEY (dept_no) REFERENCES department (dept_no)
) ENGINE = InnoDB;
```

The **ON DELETE CASCADE** says that if the **parent** row is deleted then the consequences of this deletion should **cascade** through to this table.
Another option for a foreign key relationship is

ON DELETE SET NULL

which will cause the corresponding columns to be set to NULL rather than deleted entirely.

This option would be appropriate for example in the Customer relation from the General Hardware database where the foreign key accountMgr is set to NULL if the salesperson leaves.

This lecture

We continue our coverage of the fundamentals of SQL/MySQL with nested queries, or subqueries.

A nested query is a query that involves another query as one of its component parts.

```
SELECT * FROM score
WHERE event_id =
    (SELECT event_id FROM grade_event
     WHERE date = '2004-09-09');
```

Here we have a simple query that involves two SELECT statements.
### Nested queries

The *inner query* produces just the `event_id` of the test/quiz on 9th September 2004.

```sql
SELECT event_id FROM grade_event
    WHERE date = '2004-09-09';
+----------+
<table>
<thead>
<tr>
<th>event_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
```

Thus the *outer query* is simply equivalent to

```sql
SELECT * FROM score
WHERE event_id = 3;
```

(GF Royle, N Spadaccini 2006-2010)

---

### Types of subquery

How a subquery can be manipulated depends on the type of results that it produces:

- A *scalar* subquery produces a single value (that is, a table with one row and one column) as a result
- A *column* subquery produces a single column as a result
- A *row* subquery produces a single row as a result
- A *table* subquery produces an entire table as a result

There are special operators that can be used with each of these types of query.

---

### Scalar Subqueries

A scalar subquery produces a one-row, one-column table that is treated as a single value (that is, a scalar), and can be used essentially anywhere that a single value can be used – for example, you can make comparisons with `<`, `>`, `=`, `<>` and so on.

Sometimes a scalar subquery is just used to find an unknown ID value from a different table:

```sql
SELECT * FROM city
WHERE countrycode = (SELECT code
                    FROM country
                    WHERE name = 'Australia');
```

Notice the use of `T.*` to get all of the fields from just the `city` part of the joined table.

(GF Royle, N Spadaccini 2006-2010)

---

### Equivalent to a join

A subquery like this is just an alternative to a join.

```sql
SELECT T.* FROM city T, country C
WHERE T.countrycode = C.code
AND C.name = 'Australia';
```

(GF Royle, N Spadaccini 2006-2010)
**Scalar Subqueries**

### Maximum population

More significant uses that cannot be replaced by simple joins occur when a selection is based on the result of an aggregate operation.

```sql
SELECT name, population
FROM country
WHERE population = (SELECT MAX(population)
                      FROM country);
```

<table>
<thead>
<tr>
<th>name</th>
<th>population</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>1277558000</td>
</tr>
</tbody>
</table>

*(GF Royle, N Spadaccini 2006-2010)*

### User Variables

You can also do such a query in two steps if you wish, because MySQL allows the user to define *user variables*. A user variable must begin with the @ character and can be created within a `SELECT` statement.

```sql
SELECT @maxpop := MAX(population) FROM country;
SELECT name, population
FROM country
WHERE population = @maxpop;
```

This uses a variable `@maxpop` to store the maximum population value to be used in the second query.

*(GF Royle, N Spadaccini 2006-2010)*

### Relative comparisons

Which countries are between Germany and Indonesia according to population?

```sql
SELECT name, population FROM country
WHERE population <= (SELECT population
                      FROM country
                      WHERE name = 'Indonesia')
AND population >= (SELECT population
                    FROM country
                    WHERE name = 'Germany')
ORDER BY population DESC;
```

### Which countries have above average population density?

```sql
SELECT name, population/surfacearea as density
FROM country
WHERE population/surfacearea >
     (SELECT AVG(population/surfacearea) FROM country)
ORDER BY density DESC;
```

<table>
<thead>
<tr>
<th>name</th>
<th>density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monaco</td>
<td>22666.666667</td>
</tr>
<tr>
<td>Hong Kong</td>
<td>6308.837209</td>
</tr>
<tr>
<td>Singapore</td>
<td>5771.844660</td>
</tr>
<tr>
<td>Gibraltar</td>
<td>4166.666667</td>
</tr>
</tbody>
</table>

*(GF Royle, N Spadaccini 2006-2010)*
**IN and NOT IN**

If a subquery returns more than one value, then it can be treated as a set of values and the outer query can test whether values are IN or NOT IN this set.

For example, we can find out which sailors in the `sailor` table have not reserved any boats.

```sql
SELECT * FROM sailor
WHERE sid NOT IN (SELECT sid
                   FROM reserves);
```

Further examples

Which students are not enrolled in any classes?

```sql
SELECT S.sname
FROM student S
WHERE S.snum NOT IN
     (SELECT snum FROM enrolled);
```

This uses the same idea as the previous example.

**Most populous country in each region**

Suppose we want to find the most heavily-populated country in each of the world’s regions. We know how to find the population easily enough.

```sql
SELECT C.region, MAX(C.population) as maxpop
FROM country C
GROUP BY region;
```

This tells us that, for example, that some country in the Baltic Countries has a population of 3.7 million, but does not tell us which country.

**Incorrect approach**

An obvious, but unfortunately incorrect approach would be to try

```sql
SELECT C.region, C.name, MAX(C.population) as maxpop
FROM country C
GROUP BY region;
```

This uses the same idea as the previous example.
Multiple-valued subqueries

Why is this incorrect?

This is incorrect because after the “grouping by region”, the group for the Baltic countries looks like this:

| Baltic Countries | Latvia | 2424200 |
| Baltic Countries | Estonia | 1439200 |
| Baltic Countries | Lithuania | 3698500 |

The presence of the aggregate function \textit{MAX} indicates that each group should be aggregated into a single row.

There is no ambiguity in the value of \textit{C.region} because it is constant over the group, and nor is there any ambiguity in the value \textit{MAX(population)}. However the value \textit{name} is an \textit{individual} value, and not a property of the group. Hence SQL simply takes the first value — possibly a warning should be issued if a column is named in the \textit{SELECT} but not used in the \textit{GROUP BY} clause.

Correct Approach 1

One correct approach would be to use a subquery that first determines the maximum population for each region, and then an outer query that “attaches” the correct country name to that pair.

\begin{verbatim}
SELECT C.region, C.name, C.population
FROM country C
WHERE (C.region, C.population) IN
  (SELECT C2.region, MAX(C2.population)
   FROM country C2 GROUP BY region);
\end{verbatim}

The inner is run once and produces a list of (region, maxpop) pairs, while the outer query essentially checks the values for each country to see if they match a row in this table.

Correct Approach 2

The second correct approach uses a \textit{correlated subquery} which is where the subquery refers to a table from the \textit{outer query}.

\begin{verbatim}
SELECT C.region, C.name, C.population
FROM country C
WHERE C.population =
  (SELECT MAX(population)
   FROM country C2
   WHERE C2.region = C.region);
\end{verbatim}

This subquery is called \textit{correlated} because it involves a value (\textit{C.region}) that comes from a table in the outer query.

Visualizing correlated subqueries

Conceptually we imagine a correlated subquery as being run once for each row of the table that it refers to.

For the query on the previous slide, we imagine \textit{C} being set equal to each row of the table \textit{country} in turn:

| Afghanistan | Southern and Central Asia | 22720000 |
| Netherlands | Western Europe | 15864000 |
| Netherlands Antilles | Caribbean | 217000 |

Then each time through, the maximum population of the region \textit{C.region} is computed and compared to the actual population of \textit{C}.
**Example schema**

We will use this schema from the textbook:

- **Suppliers** (sid: integer, sname: string, address: string)
- **Parts** (pid: integer, pname: string, colour: string)
- **Catalogue** (sid: integer, pid: integer, price: real)

Here, **suppliers** and **parts** are entity sets that are related by the relationship set (which has the name **catalogue**) indicating which suppliers sell which parts.

The **price** variable is a relationship attribute of this relationship set, because the price is an attribute of the **combination** of a particular seller and a particular part.

**EXISTS and NOT EXISTS**

The clauses **EXISTS** and **NOT EXISTS** can be used in conjunction with a subquery simply to see if that subquery returns any results. This kind of construct can be useful when answering “all” or “none” questions in relational tables. For example, consider the question:

**Which suppliers do not supply any parts?**

SELECT S.sname
FROM suppliers S
WHERE NOT EXISTS (SELECT * FROM catalogue C WHERE C.sid = S.sid);

**Who supplies every part**

To find out who supplies **every** part in the catalogue requires a bit of linguistic contortion.

First let’s find out which parts a supplier with id sid does **not** supply — notice that this is not a fully-formed query because sid is not qualified.

SELECT P.pid FROM parts P
WHERE NOT EXISTS (SELECT * FROM catalogue C WHERE C.pid = P.pid AND C.sid = sid);

Now a supplier supplies **every** part if we **cannot find** a part that the supplier **does not supply**.

SELECT S.sname
FROM suppliers S
WHERE NOT EXISTS (SELECT P.pid
FROM parts P
WHERE NOT EXISTS (SELECT * FROM catalogue C WHERE C.pid = P.pid AND C.sid = S.sid));
Unique suppliers

*Find the names of the parts supplied only by Acme Widget Suppliers.*

By now the general principle is becoming apparent — find the parts that are supplied by Acme Widget Suppliers, but for which there are no different suppliers.

```sql
SELECT P.pname
FROM parts P, catalogue C, suppliers S
WHERE P.pid = C.pid
AND C.sid = S.sid
AND S.sname = 'Acme Widget Suppliers'
AND NOT EXISTS (SELECT *
FROM catalogue C2
WHERE C2.sid <> S.sid
AND C2.pid = P.pid);
```

This correlated subquery uses two of the “outer” tables.

---

This lecture

This lecture covers relational algebra which is the formal language underlying the manipulation of relations.

We follow the notation from Chapter 4 of Ramakrishnan & Gehrke.

Relational algebra is a procedural language that allows us to describe operations on relations in a formal and mathematically precise.

An expression in relational algebra describes a sequence of operations that can be applied to a relation and which produces a relation as a result.

The primary operations of the relational algebra are projection, selection and joins.
Projection

Suppose we have a relation

\[ R \subseteq C_1 \times C_2 \times \cdots \times C_n, \]

where \( C_1, C_2, \ldots, C_n \) are the columns of the relation.

If \( S \) is a subset of the columns then \( \pi_S(R) \) is the relation obtained from \( R \) by deleting all the columns not in \( S \), and it is called the projection onto \( S \) of the relation \( R \).

Example Projections

Then \( \pi_{\text{customerId}, \text{name}}(R) \) is

<table>
<thead>
<tr>
<th>customerId</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1121</td>
<td>Bunnings</td>
</tr>
<tr>
<td>1122</td>
<td>Bunnings</td>
</tr>
<tr>
<td>1211</td>
<td>Mitre 10</td>
</tr>
<tr>
<td>1244</td>
<td>Mitre 10</td>
</tr>
<tr>
<td>1345</td>
<td>Joe's Hardware</td>
</tr>
<tr>
<td>1399</td>
<td>NailsRUs</td>
</tr>
</tbody>
</table>

We have projected the relation onto the two named columns, thus obtaining a smaller relation.

Another example projection

If \( R \) is the relation given above, then \( \pi_{\text{name}}(R) \) is the relation

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bunnings</td>
</tr>
<tr>
<td>Mitre 10</td>
</tr>
<tr>
<td>Joe's Hardware</td>
</tr>
<tr>
<td>NailsRUs</td>
</tr>
</tbody>
</table>

As the result of a projection is a relation which is defined to be a set, the output cannot contain any duplicate rows.
Projection and Selection

In SQL

It should be clear that there is a direct relationship between
- the projection operator $\pi$, and
- the SELECT columns FROM statement

In practical terms one difference is that SQL normally does not automatically remove duplicate rows, because this is a relatively expensive operation.

Example Selection

If $R$ is the relation defined above, then

$\sigma_{\text{customerID}<1300}(R)$

is the relation

<table>
<thead>
<tr>
<th>customerId</th>
<th>name</th>
<th>address</th>
<th>accountMgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1121</td>
<td>Bunnings</td>
<td>Subiaco</td>
<td>137</td>
</tr>
<tr>
<td>1122</td>
<td>Bunnings</td>
<td>Claremont</td>
<td>137</td>
</tr>
<tr>
<td>1211</td>
<td>Mitre 10</td>
<td>Myaree</td>
<td>186</td>
</tr>
<tr>
<td>1244</td>
<td>Mitre 10</td>
<td>Joondalup</td>
<td>186</td>
</tr>
</tbody>
</table>

Selection

Projection is an operation that extracts columns from a relation, while selection is the operation that extracts rows from a relation.

If $R$ is a relation and $B$ is a boolean function on the columns of $R$ then

$\sigma_B(R)$

is the relation with the same columns as $R$ and whose rows are the rows of $R$ for which the boolean function evaluates to true.

Equivalently we say that $\sigma_B$ selects the rows from $R$ that satisfy the boolean condition $B$.

Combining selection and projection

As the result of a selection or projection is a relation, we can use it as the “input” for another selection or projection and thus build up complex relational algebra expressions.

$\pi_{\text{accountMgr}}(\sigma_{\text{name}='Bunnings'}(R))$

This compound operation first selects the rows that correspond to “Bunnings” and then projects onto the single column accountMgr — in other words, this relational algebra expression answers the query “What is the staff id of the account manager for Bunnings”.

In SQL

The selection operator in relational algebra corresponds directly to the `WHERE` clause of SQL. Thus we can directly translate the relational algebra expression of the previous slide into an SQL query:

```sql
SELECT accountMgr FROM customer WHERE name = 'Bunnings';
```

Boolean functions

The boolean functions that can be used as the selection condition are combinations using `∧` (for AND) and `∨` (for OR) of terms of the form

```
attribute op constant
```

or

```
attribute1 op attribute2
```

where `op` is a comparison operator in the set

\{<, ≤, =, ≠, ≥, >\}.

Relational Algebra Expressions

A relational algebra expression is recursively defined to be

- A relation, or
- A unary operator applied to a single expression
- A binary operator applied to two expressions

Selection and projection are unary operators because they operate on a single relation.

Binary Operators

Relational algebra permits the use of the standard set operations:

- **Union (\(\cup\))**
  - If \(R\) and \(S\) are union-compatible, then \(R \cup S\) is the set of tuples in either \(R\) or \(S\).

- **Intersection (\(\cap\))**
  - If \(R\) and \(S\) are union-compatible, then \(R \cap S\) is the set of tuples in both \(R\) and \(S\).

- **Set Difference (\(−\))**
  - If \(R\) and \(S\) are union-compatible then \(R − S\) is the set of tuples in \(R\) that are not in \(S\).

- **Cartesian Product (\(×\))**
  - If \(R\) has arity \(r\) and \(S\) has arity \(s\), then \(R \times S\) has arity \(r + s\) and has all tuples whose projection onto the first \(r\) columns is in \(R\) and whose projection onto the last \(s\) columns is in \(S\).
Union in MySQL

Of the first three set operations, MySQL 5.0 only supports `UNION` as a direct command. This can be useful when two subsets of data are being pulled from different sets of tables.

The `UNION` can simply be placed between two quite independent `SELECT` statements as long as both return the same number of columns.

```sql
SELECT birth FROM president
UNION
SELECT death FROM president;
```

---

Joins

The *join* of two relations $R$ and $S$ is one of the most important operations in real databases and is defined as follows.

Suppose that $c$ is a boolean function that may involve the attributes of both $R$ and $S$.

Then

$$R \bowtie_c S$$

is defined to be

$$\sigma_c(R \times S).$$

In other words a *join* is the result of selecting certain rows from the Cartesian product.

---

Equijoin

In almost all the examples we have seen so far, the *join condition* has actually consisted of *equalities* between attributes of $R$ and $S$ — often an equality between a key and a foreign key.

In such a join, the resulting relation will have a number of duplicated columns — in particular any attributes used in the join condition will appear twice.

An *equijoin* of two tables is a join where the join condition is a conjunction of equalities of the form $R.attribute1 = S.attribute2$ with the columns of $S$ that appear in the join condition projected out.

---

Natural Join

The *natural join* of two relations $R$ and $S$ is the equijoin whose join condition involves every column having the same name in $R$ and $S$.

As the join condition can be determined by context, it can simply be omitted and

$$R \bowtie S$$

denotes the natural join of $R$ and $S$. 

Examples from the text

Consider the following conceptual schema taken almost directly from Ramakrishnan & Gehrke that is related to a boat-rental operation.

![Diagram of entity sets]

Queries in relational algebra

Suppose we wish to answer the question

*Which sailors have reserved boat 103?*

One expression that answers this query is

\[ \pi_{\text{sname}}(\sigma_{\text{bid}=103}(\text{reserves} \bowtie \text{sailor})) \]

An *equivalent* expression that also answers this query is

\[ \pi_{\text{sname}}(\sigma_{\text{bid}=103}(\text{reserves} \bowtie \text{sailor})) \]

Sample entity sets

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>45.0</td>
</tr>
<tr>
<td>29</td>
<td>Brutus</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>Andy</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>Horatio</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>Zorba</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>Horatio</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>Art</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>63.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>11-Aug-06</td>
</tr>
<tr>
<td>22</td>
<td>102</td>
<td>02-Aug-06</td>
</tr>
<tr>
<td>31</td>
<td>103</td>
<td>03-Aug-06</td>
</tr>
<tr>
<td>41</td>
<td>104</td>
<td>17-Aug-06</td>
</tr>
<tr>
<td>64</td>
<td>101</td>
<td>18-Aug-06</td>
</tr>
<tr>
<td>64</td>
<td>102</td>
<td>05-Aug-06</td>
</tr>
<tr>
<td>74</td>
<td>103</td>
<td>05-Aug-06</td>
</tr>
</tbody>
</table>

Unpacking the expression

The selection \( \sigma_{\text{bid}=103}(\text{reserves}) \) extracts from \( \text{reserves} \) the entries relating to boat 103.

We then compute the *natural join* of this relation with \( \text{sailor} \) getting

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>date</th>
<th>sname</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>103</td>
<td>11-Aug-06</td>
<td>Dustin</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>103</td>
<td>03-Aug-06</td>
<td>Lubber</td>
<td>55.5</td>
</tr>
<tr>
<td>74</td>
<td>103</td>
<td>05-Aug-06</td>
<td>Horatio</td>
<td>35.0</td>
</tr>
</tbody>
</table>
Unpacking this expression cont.

The final stage is the projection onto the single field \textit{sname} resulting in the final relation

\begin{center}
\begin{tabular}{l}
\textit{sname} \\
Dustin \\
Lubber \\
Horatio \\
\end{tabular}
\end{center}

The query \(\pi_{\textit{sname}}(\sigma_{\textit{bid}=103}(\text{reserves} \bowtie \text{sailor}))\) produces the same answer, but generates much larger intermediate relations.

One task of the \textit{query optimizer} is to take an SQL query and to determine an \textit{evaluation strategy} by converting it into an equivalent, but more efficient relational expression.

---

Example 2

\textit{Find the names of the sailors who have reserved a red boat}

\(\pi_{\textit{sname}}((\sigma_{\textit{colour}='red'} \bowtie \text{reserves} \bowtie \text{sailor}))\)

This query proceeds as follows:
- First \textit{select} the rows corresponding to red boats from \textit{boat}.
- Next form the natural join of that table with \textit{reserves} to find all the information about reservations involving red boats.
- Then form the natural join of \textit{that} relation with \textit{sailor} to join the personal information about the sailors.
- Finally \textit{project} out the sailor’s name.
Queries

Step 1

We can execute this step-by-step in MySQL to see what happens:

\[ \sigma_{\text{colour} = 'red'} \text{boat} \]

<table>
<thead>
<tr>
<th>bid</th>
<th>name</th>
<th>colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

Notice that the natural join (\(\bowtie\)) has removed the second occurrence of the join column bid.

Step 2

\[(\sigma_{\text{colour} = 'red'} \bowtie \text{reserves}) \bowtie \text{sailor}\]

<table>
<thead>
<tr>
<th>bid</th>
<th>name</th>
<th>colour</th>
<th>sid</th>
<th>date</th>
<th>sname</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
<td>22</td>
<td>2006-08-10</td>
<td>Dustin</td>
<td>45</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
<td>22</td>
<td>2006-08-12</td>
<td>Dustin</td>
<td>45</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
<td>31</td>
<td>2006-08-02</td>
<td>Lubber</td>
<td>55.5</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
<td>31</td>
<td>2006-08-17</td>
<td>Lubber</td>
<td>55.5</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
<td>64</td>
<td>2006-08-18</td>
<td>Horatio</td>
<td>35</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
<td>64</td>
<td>2006-08-05</td>
<td>Horatio</td>
<td>35</td>
</tr>
</tbody>
</table>

This has now joined the sailor relation using the sid field and removed the second occurrence of this field.

Step 3

\[(\sigma_{\text{colour} = 'red'} \bowtie \text{reserves}) \bowtie \text{sailor} \]

<table>
<thead>
<tr>
<th>sname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dustin</td>
</tr>
<tr>
<td>Lubber</td>
</tr>
<tr>
<td>Horatio</td>
</tr>
</tbody>
</table>

This final step projects the column sname and removes the duplicates.
In MySQL

In MySQL we can make this query in our usual way by performing the joins “manually”.

```
SELECT DISTINCT S.sname
FROM boat B, reserves R, sailor S
WHERE
  B.colour = 'red' AND
  B.bid = R.bid AND
  R.sid = S.sid;
```

In this query, the two joins are performed by the two conditions \( B.bid = R.bid \) and \( R.sid = S.sid \) while the selection of the red boat is the first condition.

Joins in MySQL

MySQL allows you to specify various joins, including natural joins, directly.

```
SELECT DISTINCT S.sname
FROM boat B NATURAL JOIN reserves NATURAL JOIN sailor S
WHERE B.colour = 'red';
```

Notice that seeing `reserves` is just used to join `boat` and `sailor`, there was no need to give it an alias.

Renaming

It is often convenient to assign a name to one of the “intermediate” tables in a relational algebra expression.

If \( E(R, S, T) \) is any relational algebra expression involving relations \( R, S, T \) for example, then

\[
\rho(A, E(R, S, T))
\]

means “create a new relation called \( A \) by evaluating the expression \( E(R, S, T) \).

In R&G, they introduce some (ugly) notation for renaming the columns of an intermediate relation, but we will not use that.

\[
\rho \text{redboat}, \sigma_{\text{colour} = 'red'} \text{boat}
\]

Example 3

Find the names of the sailors who have hired a red or a green boat

\[
\rho(\text{tempboat}, \sigma_{\text{colour} = 'red'} \lor \sigma_{\text{colour} = 'green'} \text{boat})
\]

\[
\pi_{\text{sname}}(\text{tempboat} \bowtie \text{reserves} \bowtie \text{sailor})
\]
In MySQL

We can perform this process exactly like this in MySQL if desired, but at the expense of creating two new tables.

CREATE TEMPORARY TABLE tempboat LIKE boat;
INSERT INTO tempboat
(SELECT * FROM boat WHERE colour = 'red'
OR colour = 'green');
SELECT DISTINCT S.sname
FROM tempboat NATURAL JOIN reserves NATURAL JOIN sailor S;

A temporary table will be removed when the client disconnects from the server.

Alternatively

An alternative in this case is to find the sailors who have used red boats and green boats in two separate queries.

```
SELECT S.sname
FROM boat B NATURAL JOIN reserves NATURAL JOIN sailor S
WHERE B.colour = 'red'
UNION
SELECT S.sname
FROM boat B NATURAL JOIN reserves NATURAL JOIN sailor S
WHERE B.colour = 'green';
```

**Intersection**

In relational algebra we can frame this query quite naturally by using intersection instead of union.

```
π_{sname}((σ_{colour='red'\∧\ colour='green'}\boat) \bowtie \ reservations \bowtie \ sailor)
∩
π_{sname}((σ_{colour='green'}\boat) \bowtie \ reservations \bowtie \ sailor)
```

Unfortunately, MySQL 5.0.x does not support an INTERSECTION operator so this cannot be translated directly into MySQL.
Two boats

A relational algebra query that can be translated directly into MySQL uses the concept of two boats reserved by the same sailor.

\[
\rho(R_1, \pi_{\text{sid}}(\sigma_{\text{colour} = \text{'red'}\text{boat} \bowtie \text{reserves}})) - \text{corrected from } \sigma \\
\rho(R_2, \pi_{\text{sid}}(\sigma_{\text{colour} = \text{'green'}\text{boat} \bowtie \text{reserves}})) - \text{corrected from } \sigma \\
\pi_{\text{sname}}(\text{sailor} \bowtie (R_1 \bowtie_{\text{bid}=\text{R}_1.\text{sid}=\text{R}_2.\text{sid}} R_2))
\]

Here \(R_1\) is a list of "red-boat reservations" and \(R_2\) is a list of "green-boat reservations". Notice that we cannot perform a natural join on \(R_1\) and \(R_2\) in this situation because both relations have a field called \(\text{bid}\) that we do not want to join on.

In MySQL

This translates into MySQL as

```sql
SELECT DISTINCT S.sname
FROM sailor S, reserves R1, reserves R2, boat B1, boat B2
WHERE R1.bid = B1.bid AND B1.colour = 'red'
AND R2.bid = B2.bid AND B2.colour = 'green'
AND R1.sid = S.sid AND R2.sid = S.sid;
```

We can view this query as finding two boat-reservations \((B_1, R_1)\) and \((B_2, R_2)\) that prove that a given sailor has reserved a red boat and also a green boat.

Two boats - a different way

A relational algebra query that can be translated directly into MySQL uses the concept of two boats reserved by the same sailor.

\[
\rho(R_1, \pi_{\text{sid}}(\sigma_{\text{colour} = \text{'red'}\text{boat} \bowtie \text{reserves}})) \\
\rho(R_2, \pi_{\text{sid}}(\sigma_{\text{colour} = \text{'green'}\text{boat} \bowtie \text{reserves}})) \\
\pi_{\text{sname}}(\text{sailor} \bowtie (R_1 \bowtie_{\text{sid}=\text{R}_1.\text{sid}} R_2))
\]

Here \(R_1\) is a list of the sailor ids of "red-boat reservations" and \(R_2\) is a list for "green-boat reservations".

The division operator

The division operator is an interesting operator that is useful in answering queries that involve “for all” statements.

Consider two relations \(A\) and \(B\) where \(A\) has two columns \(D_x\) and \(D_y\) and \(B\) has a single column \(D_y\).

If we “divide” \(A\) by \(B\) then the resulting relation \(Q = A/B\) has the single column \(D_x\) and is defined as follows:

\(Q\) has a row with value \(x\) if and only if

\[(x, y) \in A \text{ for all } y \in B\]
Example

If $A$ and $B$ are these two relations, then what values are in $A/B$?

We need to find values of $\text{sid}$ that occur in $A$ with every value of $\text{bid}$ — looking at the tables we see that the only value for which this is true is $\text{sid} = 22$.

Relation $A$

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>22</td>
<td>102</td>
</tr>
<tr>
<td>22</td>
<td>103</td>
</tr>
<tr>
<td>22</td>
<td>104</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
</tr>
<tr>
<td>31</td>
<td>103</td>
</tr>
<tr>
<td>31</td>
<td>104</td>
</tr>
<tr>
<td>64</td>
<td>101</td>
</tr>
<tr>
<td>64</td>
<td>102</td>
</tr>
<tr>
<td>74</td>
<td>103</td>
</tr>
</tbody>
</table>

Relation $B$

<table>
<thead>
<tr>
<th>bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
</tr>
<tr>
<td>102</td>
</tr>
<tr>
<td>103</td>
</tr>
<tr>
<td>104</td>
</tr>
</tbody>
</table>

Relation $A/B$

![Table](GF_Royle_N_Spadaccini_2006-2010_Databases_Relational_Algebra_II_20_28.png)

Why division?

What is the motivation for the name “division”?

If we consider integer division, then we could define the quotient $q = a/b$ by

“the quotient $q$ of two positive integers $a$ and $b$ is the largest integer such that $qb \leq a$.”

In relational algebra, the quotient relation $Q = A/B$ of two relations $A$ and $B$ is the maximal relation such that

$Q \times B \subseteq A.$

Queries with division

Find the names of the sailors who have reserved all the boats

$\rho(\text{temp}, (\pi_{\text{sid,bid}} \text{reserves}/\pi_{\text{bid}} \text{boat}))$

$\pi_{\text{sname}}(\text{temp} \bowtie \text{sailor})$

The presence of the phrase all the or every is usually a give-away that the division operator should be used.

Re-expressing division

Suppose that $A$ and $B$ are defined as above. Then

$\pi_{Dx}(A)$

contains all the $x$-values that appear in $A$. Which of these should be in the quotient relation $Q = A/B$?

An $x$-value does not appear in $Q$ if there is some $y$ value in $B$ such that $(x,y)$ is not in $A$. This latter relation is given by

$(\pi_{Dx}(A) \times B) - A$
Re-expressing division, cont

Therefore

$$\pi_{DX}(\pi_{DX}(A) \times B - A)$$

contains all the x-values that should not be in the quotient, and hence

$$Q = \pi_{DX}(A) - \pi_{DX}((\pi_{DX}(A) \times B) - A)$$

computes the quotient by taking all the possible x-values and then removing all those that should not be there.

Therefore division is not actually a new operation, but just a simpler way of expressing a common complex operation.

More Joins

The joins that we have seen so far are all of the form

$$A \bowtie B$$

which are examples of inner joins.

The only rows in the join are those where a row of A matches a row of B according to the join condition. However for some applications it is useful for the join to have a row for every row of A even if there is no matching row in B.

A left outer join of A and B with join condition c has two types of row

- Rows consisting of a row of A combined with a row of B where the join condition is satisfied
- Rows consisting of a row of A combined with a number of NULL fields if that row of A would not occur otherwise.

Which sailors have no reservations?

An example where this is useful is where we want to find out which sailors have no corresponding reservations.

```sql
SELECT * FROM
  sailor LEFT JOIN reserves
ON sailor.sid = reserves.sid;
```

Here the command specifies a LEFT JOIN and explicitly gives the join condition.

Result

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>2006-08-10</td>
</tr>
<tr>
<td>22</td>
<td>Dustin</td>
<td>45</td>
<td>22</td>
<td>102</td>
<td>2006-08-10</td>
</tr>
<tr>
<td>22</td>
<td>Dustin</td>
<td>45</td>
<td>22</td>
<td>103</td>
<td>2006-08-11</td>
</tr>
<tr>
<td>22</td>
<td>Dustin</td>
<td>45</td>
<td>22</td>
<td>104</td>
<td>2006-08-12</td>
</tr>
<tr>
<td>29</td>
<td>Brutus</td>
<td>33</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>55.5</td>
<td>31</td>
<td>102</td>
<td>2006-08-02</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>55.5</td>
<td>31</td>
<td>103</td>
<td>2006-08-03</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>55.5</td>
<td>31</td>
<td>104</td>
<td>2006-08-17</td>
</tr>
<tr>
<td>32</td>
<td>Andy</td>
<td>25.5</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>35</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>64</td>
<td>Horatio</td>
<td>35</td>
<td>64</td>
<td>102</td>
<td>2006-08-18</td>
</tr>
<tr>
<td>64</td>
<td>Horatio</td>
<td>35</td>
<td>64</td>
<td>100</td>
<td>2006-08-05</td>
</tr>
<tr>
<td>71</td>
<td>Zorba</td>
<td>16</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>74</td>
<td>Horatio</td>
<td>35</td>
<td>74</td>
<td>103</td>
<td>2006-08-05</td>
</tr>
<tr>
<td>85</td>
<td>Art</td>
<td>25.5</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>63.5</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Which have no reservations

To check those with no reservations, we check that the second occurrence of the field sid is NULL in the joined table.

```
SELECT sname FROM sailor LEFT JOIN reserves
    ON sailor.sid = reserves.sid
WHERE reserves.sid IS NULL;
```

+--------+
| sname   |
+--------+
| Brutus  |
| Andy    |
| Rusty   |
| Zorba   |
| Art     |
| Bob     |
+--------+

This lecture

We continue our coverage of the fundamentals of SQL/MySQL with stored routines.

A stored routine is a named set of SQL statements that is stored on the server and which can be initiated by a single call.

Stored routines are further subdivided into procedures which do not return anything (although they can assign values to variables) and functions that return values to the caller.

In MySQL 5.0.x, functions have a major limitation in that they cannot access tables, so they are essentially just a way to add special-purpose calculations to MySQL.
Rationale for stored routines

A stored routine is maintained on the server which has various consequences both positive and negative:

- A complex sequence of SQL statements can be prepared once by a professional DBA and then made available to all client programs
- Stored routines can access confidential or sensitive tables without exposing them to client programs
- Processing becomes more centralized with the server taking on a greater computational load

Basic Syntax

The basic syntax for creating the simplest possible procedure, one with no parameters and consisting of a single SQL statement is as follows:

```
CREATE PROCEDURE myproc()
    /* An SQL statement */
```

For example, in the `world` database we could issue the command:

```
CREATE PROCEDURE listCapitals()
    SELECT C.name, T.name
    FROM country C, city T
    WHERE C.capital = T.id;
```

Calling a user-defined procedure

```
CALL listCapitals();
```

```
+---------------------------+---------------+
| name | name |
+---------------------------+---------------+
| Afghanistan | Kabul |
| Netherlands | Amsterdam |
| Netherlands Antilles | Willemstad |
```

A stored procedure “belongs” to a specific database (the one in use when the `CREATE PROCEDURE` command was issued, and so this procedure belongs to the `world` database.

Procedure parameters

In order to do anything more useful than pure textual replacement, a procedure needs to have `parameters` that the user can specify on calling.

```
CREATE PROCEDURE listOneCapital(cntry VARCHAR(50))
    SELECT C.name, T.name
    FROM country C, city T
    WHERE C.capital = T.id
    AND C.name = cntry;
```

This procedure has one parameter called `cntry` which is of type `VARCHAR(50)`.
Calling the procedure

As usual, when the procedure is called the caller specifies an actual argument which is used in place of the formal parameter.

CALL listOneCapital('Australia');
+-----------+----------+| name | name | +-----------+----------+| Australia | Canberra | +-----------+----------+

Output Parameters

A procedure has no RETURN statement and hence cannot return a value to the caller. However the caller can specify a user-variable in the parameter list to which the procedure can assign a value.

CREATE PROCEDURE regionPop(rgn TEXT, OUT rpop INT)
    SELECT SUM(population) FROM country C
    WHERE C.region = rgn
    INTO rpop;

The output parameter rpop is indicated by the keyword OUT and the SELECT statement performs the selection INTO the variable.

Using output parameters

When this procedure is called the user must give a variable name for the second argument

CALL regionpop('North America', @napop);

Nothing appears on the terminal, but the variable @napop has had a value assigned to it, which can subsequently be used.

SELECT @napop;
+-----------+| @napop | +-----------+| 309632000 | +-----------+

Multiple statements

To enhance our procedures further we need to be able to perform a sequence of SQL statements inside a procedure, not just a single statement.

This can be done by putting the statements between BEGIN and END.

CREATE PROCEDURE myproc()
BEGIN
    /* A whole bunch of MySQL statements */
END

One problem that immediately arises is how to terminate each of the statements inside the BEGIN/END area — if we just use the semicolon then MySQL will think that the procedure definition has terminated.
Temporarily change delimiters

The solution to this is to temporarily change the delimiter so that we can enter the entire procedure.

```
DELIMITER ++
CREATE PROCEDURE myproc()
BEGIN
/* A whole bunch of MySQL statements */
/* each terminated with the usual semicolon */
END++
DELIMITER ;
```

The first line temporarily changes the delimiter to `++`, then the entire procedure is entered, and finally the delimiter is changed back again.

Procedure Variables

Of course, in order to use multiple statements effectively it helps to be able to use “local variables” within the procedure\(^1\).

```
CREATE PROCEDURE regionSummary(rgn TEXT)
BEGIN
    DECLARE rp INT;
    CALL regionPop(rgn, rp);
END
```

This fragment creates a local variable called `rp` and then calls the previously defined procedure to assign the total population of the specified region to that variable.

```
mysql> CALL regionSummary("Caribbean");
+--------------------+------------+---------+
| name               | population | perc    |
|--------------------+------------+---------|
| Puerto Rico        | 3869000    | 10.1442 |
| Jamaica            | 2583000    | 6.7724  |
+--------------------+------------+---------+
```

\(^1\)Henceforth I will not include the `DELIMITER` statements

Multiple statements

We can complete this procedure fragment by using the variable that we have just evaluated in a subsequent SQL statement.

```
CREATE PROCEDURE regionSummary(rgn TEXT)
BEGIN
    DECLARE rp INT;
    CALL regionPop(rgn, rp);
    SELECT C.name, C.population, 
         C.population / rp * 100 as perc
    FROM country C
    WHERE C.region = rgn
    ORDER BY C.population DESC
    LIMIT 5;
END
```

This has simply added one more `SELECT` statement that performs another query to list the five most populous countries in that region.
Other constructs

In addition to this basic functionality, stored procedures can also perform rudimentary selection and repetition with constructs such as:

- IF-THEN-ELSE
- WHILE...END WHILE
- REPEAT...END REPEAT
- LOOP...END LOOP

Largest and Smallest

Suppose that instead of the top five countries for the specified region, we wanted to list the most populous and least populous.

We could do this with three SELECT statements — one to find the minimum and maximum country populations in that region, then one each to find which country has the minimum and the maximum population.

However we could do this with just one SELECT statement provided we could process the results afterwards.

Cursors

A cursor is essentially a mechanism to store the results of a query, and to process the results row-by-row.

In MySQL 5.0.x, cursor support is very limited and currently cursors can only be created in procedures and they can only be processed row-by-row from start to finish.

Cursors essentially support only four statements:

- DECLARE CURSOR FOR declares a cursor
- OPEN... opens the cursor
- FETCH... INTO fetches the current row for processing
- CLOSE... closes the cursor

Cursor control

CREATE PROCEDURE regionLimits(rgn TEXT)
BEGIN

  DECLARE regionOnly CURSOR FOR
    SELECT C.name, C.population
    FROM country C
    WHERE region = rgn;

  OPEN regionOnly;

  /* process the rows */

  CLOSE regionOnly;
END
How many rows?
We will use a loop to process each row, and so we need to know how many rows the cursor contains; this can be found from the MySQL function FOUND_ROWS() which returns the number of rows that the last query found.

```sql
DECLARE numRows INT;
DECLARE numDone INT;
/* Declare cursor */
OPEN regionOnly;
SELECT FOUND_ROWS() INTO numRows;
WHILE numDone < numRows DO /* Process a row */
    SET numDone = numDone + 1;
END WHILE;
```

(GF Royle, N Spadaccini 2006-2010)

Storing max and min
In order to use the cursor to process each row, we need to have variables to store the name and population of the most populous and least populous countries and variables for the contents of each row. So the declaration section will need to have the following added to it:

```sql
DECLARE minP INT;
DECLARE maxP INT;
DECLARE minC VARCHAR(50);
DECLARE maxC VARCHAR(50);
DECLARE cname VARCHAR(50);
DECLARE cpop INT;
```

(GF Royle, N Spadaccini 2006-2010)

Initializing
These variables need to be initialized to have the values of the first city in the list. So immediately after the SELECT FOUND_ROWS() INTO numRows we put

```sql
FETCH regionOnly INTO cname, cpop;
SET minP = cpop;
SET maxP = cpop;
SET minC = cname;
SET maxC = cname;
SET numDone = 1;
```

(GF Royle, N Spadaccini 2006-2010)

Inside the loop
In the loop, we fetch the contents of the next row and compare them to the existing minimum/maximum values:

```sql
WHILE numDone < numRows DO
    FETCH regionOnly INTO cname, cpop;
    IF (cpop < minP) THEN
        SET minP = cpop;
        SET minC = cname;
    END IF;
    IF (cpop > maxP) THEN
        SET maxP = cpop;
        SET maxC = cname;
    END IF;
    SET numDone = numDone + 1;
END WHILE;
```
Finally

And finally after the loop we “print” the output.

SELECT minC as smallest,  
    minP as smallestPop,  
    maxC as largest,  
    maxP as largestPop;

The output from the whole procedure is then something like

CALL regionLimits("Caribbean");

<table>
<thead>
<tr>
<th>smallest</th>
<th>smallestPop</th>
<th>largest</th>
<th>largestPop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anguilla</td>
<td>8000</td>
<td>Cuba</td>
<td>11201000</td>
</tr>
</tbody>
</table>

Conclusion

Although MySQL supports stored procedures and functions, the programming tools available are very rudimentary and awkward compared to a general-purpose programming language.

Therefore while stored routines are extremely useful when they consist of things that can be expressed easily in SQL, they become very awkward when performing general processing.

Therefore in the absence of a compelling reason (e.g. security) to use stored routines, most non-SQL processing should be performed at the client, and not on the server.

This lecture

The lecture starts discussion of how a Java-based application program connects to a database using JDBC.
Application Programs

The `mysql` client program provides a low-level interface to the raw commands of MySQL, and as such is suitable only for expert users writing their own SQL/MySQL queries.

However there are many reasons to connect other application programs to an SQL and/or MySQL database — these programs may essentially be about providing access to the database in a special way or they may just use the database “in the background” to provide other functionality.

Database Access – End Users

An organization is likely to have only a few DBA professionals, but many more users who need to query the database.

Suitable client programs can be written to provide simplified interfaces, usually graphical user interfaces for end-users. These interfaces would usually:

- Customize the handling and error-checking of input
- Construct valid SQL queries from the input
- Run the SQL queries against the database
- Customize and format the output

Database Access – Expert Users

A second use for additional client programs would be to provide expert users with specialized tools for visualizing and working with a whole database — we have already seen such a tool in `DBVisualizer` and there are many more such tools.

Database Integration

As well as client programs whose primary purpose is providing database access, there are many situations where a client program may benefit from database integration.

In this situation the client program may wish to use a database in order to accomplish its primary purpose—for example, a management information system may access a database to obtain recent sales figures, but then perform significant additional processing or analysis on the results.
Database Backed Applications

A third reason to connect an application to a database is to use the database as a tool to provide the essential functionality of the application — here the database is supporting the use of the application, rather than the application supporting the use of the database.

The typical example of such an application is a web-based bulletin board system where users post and view comments, perform searches and so on. Frequently the actual comments are stored in a database, and the web-pages that are displayed are constructed “on-the-fly” from the results of database queries.

In this situation the user may not even be aware of the underlying database.

---

Connecting

There are many different ways of connecting an application written in a general purpose programming language (henceforth called the host language) to a MySQL server.

- The C API is a low-level client library for connecting C programs directly to a MySQL server.
- The Perl and PHP APIs provide higher level APIs for connecting to a number of different databases from Perl or PHP respectively
- JDBC is a standard API for connecting Java programs to a number of different databases.

In this lecture, we will consider how to connect Java applications to MySQL.

---

What is JDBC?

JDBC stands for Java DataBase Connectivity — it is an API (Application Programmers Interface) that defines a set of interfaces for interaction with a database.

Each database vendor writes a specific JDBC driver which is a set of classes that implements the JDBC interface on their database. For example, Oracle has a JDBC driver for Oracle databases, MySQL has a JDBC driver for MySQL databases and so on.

If the application programmer also writes code that conforms to the JDBC standard, then the application can interact with a number of different databases simply by loading a different driver.

---

Conceptual Diagram

Diagram from “Java Enterprise in a Nutshell” by Crawford, Farley & Flanagan, O’Reilly.
JDBC Architecture

A JDBC (Java DataBase Connectivity) application can be viewed as consisting of four components:

- The Java application program
  The application initiates a connection with the server, submits SQL statements and processes the results.

- The driver manager
  The driver manager loads JDBC drivers and passes the JDBC calls to the correct driver.

- The drivers
  A driver is database-specific code that translates JDBC calls into the right form for its particular database, and translates the results back into the JDBC standard.

- The data sources
  The underlying database or databases.

JDBC Basics

A JDBC application must perform the following steps:

- Load the driver class for the particular database being used.
- Open a connection to the database.
- Create SQL queries and use the connection to execute them.
- Process the results obtained from the database.

Loading a driver

In your class file you must always import Java's sql package.

```java
import java.sql.*;
```

A driver is loaded by using Java's dynamic loading mechanism `Class.forName` with the correct driver name, which for the MySQL driver is `com.mysql.jdbc.Driver`.

This method might throw an exception, so this must be caught.

```java
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
    System.out.println("Cannot load Driver");
}
```

Opening a connection

Once the driver has been loaded the static methods in the class `DriverManager` become available; the important method here is the method

```java
public static Connection getConnection(String url) throws SQLException
```

This method takes a `String` in a URL-like format

```java
jdbc:driver:database
```

which specifies JDBC, then the name of the driver and then connection details for the database (user name, password if any, and database).

---

1 The JDBC-related classes discussed here are in the package `java.sql`.
Opening a local connection

Connecting as the root user to the world database stored on a MySQL server running on the local host would be done as follows:

```java
try {
    String url = "jdbc:mysql://localhost/world?user=root";
    Connection conn = DriverManager.getConnection(url);
} catch (SQLException e) {
    System.out.println("Cannot open Connection");
    System.exit(0);
}
```

The precise form of the URL will depend on the particular driver, but in all cases the result is the object `conn` which represents an open connection to the database.

Queries

An actual SQL query is executed by the method

```java
public ResultSet executeQuery(String sql)
```

of the Statement object.

A query that executes successfully returns a ResultSet object which represents the table that the SQL query returns.

It is important to note that the ResultSet object “belongs” to the Statement object and that if the Statement object is re-used for another query, then the ResultSet object will also be closed.

Statements

An object from the Statement class is used to send your SQL statements to the database — such an object is obtained from the Connection object that we just obtained from the DriverManager.

```java
try {
    String url = "jdbc:mysql://localhost/world?user=root";
    Connection conn = DriverManager.getConnection(url);
    Statement stmt = conn.createStatement();
} catch (SQLException e) {
    System.out.println("Cannot open Connection");
    System.exit(0);
}
```

Executing a query

Suppose we simply want to execute a query that returns the data stored in the country table from our database.

```java
Statement stmt = conn.createStatement();
String sql = "SELECT * FROM country";
ResultSet rs = stmt.executeQuery(sql);
```

(This code must all be placed within the try/catch clause because pretty much all of these methods might throw an SQLException.)
Processing the results

A `ResultSet` object can be viewed as a table of results that can be stepped through “row-by-row” — at each stage the object has a notion of a “current row”.

In order to process the results we need methods to

- Change the current row
- Determine column values from the current row

Changing the current row

The key methods for moving through a `ResultSet` are

- `public boolean first()`  
  This moves to the first row of the result set
- `public boolean next()`  
  This moves to the next row of the result set, returning `false` if there are no further rows to move to.

In earlier versions of JDBC a result set could only be processed by starting at the first row and going through one row at a time. This is still the JDBC specification of the default `ResultSet` but the MySQL drivers return a “scrollable” `ResultSet` no matter what you ask for.

Process each row

Thus to examine each row in turn uses code of the following form (assume that `rs` has already been obtained in the manner described above).

```
while (rs.next()) {
    // process the row
}
```

Reading column values

Reading a column value from the current row of the `ResultSet` is done by one of the `getXXX()` methods, where `XXX` is a Java type. For example, any column that is a `CHAR`, `VARCHAR` or `TEXT` type in the database can be accessed by using

- `getString(String columnName)`
- `getString(int columnIndex)`

where the column to be retrieved can either be specified by `name` or by `position`.

Just to confuse everyone, the columns in SQL are indexed from position 1, not the 0-based indexing of Java, C, C++ etc.
Brief Example

This code simply pulls out the columns name and population from each row of the ResultSet and prints them out.

```java
while (rs.next()) {
    String cntry = rs.getString("name");
    long popn = rs.getLong("population");

    System.out.println(cntry + " " + popn);
}
```

What have we achieved?

In one sense, all we have done is written a Java client program that executes the SQL statement

```
SELECT name, population FROM country;
```

in a round-about way.

The important thing to realize is that we now have the database data inside the Java program where it can be post-processed in any number of ways.

This lecture

This lecture introduces normal forms, decomposition and normalization.

We will explore problems that arise from poorly designed schema, and introduce "decomposition".
Redundancy

One of the main reasons for using relational tables for data is to avoid the problems caused by redundant storage of data. For example, consider the sort of general information that is stored about a student:
- Student Number
- Name
- Address
- Date of Birth

A number of different parts of the university may keep different additional items of data regarding students, such as grades, financial information and so on.

Problems with redundancy

Apart from unnecessary storage, redundancy leads to some more significant problems:
- **Update Anomalies**
  - If one copy of a data item is updated — for example, a student changes his or her name, then the database becomes inconsistent unless every copy is updated.
- **Insertion Anomalies**
  - A new data item, for example a new mark for a student, cannot be entered without adding some other, potentially unnecessary, information such as the student’s name.
- **Deletion Anomalies**
  - It may not be possible to delete some data without losing other, unrelated data, as well (an example is on the next slide).

Deletion Anomalies

A deletion anomaly occurs when a table storing redundant information becomes a proxy for storing that information properly.

For example, suppose that a company pays fixed hourly rates according to the level of an employee.

<table>
<thead>
<tr>
<th>Name</th>
<th>Level</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>10</td>
<td>25.00</td>
</tr>
<tr>
<td>Jones</td>
<td>8</td>
<td>20.50</td>
</tr>
<tr>
<td>Tan</td>
<td>10</td>
<td>25.00</td>
</tr>
<tr>
<td>White</td>
<td>9</td>
<td>22.00</td>
</tr>
</tbody>
</table>

This table contains redundant data, because the student’s name is needlessly repeated.

If the financial system also stores student numbers and names, then there is redundancy between tables as well as within tables.
Redundancy

What if Jones leaves?

If Jones happens to be the only employee currently at level 8, and he leaves and is deleted from the database, then the more general information that “The hourly rate for Level 8 is $20.50” is also lost.

In this situation the right approach is to keep a separate table that relates levels with hourly rates, and to remove the “rate” information from the employee table.

<table>
<thead>
<tr>
<th>Level</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>8</td>
<td>20.50</td>
</tr>
<tr>
<td>9</td>
<td>22.00</td>
</tr>
<tr>
<td>10</td>
<td>25.00</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

(GF Royle, N Spadaccini 2006-2010) Databases - Normalization I

Separating the student tables

The redundancy problems with the student information can also be resolved by creating a separate table with just the basic student information.

<table>
<thead>
<tr>
<th>Student Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>14058428</td>
<td>John Smith</td>
</tr>
<tr>
<td>15712381</td>
<td>Jill Tan</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student Number</th>
<th>Unit Code</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>14058428</td>
<td>CITS 3240</td>
<td>72</td>
</tr>
<tr>
<td>14058428</td>
<td>CITS 1200</td>
<td>68</td>
</tr>
<tr>
<td>14058428</td>
<td>CITS 2200</td>
<td>77</td>
</tr>
<tr>
<td>15712381</td>
<td>CITS 1200</td>
<td>88</td>
</tr>
<tr>
<td>15712381</td>
<td>CITS 2200</td>
<td>82</td>
</tr>
</tbody>
</table>

What is the algorithm for deciding how to split a relation?

Decomposition

Both of these examples were solved by replacing the original redundant table by two tables each containing a subset of the original fields. This leads to the following definition:

A decomposition of a relational schema R consists of replacing the relational schema by two (or more) relational schemas each containing a subset of the attributes of R and that together contain all of the attributes of R.

Data stored under the original schema is projected onto each of the schemas in the decomposition.

Decomposition - formally

Given a relation \( R(A_1, A_2, \ldots, A_n) \) we may decompose \( R \) into the relations \( S(B_1, B_2, \ldots, B_m) \) and \( T(C_1, C_2, \ldots, C_k) \) such that:

1. \( \{A_1, A_2, \ldots, A_n\} = \{B_1, B_2, \ldots, B_m\} \cup \{C_1, C_2, \ldots, C_k\} \)
2. \( S = \pi_{B_1, B_2, \ldots, B_m}(R) \)
3. \( T = \pi_{C_1, C_2, \ldots, C_k}(R) \)
Example

Suppose that $R$ is the original “Student Number / Name / Unit Code / Mark” schema above — we’ll abbreviate this to

$$R = SNUM$$

(S = Student Number, N = Name, U = Unit Code, M = Mark).

Then the decomposition suggested above would decompose $R$ into

$$R_1 = SUM \quad R_2 = SN$$

Storing and Recovering Data

Any data stored in the original schema is stored in the decomposed schema(s) by storing its projections onto $R_1$ and $R_2$.

We can then recover the original data by performing a join of the decomposed relations. In this situation we have the property that for every legal instance $r$ of $R$

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r)$$

In other words, the original relation is the join of the decomposed relations.

Lossless-join decomposition

The property that any instance of the original relation can be recovered as the join of the decomposed relations is called lossless-join decomposition.

A decomposition of a relational schema $R$ into $R_1$ and $R_2$ is lossless-join if and only if the set of attributes in $R_1 \cap R_2$ contains a key for $R_1$ or $R_2$.

For our example above, $R_1 \cap R_2$ is the single attribute $S$ (student number) which is a key for $R_2 = SN$ and hence the decomposition is lossless-join.

Other decompositions

In general, an arbitrary decomposition of a schema will not be lossless join.

The attributes in $R_1 \cap R_2$ is $B$ which is a key for neither $AB$ nor $BC$, so the condition for lossless join is not met.
Lossy join

Now consider the join $\pi_{AB}(r) \bowtie \pi_{BC}(r)$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_3$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_1$</td>
<td>$c_3$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
</tbody>
</table>

This contains two tuples that were not in the original relation — because $b_1$ is associated with both $a_1$ and $a_3$ in the first relation, and $c_1$ and $c_3$ in the second.

Problems with decomposition

Some types of redundancy in (or between) relations can be resolved by decomposition. However decomposition introduces its own problems, in particular the fact that queries over the decomposed schemas now require joins; if such queries are very common then the deterioration in performance may be more severe than the original problems due to redundancy.

To make informed decisions about whether to decompose or not requires a formal understanding about the types of redundancy and which can be resolved through decomposition — this is the theory of functional dependencies.

Functional dependencies

A functional dependency (an FD) is a generalization of the concept of a key in a relation.

Suppose that $X$ and $Y$ are two subsets of the attributes of a relation such that whenever two tuples have the same values on $X$, then they also have the same values on $Y$. In this situation we say that $X$ determines $Y$ and write $X \rightarrow Y$.

Note that this condition must hold for any legal instance of the relation.

Keys

The obvious functional dependencies come from the keys of a relation.

For example, in the student-number / name relation $SN$ we have the obvious functional dependency $S \rightarrow N$ meaning that the student number determines the name of the student.

Obviously $S$ determines $S$ and so $S \rightarrow SN$ which is just another way of saying that the student-number is a key for the whole relation.
Superkeys

A key is a minimal set of attributes that determines all of the remaining attributes of a relation.

For example, in the SNUM relation above, the pair SU is a key because the student number and unit code determine both the name and the mark, or in symbols

\[ SU \rightarrow SNUM. \]

(It is clear that no legal instance of the relation can have two tuples with the same student number and unit code, but different names or marks.)

Any superset of a key is called a superkey — it determines all of the remaining attributes, but is not minimal.

Reasoning about FDs

Often some functional dependencies will be immediately obvious from the semantics of a relation, but others may follow as a consequence of these initial ones.

For example, if \( R \) is a relation with FDs \( A \rightarrow B \) and \( B \rightarrow C \), then it follows that

\[ A \rightarrow C \]

as well.

The set of all FDs implied by a set \( F \) of FDs is called the closure of \( F \) and is denoted \( F^+ \). In order to consider the various normal forms it is important to be able to calculate \( F^+ \) given an initial set \( F \) of FDs.

Example

Consider a relation \( R \) with four attributes \( ABCD \), and suppose that the following functional dependencies hold:

\[ ABC \rightarrow D \]
\[ D \rightarrow A \]

What are the keys for \( R \)?

The first FD shows us that \( ABC \) is a key for this relation, but this is not the only key for this relation.

The second FD shows us that \( D \) determines \( A \), and so \( BCD \) determines all the attributes and hence is a (candidate) key.

Armstrong’s Axioms

Armstrong’s Axioms is a set of three rules that can be repeatedly applied to a set of FDs:

- Reflexivity: If \( Y \subseteq X \) then \( X \rightarrow Y \).
- Augmentation: If \( X \rightarrow Y \) then \( XZ \rightarrow YZ \) for any \( Z \).
- Transitivity: If \( X \rightarrow Y \) and \( Y \rightarrow Z \) then \( X \rightarrow Z \).

We can augment these with two rules that follow directly from Armstrong’s Axioms.

- Union: If \( X \rightarrow Y \) and \( X \rightarrow Z \) then \( X \rightarrow YZ \).
- Decomposition: If \( X \rightarrow YZ \), then \( X \rightarrow Y \) and \( X \rightarrow Z \).
The key point about Armstrong’s axioms is that they are both *sound* and *complete*. That is, if we start with a set $F$ of FDs then:

- Repeated application of Armstrong’s axioms to $F$ generates only FDs in $F^+$.
- Any FD in $F^+$ can be obtained by repeated application of Armstrong’s axioms to $F$.

Consider a relation with attributes $ABC$ and let

$$ F = \{A \rightarrow B, B \rightarrow C\} $$

Then from transitivity we get $A \rightarrow C$, by augmentation we get $AC \rightarrow BC$ and by union we get $A \rightarrow BC$. FDs that arise from reflexivity such as

$$ AB \rightarrow B $$

are known as *trivial* dependencies.

This lecture describes the Boyce-Codd normal form.
Normal forms

There is a substantial theory of various types of normal forms; essentially these are guarantees that if a relation schema does not have various types of functional dependency, then certain types of problem with redundancy cannot occur.

Associated with each type of normal form is a theory of decomposition which explains under what circumstances a schema can be decomposed in such a way that the resulting relations are in the desired normal form.

Normal form hierarchy

The normal forms based on FDs alone form a hierarchy with increasingly strict limitations on the allowed functional dependencies as follows:

- First normal form (1NF)
- Second normal form (2NF)
- Third normal form (3NF)
- Boyce-Codd normal form (BCNF)

Thus any schema that is in BCNF is automatically in 3NF, 2NF and 1NF.

BCNF

A relational schema is in Boyce-Codd normal form if for every functional dependency \( X \rightarrow A \) (where \( X \) is a subset of the attributes and \( A \) is a single attribute) either

- \( A \in X \) (that is, \( X \rightarrow A \) is a trivial FD), or
- \( X \) is a superkey.

In other words, the only functional dependencies are either the trivial ones (which always hold) or ones based on the keys of the relation.

If a relational schema is in BCNF, then there is no redundancy within the relations.

No redundancy in BCNF

Loosely speaking, a relational schema in BCNF is already in its “leanest” possible form — each attribute is determined by the key(s) alone so nothing that is stored can be deduced from a smaller amount of information.

The student number / name / unit code / mark relation \( SNUM \) from last lecture is not in BCNF because the key was

\[ SU \rightarrow SNUM \]

but there is a functional dependency

\[ S \rightarrow N \]

where \( S \) is not a superkey.
BCNF decomposition

Suppose a relation $R$ is not in BCNF. Then there must be some functional dependency

$$X \rightarrow A$$

where $X$ is not a superkey.

In this case, the relation can be decomposed into the two relations

$$R_1 = R - A \quad R_2 = XA$$

where $R - A$ simply means dropping the attribute $A$ from the schema, and $XA$ is the relation whose attributes are those in $X$ together with $A$.

BCNF decomposition cont.

If either $R_1$ or $R_2$ is not in BCNF then the process can be continued, by decomposing them in the same fashion.

By continually decomposing any relation not in BCNF into smaller relations, we must eventually end up with a collection of relations that are in BCNF.

Therefore any initial schema can be decomposed into BCNF.

The student example

Recall in the $SNUM$ relation we had the following FDs

$$SU \rightarrow SNUM$$

$$SU \rightarrow M$$

$$S \rightarrow N$$

$SU$ is the key for the relation, but the $FD S \rightarrow N$ violates BCNF. The decomposition algorithm gives the new relations are

$$R_1 = SNUM - N = SUM \quad R_2 = SN$$

$R_2$ is BCNF and $R_1$ has only one non-trivial $FD, SU \rightarrow M$ and is BCNF. This is fortuitously the decomposition we “randomly” chose in the previous lecture.

BCNF decomposition - revisited

This algorithm can be described more completely as suppose a relation $R$ is not in BCNF. Then there must be some functional dependency

$$X \rightarrow A$$

where $X$ is not a superkey.

Compute the closure of FD on $X$, that is, $X^+$. The relation can be decomposed into the two relations

$$R_1 = R - (X^+ - X) \quad R_2 = X^+$$

where $R - (X^+ - X)$ simply means dropping the attributes that are functionally dependent on $X$ from the schema, and $X^+$ is the relation whose attributes are those in $X$ together with its functional dependents.
The movie example

Consider a Movie relation consisting of

\{movieTitle, year, studioName, president, presAddr\}

With the three FDs one can assume in this relation

\[ \text{movieTitle, year} \rightarrow \text{studioName} \]
\[ \text{studioName} \rightarrow \text{president} \]
\[ \text{president} \rightarrow \text{presAddr} \]

If we compute the closing sets on these five attributes we find the only key for this relation is \{movieTitle, year\}. The last two FDs violate BCNF. Suppose we choose to decompose starting with

\[ \text{studioName} \rightarrow \text{president} \]

The closure of studioName includes presAddr so we get

\[ \text{studioName} \rightarrow \text{president, presAddr} \]

Example from R & G

Consider a relation schema CSIDPQV that represents contracts for suppliers to supply certain parts to certain projects.

- \(C\) is the Contract ID
- \(S\) is the Supplier ID
- \(J\) is the Project ID
- \(D\) is the Department ID
- \(P\) is the Part ID
- \(Q\) is the Quantity
- \(V\) is the Value

The resulting two relations are

\[ \{\text{movieTitle, year, studioName}\} \]
\[ \{\text{studioName, president, presAddr}\} \]

The first is BCNF, the second has a the key studioName. The second is not BCNF because of the presence of the FD president \(\rightarrow\) presAddr, in which president is not a super key. We apply the algorithm again to get the final BCNF decompositions,

\[ \{\text{movieTitle, year, studioName}\} \]
\[ \{\text{studioName, president}\} \]
\[ \{\text{president, presAddr}\} \]

Example cont.

Now we add some constraints that can be expressed as FDs.

- The contract ID is a key
  \[ C \rightarrow \text{CSIDPQV} \]
- A project purchases each type of part in a single contract
  \[ JP \rightarrow C \]
- A department purchases at most one part from each supplier
  \[ SD \rightarrow P \]

Is this schema in BCNF? The first FD \(JP \rightarrow C\) does not violate BCNF because \(JP\) is a key. However \(SD\) is not a key, and so \(SD \rightarrow P\) violates BCNF.
Decompose the relation

Guided by the FD $SD \rightarrow P$ we get the decomposition

$$R_1 = CSJDQV \quad R_2 = SDP$$

where both schemas are in BCNF.

(Note that this is not obvious — it requires using Armstrong’s axioms to calculate the closure of all the given FDs and checking that none of them violate the BCNF conditions.)

At first sight it would seem that we have resolved the problems with redundancy — simply decompose into BCNF at all times! However this is unfortunately not true.

A subtle problem

There is a subtle problem with the decomposition of $R$ into

$$R_1 = CSJDQV \quad R_2 = SDP$$

which is related to maintaining the key constraint $JP \rightarrow C$.

Whenever a new tuple is added to the original relation $R$ it is simple to enforce the key constraint $JP \rightarrow C$, just by checking the tuples in that one table.

However, in the decomposed relation, entering the data about a new contract involves updating two relations with the $J$ attribute going into $R_1$ and the $P$ attribute into $R_2$.

Therefore to enforce the key constraint would require the DBMS to perform a join on $R_1$ and $R_2$ just to check that a particular $JP$ pair does not appear twice.

Dependency-preserving decomposition

It would be desirable if any decomposition that we used was not only lossless-join, but also dependency preserving.

Intuitively this means that any functional dependency in the original relation can be enforced by examining just one of the relations in the decomposition.

If $F$ is a set of functional dependencies on a relation schema $R$, then a proposed decomposition of $R$ into relations with attributes $X$ and $Y$ is dependency preserving if and only if

$$(F_X \cup F_Y)^+ = F^+$$

where $F_X$ is the set of FDs involving only attributes from $X$, and similarly for $F_Y$.

Example cont.

The idea underlying a dependency-preserving decomposition is that the FDs on the original relation can be enforced just by enforcing $F_X$ on $R_1$ or $F_Y$ on $R_2$.

In the contracts example, the decomposition involves attribute sets

$$X = CSJDQV \quad Y = SDP$$

The problem occurs because the dependency $JP \rightarrow C$ is not in either $F_X$ nor in $F_Y$.

It is conceivable that $JP \rightarrow C$ might be in $(F_X \cup F_Y)^+$ through some chain of inferences using Armstrong’s axioms, but it is not true in this case.
BCNF is too strong

In fact, there is no dependency-preserving decomposition of the contracts relation into BCNF — essentially BCNF is “too restrictive” a condition.

This motivates the definition of 3rd normal form which is somewhat technical — essentially it relaxes the conditions for BCNF just enough to guarantee that there will be a dependency-preserving decomposition of any relation schema into 3NF.

Thus we have a trade-off where allowing a controlled amount of redundancy gives us more freedom in decompositions.

This lecture

This lecture describes 3rd normal form.

BCNF - recap

The BCNF decomposition of a relation is derived by a recursive algorithm. A lossless-join decomposition is derived which may not be dependency preserving.

The decomposition is too restrictive. To make the decomposition in the previous example dependency preserving we can cover the FD $JP \rightarrow C$ by adding its attributes as a relation

$$R_1 = CSJDQV \quad R_2 = SDP \quad R_3 = JPC$$

We have added the required FD involving key attributes that were prohibited by BCNF.
3NF

A relational schema is in 3NF if for every functional dependency \( X \rightarrow A \) (where \( X \) is a subset of the attributes and \( A \) is a single attribute) either

- \( A \in X \) (that is, \( X \rightarrow A \) is a trivial FD), or
- \( X \) is a superkey, or
- \( A \) is part of some key for \( R \).

Thus the definition of 3NF permits a few additional FDs involving key attributes that are prohibited by BCNF.

Properties of 3NF

The most important property of 3NF is the result that there is always a lossless-join and dependency-preserving decomposition of any relational schema into 3NF.

While not conceptually difficult, the algorithm to decompose a relation schema into 3NF is quite fiddly and much more complicated than the simple algorithm for decomposition into BCNF.

The contracts example

The contracts example was not in BCNF because of the FD

\[ SD \rightarrow P \]

However as \( P \) is part of a key (\( JP \)), this FD does not violate the conditions for 3NF. Therefore neither of the two given FDs violate the conditions for 3NF.

Checking all the other (implied) FDs shows that contracts is already in 3NF and thus need not be decomposed further.

3NF Synthesis

The BCNF algorithm is from the perspective of deriving a lossless-join decomposition, and dependency preservation was addressed by adding extra relation schemas.

An alternative approach involves synthesis that takes the attributes of the original relation \( R \) and the minimal cover \( F \) of the FDs and adds a relation schema \( XA \) for each FD \( X \rightarrow A \) in \( F \).

The resulting collection of relations is 3NF and preserves all FDs. If it is not lossless-join, it is made so by adding a relation that contains those attributes that appear in some key.
The two approaches

The decomposition algorithm derives the relations while maintaining the keys, thus emphasizing the lossless-join property, and deals with dependency preservation by adding the necessary relations.

The synthesis algorithm builds the relations from the set of FDs, emphasizing the dependencies, and then deals with lossless-join issues after the fact by adding the necessary relation.

The latter is an algorithm that has polynomial complexity.

The Synthesis Algorithm

Given a relation \( R \) and a set of functional dependencies \( X \rightarrow Y \) where \( X \) and \( Y \) are subsets of the attributes of \( R \).

1. Compute the closures of the FDs.
2. Use ➀ to find all the candidate keys of the relation \( R \).
3. Find the minimal cover of the relation \( R \).
4. Use ❼ to produce \( R_i \) decompositions in 3NF.
5. If no \( R_i \) is a super key of \( R \), add a relation containing the key.

Candidate keys

1. Use ➀ to find all the candidate keys of the relation \( R \).

If any \((X)^+\) covers all the attributes of \( R \) then \( X \) is a key. It is possibly a super key with redundant attributes contained within \( X \) (these will be obvious). Otherwise find the \((X)^+\) that maximally covers \( R \) and add to \( X \) attributes of \( R \) missing from the closure.

FD closures

- Compute the closures of the FDs.

For each \( X \rightarrow Y \), derive \((X)^+\) the set of attributes of \( R \) covered by \( X \rightarrow Y \).
Minimal cover

Find the minimal cover of the relation $R$.

**Algorithm:**

1. **Step 1** Decompose all the FDs to the form $X \rightarrow A$, where $A$ is a single attribute.
2. **Step 2** Eliminate redundant attributes from the LHS.
   
   If $XB \rightarrow A$, where $B$ is a single attribute and $X \rightarrow A$ is entailed within the set, then $B$ was unnecessary.
3. **Step 3** Delete the redundant FDs.

3NF relations

Use ① to produce $R_i$ decompositions in 3NF.

**Algorithm:**

1. **Step 1** Combine all FDs with the same LHS.
   
   For each $X$ such that $X \rightarrow A$, $X \rightarrow B$ ...
   
   $⇒ X \rightarrow AB$...
2. **Step 2** Form a relation $R_i$ of all attributes in each FD.
   
   For each $X \rightarrow ABC$, form
   
   $⇒ \{X, A, B, C\}$.

Exercise

Consider $R = \{A, B, C, D, E, G\}$ and the following FDs

1. $AB \rightarrow C$
2. $C \rightarrow A$
3. $BC \rightarrow D$
4. $ACD \rightarrow B$
5. $D \rightarrow EG$
6. $BE \rightarrow C$
7. $CG \rightarrow BD$
8. $CE \rightarrow AG$

If no $R_i$ is a super key of $R$, add a relation containing the key.

If $UVW$ is a key for $R$, and no relation $R_i$ formed in the 3NF decomposition contains a super key of $R$, then add $\{U, V, W\}$ as a relation.

Applying Armstrong’s Axioms gives $BD$ and $CD$ are also keys.

- $AB$ is a key
- $BC$ is a key
- $ACD$ is a super key
- $BE$ is a key
- $CG$ is a key
- $CE$ is a key
Generate minimal cover

The synthesis process

Final 3NF decomposition

Manual normalization

Data typically employed in a business is usually presented as below.

However to be useful in a database the data needs to be normalized.
### First normal form – 1NF

First normal form (1NF) requires attribute values be atomic — that is, individual values rather than lists or sets and that a primary key is defined.

<table>
<thead>
<tr>
<th>Order_ID</th>
<th>Order Date</th>
<th>CustomerID</th>
<th>Customer Name</th>
<th>Customer Address</th>
<th>Product_ID</th>
<th>Product Description</th>
<th>Material</th>
<th>Unit Price</th>
<th>Ordered Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>04/05/2010</td>
<td>2</td>
<td>FurnCo</td>
<td>35 5 St, NY</td>
<td>21</td>
<td>Sofa</td>
<td>Leather</td>
<td>2500.00</td>
<td>1</td>
</tr>
<tr>
<td>1234</td>
<td>04/05/2010</td>
<td>2</td>
<td>FurnCo</td>
<td>35 5 St, NY</td>
<td>9</td>
<td>TV Bench</td>
<td>Pine</td>
<td>300.00</td>
<td>4</td>
</tr>
<tr>
<td>6789</td>
<td>05/05/2010</td>
<td>9</td>
<td>CheapAs</td>
<td>35 N St, LA</td>
<td>17</td>
<td>Chair</td>
<td>Mahogany</td>
<td>500.00</td>
<td>6</td>
</tr>
<tr>
<td>6789</td>
<td>05/05/2010</td>
<td>9</td>
<td>CheapAs</td>
<td>35 N St, LA</td>
<td>10</td>
<td>Desk</td>
<td>Particle</td>
<td>650.00</td>
<td>1</td>
</tr>
</tbody>
</table>

This table is in 1NF. The primary key is (Order ID, Product ID).

---

### Second normal form – 2NF

2NF is achieved by creating three (3) relations with the following attributes and keys.

A relation in first normal form (1NF) has the above functional dependencies.

A relation is in Second normal form (2NF) if it is in 1NF and contains no partial dependencies.

A partial functional dependency exists when a nonkey attribute is functionally dependent on part of, but not all of, the primary key.

---

### Third normal form – 3NF

A relation in second normal form (2NF) still has the above functional dependencies.

A relation is in Third normal form (3NF) if it is in 2NF and contains no transitive dependencies.

A transitive functional dependency in a relation is a functional dependency between two or more nonkey attributes.
Third normal form – 3NF

3NF is achieved by creating an additional relation with the following attributes and keys.

Practical Use of Normalization

In practice, the DBA should be guided by the theory of normalization, but not slavishly follow it.

When the ER diagram is converted into tables, any functional dependencies should be identified and a decision made as to whether the tables should be normalized or not.

Occasionally it will even be appropriate to de-normalize data if the most frequently occurring queries involve expensive joins of the normalized tables.

Multi-valued Attributes

A Single-valued attribute of an entity instance has just one value. This has been the typical scenario we have discussed so far and know how this is handled in a schema definition.

A Multi-valued attribute is an attribute that can take on more than one value for a given entity instance. Diagrammatically this is indicated in an ERD by a double edged ellipse.

The first relation contains all the attributes of the entity type except the multi-valued attribute. The second relation contains the attributes that form the primary key of the second relation. The first of these attributes is the primary key of the first relation, and hence it is a foreign key.
This lecture describes how to enforce constraints and execute triggers.

Active Databases

Working databases

SQL allows one to create “active” elements in a database. An *active* element in an expression or statement that is stored in the database and executed at the appropriate “time”.

The *time* may be when an insertion, update or deletion on a relation is attempted, or equally if changes result in a *boolean* value being TRUE.

We have discussed some *active* elements already. Key constraints (PRIMARY/FOREIGN) are examples of statements that have effect when changes are made to the database.

Why and where

Updates to databases can be wrong in a variety of ways. For example,

- There may be typographical and transcription errors
- Data values may be outside of an allowed range
- Data values have an interrelationship

These constraints can be written at the application level, however it is better to store these checks in the database and let the DBMS administer them. Duplication is avoided and the check is never “forgotten”.

Support for integrity constraints is present in most DBMSs, but support for “checks”, “assertions” and “triggers” is considerably weaker.
Primary Keys

An attribute or set of attributes of a relation that are a key are identified with the keywords PRIMARY KEY or UNIQUE.

Collectively the value(s) of the attributes of the key are unique, and thereby uniquely identify the tuple (row) of the relation (table).

Any update that effects the values of the key, or violate the key integrity will result in an error state for the DBMS.

Foreign keys

Referential-integrity constraints can be declared with the key word FOREIGN KEY. This constraint asserts that the value appearing in one relation must also appear as the primary-key component(s) of another relation.

Consider the following schema

```sql
CREATE TABLE S ( ps INT ... FOREIGN KEY (ps) REFERENCES M(pm) );
```

Foreign key exceptions

A runtime exception to referential-integrity will occur when we

- insert an S tuple, with a ps that is a not NULL value and is not the pm component of any M tuple
- update an S tuple to change the ps to a non NULL value that is not the pm component of any M tuple
- delete an M tuple and its pm component, which appears as the ps component of some S tuple
- update an M tuple that changes a pm component, which is the same as the ps component of some S tuple

Not-Null constraints

A simple attribute constraint is NOT NULL. The effect is to disallow tuples in which the particular attribute is NULL.

This use of NOT NULL can be important for UNIQUE attributes. In the case of UNIQUE attributes, NULL values are “accepted and ignored”.

This means NULL is considered a legitimate value AND multiple instances of NULL values for an attribute is not considered a violation of uniqueness.

This can be avoided by declaring the attribute to be NOT NULL.
Attribute constraints

CHECK constraints - attribute

More complex constraints can be attached to an attribute by use of the keyword CHECK followed by a conditional expression which must hold for every value of that attribute.

Such CHECKs are likely to be limits on values, arithmetic inequalities, or restrictions to an enumeration of values (ENUM also achieves this).

The CHECK is executed whenever a tuple is assigned a new value for the attribute to which the CHECK is attached.

The constraint is checked only when changes occur to the attribute to which the constraint is associated. It is possible for this constraint to be violated if other values involved in the constraint are changed.

Example attribute-based CHECK constraints

p INT CHECK( p >= 0 AND p <= 100)
Require the value of p is an integer between 0 and 100 inclusive.

gender CHAR(1) CHECK( gender IN ('F', 'M'))
Has the same effect as an ENUM definition.

ps INT CHECK( ps IN (SELECT pm FROM M))
An erroneous simulation of referential-integrity maintenance.

Tuple constraints

CHECK constraints - tuple

If we declare a CHECK constraint in a schema at the same level as an attribute declaration, then that is interpreted as a condition on the tuple of the relation. The attributes of the relation may be referred to in the expression.

CREATE TABLE M ( title CHAR(5),
  ..., gender CHAR(1),
  dob DATE,
  CHECK(gender = 'F' OR title NOT LIKE 'Ms.%') );

MySQL parses but ignores CHECK constraint declarations.

More powerful active elements

The previous active elements were associated with tuples or components of tuples. More powerful elements are associated with neither, and are on a par with tables. These are assertions and triggers.

An assertion is a boolean-valued SQL expression that is always true.

A trigger is a series of actions associated with certain events.

Assertions are easy to define, very difficult to implement. The DBMS must determine if modifications affect an assertion’s truth - declarative

Triggers specifically identify what the DBMS needs to do - procedural
Assertions

CREATE ASSERTION <a-name> CHECK (<condition>)

The condition must be true when the assertion is created and must remain true or the database modification is rejected.

CREATE ASSERTION TooRich CHECK {
  (NOT EXISTS
  (SELECT company-name
  FROM Company, Executives
  WHERE ceo = eid AND networth < 10000000
  )
  );
}

Ensure that no company has as its CEO someone whose net worth is less than $10,000,000.

MySQL does not implement assertions.

Triggers

Triggers are also known as event-condition-action or ECA rules.

- Triggers are awakened by certain programmer specified events, e.g. insert, delete or update on a relation or on a transaction end.
- Triggers test a condition. If the condition does not hold, nothing else is done.
- If the condition is satisfied, the action associated with the trigger is performed. It can modify the effects of the event, or even abort the transaction. It can perform any sequence of DB operations that may have nothing to do with the particular event.

SQL Triggers

Features of an SQL trigger are;

- The check of the condition and the action may be executed on the state of the database before the triggering event is executed, or the state that exists after the event is executed.
- The condition an action can refer to the old and the new values of tuples after an update.
- It is possible to limit update events to a certain attribute or set of attributes.
- A trigger may execute once for each modified tuple (row-level trigger) or once for all tuples that are changed by an SQL statement (statement-level trigger).

Example Trigger

CREATE TRIGGER NetWorthTrigger
AFTER UPDATE of netWorth on MovieExec
REFERENCING OLD ROW as OldTuple,
NEW ROW as NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
UPDATE MovieExec
SET netWorth = OldTuple.netWorth
WHERE cert = NewTuple.cert;

This trigger is designed to foil (by undoing) any update on an attribute (netWorth) with a lower value.

There is limited support for triggers in MySQL, but restricted to users with SUPER privileges. As of Version 5.1.6 triggers can be created and dropped when appropriate privileges are set.
This lecture introduces the concepts underlying database transactions.

Concurrency

One of the most important properties of a modern DBMS is its ability to manage multiple client sessions simultaneously and transparently to the users of the database.

It is important for a DBA to understand how such concurrency control is managed by the database as it can have a significant impact on the overall performance of the database.

Transactions

We define a transaction as any one execution of a user program.

In this context, a user program consists of a number of statements that read and write database objects (i.e. tables, values etc), before finally committing at which point any changes to the state of the DB are made permanent (i.e. written to disk).

For certain applications, it is critical that all the statements in a transaction run to completion without interference from other users.
Concurrency

Bank Transfers

The “canonical example” of an application where correct treatment of transactions is critical is transferring money in a bank.

For example, suppose that a user at an ATM transfers money between two accounts.

```sql
UPDATE accounts SET balance = balance - 500
WHERE id = 1;
UPDATE accounts SET balance = balance + 500
WHERE id = 2;
```

It is crucial that either both statements occur or neither do — for example, if the computer crashes after the first one has occurred then the system must be able to recognize and recover from that.

(GF Royle, N Spadaccini 2006-2010)

Databases - Transactions I

ACID

Transaction Properties

A transaction-safe database engine must ensure that the following four properties — known by the acronym ACID — are maintained.

- Atomicity
- Consistency
- Isolation
- Durability

ACID

Atomicity

The word atomic is used in a number of contexts to denote indivisible. In a DB context, transactions are atomic if the system ensures that they cannot be “half-done” — in other words, the user is guaranteed that either the entire transaction completes or it fails and has no effect on the database.

The bank transfer example above is one application where users would rely on the atomicity of transactions.

ACID

Consistency

Transactions must preserve the consistency of the database.

More precisely, if the database is in a consistent state, and a transaction is executed to completion on its own (i.e. with no concurrently executing transactions) then the state of the database after the transaction should also be consistent.

This is basically a fancy way of saying that the user’s programs should be correct. Transaction consistency is therefore the responsibility of the user not the DBMS.
Isolation

*Isolation* means that the user of the DB should be able to execute a transaction without regard for concurrently executing transactions. In other words, the user’s actions should be isolated from the actions of other users — at least for the duration of the transaction.

Durability

*Durability* means that once the user is informed of the successful completion of a transaction, then its effects on the database are persistent. Thus the user should be shielded from any possible problems (e.g., system crashes) that might occur after being notified that the transaction has successfully completed.

Atomicity and Durability

Ensuring *atomicity* requires the DBMS to be able to *undo* the effect of earlier statements if the entire transaction is aborted, either by the DBMS itself (if a later statement fails) or for some external reason (system crash, power cut etc).

The basic mechanism used for this is that the DBMS maintains a *log* of all changes to the database. Every action that causes a change to the state of the database is *first* recorded in the log file, which is then saved to disk. Finally the new state of the database is written to disk.

Write Ahead Log

The property that changes are *logged* before they are actually made on disk is called *write ahead log*. If a transaction is aborted, then the DBMS can consult the log in order to determine which actions need to be undone in order to restore the database to its initial state.

In the case of a system crash, the *recovery manager* uses the log to determine whether there are any completed transactions that still need to be written to disk.

Complete details of the logging process and the recovery manager are complicated and require a detailed understanding of the physical aspects of computers.
Interleaving

There would be no problem with isolation if the DBMS were able to simply run each transaction to completion at a time before starting the next one.

However in practice, it is vital to *interleave* the actions of transactions in order for the system to be usable in practice.

![Action Diagram]

Notation

We use the notation $R(O)$ and $W(O)$ to indicate the actions of reading a database object $O$ and writing a database object $O$.

Then a transaction can be considered to be a sequence of reading and writing actions ending when the transaction commits.

![Notation Diagram]

Motivation for Interleaving

Interleaving transactions (properly) allows multiple users of the database to access it at the same time.

While one transaction is performing an I/O task, another one can perform a CPU-intensive task thus maximising the *throughput* of the system.

Strict serial execution of transactions would be impractical because large numbers of short transactions would become "queued up" behind a long running transaction waiting for it to finish.

Thus managing a collection of interleaved transactions is a fundamental task for a DBMS.

Interleaving Anomalies

There are a variety of *anomalies* that can arise from an unfortunate choice of *schedule* for interleaved transactions.

Each of these anomalies could leave the database in an inconsistent state that could not arise if the two transactions were not interleaved.

- Dirty Reads
- Nonrepeatable Reads
- Phantoms
Dirty Reads

A dirty read occurs when one transaction reads a database value that has been altered by a transaction that has not yet committed.

Two major problems can arise from dirty reads:

- The database may be in a temporarily inconsistent state due to the partially completed transaction.
- The partially completed transaction may subsequently be aborted restoring the value to its original state.

Dirty Read Example

Suppose that $T_1$ transfers $100 from account $A$ to account $B$, while $T_2$ adds 5% interest to each account, and the following schedule is used.

<table>
<thead>
<tr>
<th>Time</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$R(A)$</td>
<td>$R(A)$</td>
</tr>
<tr>
<td></td>
<td>$W(A)$</td>
<td>$W(A)$</td>
</tr>
<tr>
<td></td>
<td>$W(A)$</td>
<td>$R(B)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$W(B)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Dirty Read Example cont.

If $A$ and $B$ have a $1000 balance initially then this schedule would proceed as follows:

- $T_1$ deducts $100 from $A$ so balance is $900$.
- $T_2$ adds 5% interest to $A$ so balance is $945$.
- $T_2$ adds 5% interest to $B$ so balance is $1050$.
- $T_1$ adds $100 to $B$ so balance is $1150$.

Neither of the two possible serial schedules (i.e. $T_1$ first, then $T_2$ or vice versa) would give these values, and in fact $5 interest has been lost.

The fundamental problem is that $T_1$ put the DB into an inconsistent state, and $T_2$ used the inconsistent values before $T_1$ could restore the DB.

Unrepeatable Reads

An unrepeatable read is essentially the dirty-read problem in reverse order in that a value gets changed by another transaction after it has been read, rather than before.

In this situation, transaction $T_1$ reads a value which is then changed by $T_2$. If $T_1$ subsequently re-reads the value then it gets a different value, even though it hasn’t changed it.

This violates the isolation property because transaction $T_1$ should be able to complete as though it is the only transaction currently executing.
Phantoms

A *phantom* is a variant of the unrepeateable read problem that occurs when one transaction performs a `SELECT` statement with some selection criteria, and then subsequently another transaction inserts a new row.

If the first transaction now uses the same criteria again for a subsequent `SEARCH` or `UPDATE` statement, then a new row will suddenly appear, known as a *phantom* row.

This lecture

This lecture discusses how a DBMS schedules interleaved transactions to avoid the anomalies described in the lecture Transactions I.

Schedules

A schedule (of interleaved statements) is called *serializable* if its effect on any consistent database instance is equivalent to running the transactions in *some* serial order.

A schedule is called *recoverable* if a transaction $T_1$ that reads values changed by $T_2$ only commits *after* $T_2$ commits.

The job of the DBMS is to ensure that the only allowed schedules are *serializable* and *recoverable*.
Serializable . . .

This schedule is *serializable* because if both transactions commit as shown, then the effect is the same as running T₁ and then T₂.

<table>
<thead>
<tr>
<th>T₁</th>
<th>Time</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td>R(C)</td>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td>W(C)</td>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

but not recoverable

This schedule is not recoverable because if T₁ aborts, then the earlier statements need to be undone; however T₂ has already used the value of A and committed.

<table>
<thead>
<tr>
<th>T₁</th>
<th>Time</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td>R(C)</td>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td>W(C)</td>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td>Abort</td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Locking

The main way in which a DBMS ensures that only serializable, recoverable schedules are allowed is through *locking protocols*.

A *lock* is a flag, or indicator, that can be attached to a database object indicating that it is in use by a transaction; a second transaction wishing to use the same DB object may have to wait until the first transaction has finished it.

A *locking protocol* is a set of rules that determine what types of lock to use in particular situations.

Strict Two-Phase Locking

The most widely used locking protocol is *Strict Two-Phase Locking* (Strict 2PL) which uses two rules.

- A transaction that wishes to *read* an object first requests a *shared lock* on that object, while a transaction that wishes to *modify* an object first requests an *exclusive lock* on that object.
- All locks held by a transaction are released when the transaction completes (commits or aborts).
Altering a schedule

Now consider the non-recoverable schedule from above with Strict 2PL.

<table>
<thead>
<tr>
<th>Time</th>
<th>Transaction T1</th>
<th>Transaction T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(C)</td>
<td>W(C)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>S(A)</td>
<td></td>
<td>blocks</td>
</tr>
</tbody>
</table>

Abort
Commit

Blocking

When transaction $T_2$ requests a shared lock (to read $A$) the DBMS cannot grant that until $T_1$ releases its exclusive lock. Therefore $T_2$ is blocked until $T_1$ completes — either by committing or aborting.

Thus the non-recoverable schedule is disallowed under the operation of Strict 2PL.

Deadlock

A transaction $T_1$ is blocked when it is waiting for another transaction, say $T_2$ to release a lock.

But what happens if $T_2$ is also blocked because it is waiting for $T_1$ to release a lock?

In this situation, the two (or more) transactions are deadlocked and without DBMS intervention they would both remain blocked indefinitely.

The DBMS must therefore determine when a transaction is deadlocked and abort it. Detecting deadlock can be done by maintaining an explicit list of blocked transactions and who they are waiting for and then detecting cycles in that list. Alternatively a simpler mechanism is simply to abort any transaction that has been blocked for too long, assuming that it is deadlocked.

Performance

Because it causes some transactions to be blocked, any locking protocol will cause some degradation in performance.

If there are only a few transactions active, then adding another one will cause the total throughput to increase as interleaving allows it to use CPU idle time. As more transactions are added, some blocking occurs and so the increase in throughput is less. Eventually, each additional transaction conflicts with so many others that the total throughput is reduced. At this point the system is said to be thrashing.
**Locking**

**Diminishing returns**

[Graph showing the relationship between throughput and active transactions]

Locking performance can be substantially affected by the *granularity* of the locking protocol. For example, suppose that a transaction is going to modify a few rows in a large table — a coarse-grained locking protocol would lock the entire table, while a fine-grained locking protocol would lock just the individual rows that will be affected.

**Transactions in MySQL**

In **MySQL** it is necessary to use a transaction-safe database engine — at the current time (2009) that means using either InnoDB or BDB (Berkeley DB).

The default database engine, MyISAM does not support transactions — in effect it behaves as though each statement were a single-statement transaction.

In particular, with MyISAM it is not possible to guarantee the atomicity of more than a single statement.

**InnoDB**

The InnoDB storage engine *does* support transactions. A transaction can be initiated by a user with the statement

\[
\text{START TRANSACTION;}
\]

All statements after that will be deemed to form part of the same transaction until one of the statements

\[
\text{COMMIT;}
\]

or

\[
\text{ROLLBACK;}
\]

occurs.
Example

Suppose students are forming project *groups* and being entered into the following database table.

```sql
CREATE TABLE groups(
    id INT,
    sNum INT,
    UNIQUE(sNum)) Engine = InnoDB;
```

The **UNIQUE** keyword will ensure that no student accidentally ends up allocated to more than one group.

Initial data

Suppose that the first group has been entered:

```sql
SELECT * FROM groups;
+------+------+| id | sNum | +------+------+| 1 | 1537 | | 1 | 1433 | +------+------+
```

and that the second proposed group comprises students 1010 and 1537; of course 1537 is already in a group and so adding this group should fail.

Use a transaction

```sql
START TRANSACTION;
INSERT INTO groups VALUES(2,1010);
INSERT INTO groups VALUES(2,1537);
ERROR 1062 (23000): Duplicate entry '1537' for key 1
```

The first student was correctly added, but the second violates the key constraint — now the entire transaction can be aborted by issuing the **ROLLBACK** statement.

```sql
ROLLBACK;
SELECT * FROM groups;
+------+------+| id | sNum | +------+------+| 1 | 1537 | | 1 | 1433 | +------+------+
```

Rolling Back

In addition to explicitly issuing a **ROLLBACK** statement, this will happen **automatically** if the client becomes disconnected before the transaction is committed. Therefore critical transactions will never get left half-done.

There is also the facility to **partially** roll-back a transaction to an intermediate "checkpoint" that is declared inside a transaction.

```sql
SAVEPOINT doneOne;
```

If something goes wrong in the next few statements, the user can

```sql
ROLLBACK TO SAVEPOINT doneOne;
```

rather than rolling back the entire transaction.
Isolation Levels

MySQL permits the user to choose how “isolated” they wish each transaction to be by choosing between

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

The user can set the isolation level on a per-session or even per-transaction basis, using statements such as:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED
```

Isolation level Summary

This table summarizes the anomalies that can or cannot arise at the different isolation levels.

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>No³</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

³MySQL-specific

Databases - Data Mining
This lecture

This lecture introduces data-mining through market-basket analysis.

Data Mining

An organization that stores data about its operations will rapidly accumulate a vast amount of data.

For example, a supermarket might enter check-out scanner data directly into a database thus getting a record of all purchases made at that supermarket.

Alternatively data might be collected for the express purpose of data mining. For example, the Busselton Project is a longitudinal study that has accumulated 30-years of health-related data about the people in Busselton. (See http://bsn.uwa.edu.au.)

Data Mining, or more generally Knowledge Discovery in Databases (KDD) refers to the general process of trying to extract interesting or useful patterns from a (usually huge) dataset.

Data Mining Research

Research into data mining is one of the most active areas of current database research, with a number of different aspects:

- KDD techniques
  Research into the theoretical statistical techniques underlying KDD, such as regression, classification, clustering etc.

- Scalability
  Research into algorithms for these techniques that scale effectively as the data volume reaches many terabytes.

- Integration
  Research into integrating KDD tools into standard databases.

Rules

One of the fundamental types of interesting pattern is to identify associations between observations that might reflect some important underlying mechanism.

For example, the Busselton Project may find associations between health-related observations: perhaps a correlation between elevated blood pressure at age 30 and the development of Type 2 diabetes at age 50.
Market Basket Analysis

We will only consider one simple technique, called market basket analysis, for finding association rules.

A market basket is a collection of items associated with a single transaction.

The canonical example of market basket analysis is a supermarket customer who purchases all the items in their shopping basket.

By analysing the contents of the shopping basket one may be able to infer purchasing behaviours of the customer.

The aim of market basket analysis is to analyse millions of transactions to try and determine patterns in the items that are purchased together.

This information can then be used to guide specials, display layout, shop-a-docket vouchers, catalogues and so on.

Sample Data

<table>
<thead>
<tr>
<th>transID</th>
<th>custID</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>201</td>
<td>pen</td>
</tr>
<tr>
<td>111</td>
<td>201</td>
<td>ink</td>
</tr>
<tr>
<td>111</td>
<td>201</td>
<td>milk</td>
</tr>
<tr>
<td>111</td>
<td>201</td>
<td>juice</td>
</tr>
<tr>
<td>112</td>
<td>105</td>
<td>pen</td>
</tr>
<tr>
<td>112</td>
<td>105</td>
<td>ink</td>
</tr>
<tr>
<td>112</td>
<td>105</td>
<td>milk</td>
</tr>
<tr>
<td>113</td>
<td>106</td>
<td>pen</td>
</tr>
<tr>
<td>113</td>
<td>106</td>
<td>milk</td>
</tr>
<tr>
<td>114</td>
<td>201</td>
<td>pen</td>
</tr>
<tr>
<td>114</td>
<td>201</td>
<td>ink</td>
</tr>
<tr>
<td>114</td>
<td>201</td>
<td>juice</td>
</tr>
</tbody>
</table>

(Taken from Section 26 of R & G.)

Terminology

An itemset is a set of one or more items: for example \{pen\} is an itemset, as is \{milk, juice\}.

The support of an itemset is the percentage of transactions in the database that contain all of the items in the itemset.

For example:

- The itemset \{pen\} has support 100% \(\{\text{pen}\}\) are purchased in all 4 transactions
- The itemset \{pen, juice\} has support 50% \(\{\text{pen, juice}\}\) are purchased together in 2 of the 4 transactions
- The itemset \{pen, ink\} has support 75% \(\{\text{pen, ink}\}\) are purchased together in 3 of the 4 transactions
Frequent Itemsets

The first step of mining for association rules is to identify frequent itemsets — that is, all itemsets that have support at least equal to some user-defined minimum support.

In this example, setting minimum support to 70% we would get

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>{pen}</td>
<td>100%</td>
</tr>
<tr>
<td>{ink}</td>
<td>75%</td>
</tr>
<tr>
<td>{milk}</td>
<td>75%</td>
</tr>
<tr>
<td>{pen, ink}</td>
<td>75%</td>
</tr>
<tr>
<td>{pen, milk}</td>
<td>75%</td>
</tr>
</tbody>
</table>

Finding Frequent Itemsets

In a toy example like this, it is simple to just check every possible combination of the items, but this process does not scale very well!

However it is easy to devise a straightforward algorithm based on the a priori property

Every subset of a frequent itemset is also a frequent itemset.

The algorithm proceeds by first finding single-element frequent itemsets and then extending them, element-by-element until they are no longer frequent.

Sample run

The first scan of our sample relation yields the three itemsets

{pen}, {ink}, {milk}

In the second step we augment each of these by an additional item that is itself a frequent item, and then check each of the itemsets

{pen, ink}, {milk, ink}, {pen, milk}

thereby determining that

{pen, ink}, {pen, milk}

are frequent itemsets.

Sample run cont.

Now the only possible 3-item itemset to check would be

{pen, ink, milk}

but this can be rejected immediately because it contains a subset

{milk, ink}

that is not itself frequent.

With a huge database, the scan to check the frequency of each candidate itemset dominates the time taken and hence eliminating an itemset without a scan is very useful.
Association rules

An association rule is an expression such as

\{pen \Rightarrow \text{ink}\}

indicating that the occurrences of “ink” in a transaction are associated with the occurrences of “pen”.

The overall aim of market basket analysis is to try to find association rules in the data. If an association rule extracted from the data represents a genuine pattern in shopper behaviour, then this can be used in a variety of ways.

Although the terminology is all about shopper behaviour, the concepts can easily be translated to more “significant” projects such as trying to associate behaviour or diet with disease or mortality.

Support of a rule

Suppose that $X$ and $Y$ are itemsets. Then the support of the association rule

$X \Rightarrow Y$

is the support of the itemset $X \cup Y$.

Thus the support of the rule

$\{pen \Rightarrow \text{ink}\}$

is 75%.

Normally market basket analysis is only concerned with association rules involving frequent itemsets, because while there may be a very strong association between, say, lobster and champagne, this will not form a large proportion of sales.

Confidence of a rule

The confidence of a rule $X \Rightarrow Y$ is the proportion of transactions involving $X$ that also involve $Y$.

In other words, if $s(X)$ denotes the support of $X$, then the confidence is

\[\frac{s(X \cup Y)}{s(X)}\]

Thus in our example,

$\{pen \Rightarrow \text{ink}\}$

has a confidence level of

\[\frac{75}{100} = 75\%\]

whereas the rule

$\{\text{ink} \Rightarrow \text{pen}\}$

has confidence 100%.

Beer and Nappies

One of the most well-known marketing stories is the (possibly apocryphal) story of beer and nappies.

A Wal-Mart manager noted one Friday that a lot of customers were buying both beer and nappies. Analysing past transaction data showed that while beer and nappies were not particularly associated during the week, there was a sudden upsurge in the association on Friday evenings.

Thinking about why there might be this association, the manager concluded that because nappies are heavy and bulky, the job of buying nappies was often left to fathers who picked them up after work on Fridays, and also stocked up on beer for the weekend.
Cross-selling

The manager responded to this information by putting the premium beer displays and specials right next to the nappy aisle.

The fathers who previously bought regular beer were now encouraged to buy the premium beer, and some of the fathers who hadn’t even thought about beer started to buy it.

This version of the story paraphrased from http://www.information-drivers.com/market_basket_analysis.htm.

Interpreting association rules

The 100% confidence level indicates that the data shows that if shoppers buy ink, then they always buy a pen as well.

How should an association rule of this type be interpreted?

Clearly there is a high correlation between buying pens and buying ink. When faced with a high correlation, it is tempting, but incorrect, to assume that the rule indicates a causal relationship.

“Buying ink causes people to buy pens”

Example scenario

A Pizza restaurant records the following sales for pizzas with extra toppings, in various combinations. The toppings are mushrooms (M), pepperoni (P) and extra cheese (C).

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Pizza Sales</th>
<th>Extra Toppings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>150</td>
<td>P</td>
</tr>
<tr>
<td>3</td>
<td>200</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>400</td>
<td>M &amp; P</td>
</tr>
<tr>
<td>5</td>
<td>300</td>
<td>M &amp; C</td>
</tr>
<tr>
<td>6</td>
<td>200</td>
<td>P &amp; C</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>M, P &amp; C</td>
</tr>
<tr>
<td>8</td>
<td>550</td>
<td>None</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2000</strong></td>
<td></td>
</tr>
</tbody>
</table>

Required analysis

Complete a market-basket analysis to answer the following questions:

- Find the frequent itemsets with minimum support 40%.
- Find association rules with minimum confidence 50%.
- What is the strongest inference we can make about consumer behaviour when choosing extra toppings?
### Support - single item sets

- **Find the frequent itemsets** with minimum support 40%.

  $\{M\} = \frac{100 + 400 + 300 + 100}{2000} = 45\%$

  $\{P\} = \frac{150 + 400 + 200 + 100}{2000} = 42.5\%$

  $\{C\} = \frac{200 + 300 + 200 + 100}{2000} = 40\%$

### Support - two item sets

- **Find the frequent itemsets** with minimum support 40%.

  $\{M, P\} = \frac{400 + 100}{2000} = 25\%$

  $\{M, C\} = \frac{300 + 100}{2000} = 20\%$

  $\{P, C\} = \frac{200 + 100}{2000} = 15\%$

### Rule confidence

- **Find association rules with minimum confidence** 50%.

  $M \rightarrow P = \frac{\{M, P\}}{\{M\}} = \frac{25}{45} = 55.6\%$

  $P \rightarrow M = \frac{\{M, P\}}{\{P\}} = \frac{25}{42.5} = 58.8\%$

  $C \rightarrow M = \frac{\{M, C\}}{\{C\}} = \frac{20}{40} = 50\%$

All other associations are < 50%.

### Inference

- **What is the strongest inference we can make about consumer behaviour when choosing extra toppings?**

  People who order a pizza with extra pepperoni are likely to order extra mushrooms.