Lecture 1 — Introduction

What is Database?

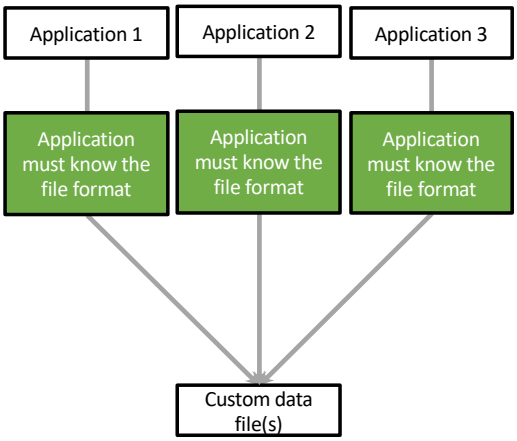
Database: “Organized collection of data. Structured, arranged for ease and speed of search and retrieval.”

Database Management System (DBMS):

Designed scenario: Software that is designed to enable users and programs to store, retrieve and update data from data base.

Create/Modify/Access: A software must have a set of standard functions to be called DBMS

What do we need DBMS? —



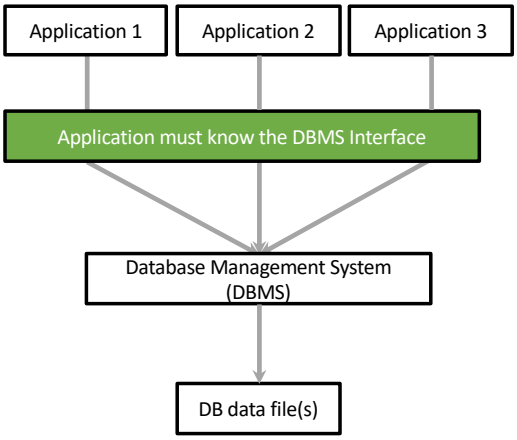

Pre-DBMS Methods

Applications stores data as files, each application is its own format, other applications need to understand the same format to be able to understand.

Leads to duplicated code, and thus waster effort. Also causes compatibility issues.

For common data format duplicated codes is still needed to read the different file formats.

There also exists synchronization issues that occurs when the data is accessed simultaneously, complicating the coordination throughout different applications.




DBMS Approach

Work as a delegate for the common collection of data, provides a common API for applications (such as a C library header file or Java jar class file that defines the interface to interact with the DBMS).

Advantage is that when there is multiple applications interacting with the DBMS the interaction will be coordinated

Commonly Seen DBMS

Oracle
DB2
MySQL
Ingres
PostgreSQL
Microsoft SQL Server
MS Access

Database Applications

Example 1: Membership cards of a chain store.
Example 2: Banking services, account balance.

DBMS Functions (Must have functions)

Allow users to store, retrieve and update data
Ensure either that all the updates corresponding to a given action are made that none of them is made (Atomicity)
Ensure that DB is updated correctly when multiple users are updating it concurrently
Recover the DB in the event it is damaged in any way
Ensure that only authorized users can access the DB
Be capable of integrating with other software

The Relational Model — and the relational database management systems (RDBMS)

## The Relational Model

The relational model is **one approach** to managing data. Originally introduced by E.F. Codd in his paper "A Relational Model of Data for Large Shared Databanks", 1970.

- An earlier model is called the navigation model.
- RDBMS are based on the relational model.

The model uses a structure and language that is consistent with **first-order predicate logic**

- Provides a declarative method for specifying data and queries
- Details are covered in the Chapter 4 of the textbook.

## Relational Model: Terminology* (* = the knowledge that is needed throughout the module)

A **relation** is a mathematical concept. The physical form of a relation is a table with columns and rows.

A **attribute** is a named column of a relation

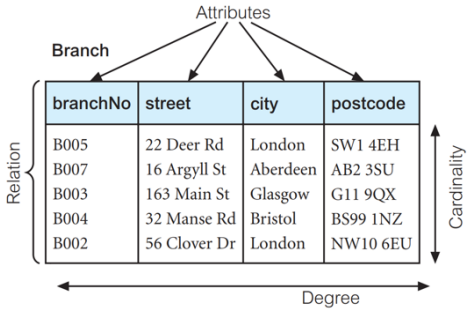
A **domain** is the set of allowable values for attributes

- Age must be a positive integer.
- Post codes have length limit.

A **tuple** is a row of relation. (Order of tuples does not matter)

The **degree** of a relation is the number of attributes it contains.

The **Cardinality** is the number of tuples in a relation.



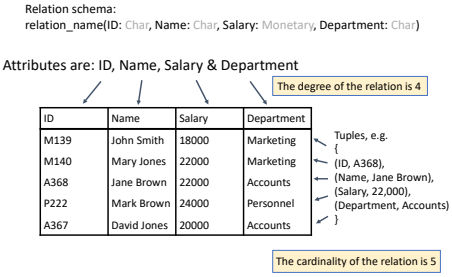

## Terminology*

**Relation schema:**

- The definition of a relation, which contains the name and domain of each attribute
- Formally "A named relation defined by a set of attributes and domain name pairs"

**Relational database schema:**

- A set of relation schemas each with a distinct name.




## Alternative terminologies

| | | |
|---|---|---|
| Relation | == table, | file |
| Tuple | == row, | record |
| Attribute | == Column , | Field |

## Relation: Properties

A relation also has the following properties:

- Its name is unique in the relational database scheme.
- Each cell contains exactly one atomic value.
- Each attribute of a relation must have a distinct name.
- The values of an attribute are from the same domain.
- No duplicate tuples.
- The order of attributes has no significance.
- The order of tuples has no significance.
  - (But in practice that's not the case, the order of tuples in data base effects its performance)

Relational Keys

Data Integrity

In relational databases, related information are often grouped together to form tables.

For example, in a company that has many branch offices:

The information related to staff is put together.

The information related to office is saved in another table.

There is also a lot of referencing among tables

i.e. given a ID to a staff, you should be able to locate the address of his office

Integrity Consideration 1

We want to minimize data redundancy within a relation

The database should be able to check for any duplicate tuples before tuples are added or modified. (Referring to the table, above, if the user wants to insert another tuple with a duplicating ID it makes no sense)

This is enforced by something called **Primary key**.

Relational Keys: Super Key*

**Super key**: one or more attributes that uniquely identifies a tuple within a relation.

Relational Keys: Candidate Key & Primary Key*

**Candidate key**: a super key such that no proper subset is a super key within the relation

Uniqueness: Every tuple has a unique value for that set of attributes

Minimality: No proper subset of the set has the uniqueness property: minimality

**Primary key:** the candidate key that is selected to identify tuples uniquely within the relation.

Choosing Candidate Keys

U can't necessarily infer the candidate less based solely on the data in your table

More often than not, an instance of a relation will only hold a small subset of all the possible values

E.g. Restaurant booking number queue might reset to 1 after a large number

(Thus this should not be selected as a candidate key.)

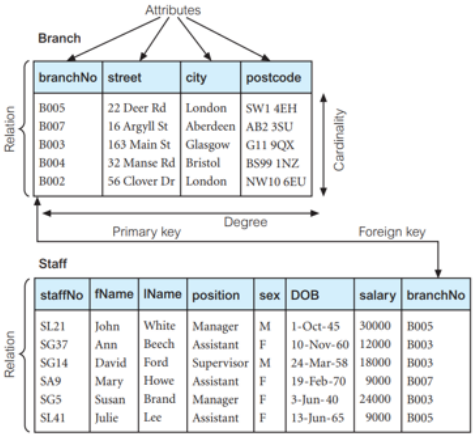
You must combine the real world knowledge to select candidate keys.

Integrity Consideration 2

It is very common for tuples in one relation to reference data from another relation

As a result, a database should provide such mechanism to ensure correct references.

This is enforced by something called **foreign key**.




Relational Keys: Foreign Key*

**Foreign Key**

One or more attributes within one relation that **must match** the candidate key of some (possibly the same) relation

Example: we want the values of the 'branchNo' in relation staff to be one of the 'branch No' in the relation Branch.

Relational Keys an d Integrity Constraints

Primary Key enforces **entity integrity**

Foreign Key enforces **referential integrity**

Why they are important?

Lecture 2 — Relational Algebra

Relational Algebra

DBMS: Software that is designed to enable users to program to store, retrieve and update data.

The language used to describe these operations is called Structured Query Language (SQL)

What is relational algebra?

Theoretical foundation of (part of) SQL.

Query: "Find all universities with > 20000 students"

Relational Algebra:

$$\pi_{\text{uName}}\left(\sigma_{\text{Enrollment}>20000}(\text{University})\right)$$

SQL:

```
SELECT uName FROM University WHERE
University.Enrollment > 20000
```

Relational Algebra and SQL are declarative (Not procedural like C/Java)

No need to specify the steps of data processing

Relational Algebra: Operator Properties

Like functions in Java or C **operators** in relational algebra requires operands and returns results.

One of the operands must be a relation.

The "return result" is a temporary relation (called View) that has been processed.

The returned relation can be then processed by another operator.

This property is called closure: the ability that allows expression to be nested (Textbook 5.1)

Relational Algebra Vs SQL

Relational algebra is set-based, that means duplicate tuples in result are always eliminated, while duplicates are kept in SQL results.

Relational Algebra: Basic operator

Selection: Sigma ($\sigma$)

Usage: $\sigma_{\text{predicate}}(R)$

Predicate: a function with parameters that either returns a true or false.

The selection operation works on a single relation $R$ and defines a relation that contains only those tuples of $R$ that specified condition (predicate).

Query: "List all staffs with a salary greater then £10,000"

$$\sigma_{\text{salary}>10000}(\text{staff})$$

Or $\sigma_{\text{staff.salary}>10000}(\text{staff})$ when the column name salary appears in two tables

Predicate also supports logical operators $\wedge$ (AND), $\vee$ (OR), and $\sim$ (NOT).

$$\sigma_{\text{salary}>10000\wedge\text{salary}<20000}(\text{staff})$$

Projection: PI ($\pi$)

Usage: $\pi_{a_1,a_2,a_3,\ldots,a_n}(R)$

Where $a_1, a_2, a_3, \ldots, a_n$ is a subset of attribute names form the relation $R$

The Projection operation works on a single relation $R$ and defines a relation that contains vertical subset of $R$, extracting the values of specified attributes and eliminating duplicates.

Query: "Produce a list of salaries for all staff, showing only the staffNo, fName, lName, and salary details."

$$\pi_{\text{staffNo,fName,lName,salary}}(R)$$

Union: ∪

Usage" $R_1 \cup R_2$

The union of two relations $R_1$ and $R_2$ defines a relation that contains tall the tuples of $R_1$ or $R_2$ or both $R_1$ and $R_2$ with all duplicated tuples being eliminated. $R_1$ and $R_2$ must be **union-compatible**.

Query: "List all cities where there is either a branch office or a property for rent"

$\pi_{\text{city}}(\text{Branch}) \cup \pi_{\text{city}}(\text{PropertyForRent})$

Union Compatible

Union-compatible means that the number of attributes must be the same and their corresponding data types also match.

Union compatible:

A: (First_name (char), Last_name(char), Date_of_birth(date))
B: (Fname(char), Lname(char), DOB(date))

Both table A and B have 3 attributes with same corresponding data types

Not union compatible:

A: (First_name (char), Last_name(char), Date_of_birth(**date**))
B: (Fname(char), Lname(char), PhoneNumber(**number**))

Or:

A: (First_name (char), Date_of_birth(**date**), Last_name(**char**))
B: (Fname(char), Lname(**char**), DOB(**date**))

Set difference: Minus −

Usage: $R - S$

The Set difference operation defines a relation consisting of tuples that are in relation $R$, but not in $S$. $R$ and $S$ must be union-compatible.

Query: "List all cities where there is a branch office but no properties for rent"

$\pi_{\text{city}}(\text{Branch}) - \pi_{\text{city}}(\text{PropertyForRent})$

Intersection: ∩

Usage: $R \cap S$

The Intersection operation defines a relation consisting of the set of all tuples that are in both $R$ and $S$. $R$ and $S$ must be union-compatible.

Same asL $R - (R - S)$

Query: "List all cities where there is both a branch office and at least one property for rent"

$\pi_{\text{city}}(\text{Branch}) \cap \pi_{\text{city}}(\text{PropertyForRent})$

## Cartesian Product and Joins

Cartesian product: ×

Usage: $R \times S$

The Cartesian product operation defines a relation that is the concatenation of every tuple of relation $R$ with every tuple of relation $S$.

Eg. $(\pi_{\text{clientNo,fName,lName}}(\text{Client})) \times (\pi_{\text{clientNo,propertyNo,comment}}(\text{Viewing}))$

The result of the operation is not very meaningful, but based on that, we can filter out rows and obtain some useful information.

For example,

Query: "list the name and comments of all clients who have viewed a property for rent"

$\sigma_{\text{Client.clientNo=Viewing.clientNo}}((\pi_{\text{clientNo,fName,lName}}(\text{Client})) \times (\pi_{\text{clientNo,propertyNo,comment}}(\text{Viewing})))$

Theta join $\bowtie_F$

Usage: $R \bowtie_F S$

The Theta join defines a relation that contains tuples satisfying the predicate $F$ from $R \times S$. The predicate $F$ is of the form "$R.a_i \theta S.b_i$" where $\theta$ may be one of the comparison operators ( $<$ , $\leq$ , $>$ , $\geq$ , $=$ , $\neq$ ).

If $F$ contains only equality ( $=$ ), it is instead called **Equijoin**.

The previous example query can be simply represented by a Theta Join:

$R \bowtie_F S = \sigma_F(R \times S)$

$(\pi_{\text{clientNo,fName,lName}}(\text{Client})) \bowtie_{\text{Client.clientNo=Viewing.clientNo}} (\pi_{\text{clientNo,propertyNo,comment}}(\text{Viewing}))$

Natural Join $\bowtie$

Usage: $R \bowtie S$

The Natural join is an Equijoin of the two relations $R$ and $S$ over all common attributes $x$. One occurrence of each common attribute is eliminated from the result.

Query: "List the names and comments of all clients who have viewed a property for rent"

$(\pi_{\text{clientNo,fName,lName}}(\text{Client})) \bowtie (\pi_{\text{clientNo,propertyNo,comment}}(\text{Viewing}))$

The natural join works based on attributes names and their types (domain)

If two tables have multiple attributes with the same names and data types, then all these attributes will be selected. For example:

A: (**First_name (char)**, **Last_name(char)**, Date_of_birth(date))

B: (**First_name (char)**, **Last_name(char)**, Phone_Number(date))

Left Outer Join: $⟕$

Usage: $R ⟕ S$

The (left) Outer join is a join in which tuples from $R$ that do not have matching values in the common attributes of S are also included in the result relation.

Missing values in the second relation are set to null.

Query: "Produce a status report on property viewings"

$(\pi_{\text{propertyNo,street,city}}(\text{PropertyForRent})) ⟕ \text{Viewing}$

Right Outer Join (⟖) and Full Outer Join (⟗) also exists.

Rename Operator: Rho ($\rho$)

Usage: $\rho(E)$ or $\rho_{a_1,a_2,\ldots,a_n}(E)$

The Rename operation provides a new name $S$ for the expression $E$, and optionally names the attributes as $a_1, a_2, \ldots, a_n$.

Why rename?

Natural Join relies on the attribute name to work properly.

Sometimes attributes in different tables referring to the same kind of information have different names.

Query: "Change 'Univesity' table to 'uni'"

$\rho_{\text{uni}}(\text{University})$

Query: "changes 'Branch' table to 'b' without chains its tuples"

$\rho_{\text{b(col1,col2,col3,col4)}}(\text{Branch})$

Division, Aggregation, Grouping

Division: ÷

Usage: $R \div S$

Divison is used when we wish to express queries with "all":

Eg. "Which persons have a bank account at ALL the banks in the country?"

Eg. "Which students are registered on ALL the courses"

Assume relation $R$ is defined over the attribute set $A$ and relation $S$ is defined over the attribute set $B$ such that $B \subseteq A$ ($B$ is a subset of $A$)

Query: "Identify all clients who have viewed all properties with three rooms"

$\pi_{\text{clientNo,propertyNo}}(\text{Viewing})$ A: (clientNo, propertyNo)

$\pi_{\text{propertyNo}}(\sigma_{\text{rooms=3}}(\text{PropertyForRent}))$ B: (propertyNo)

$(\pi_{\text{clientNo,propertyNo}}(\text{Viewing})) \div (\pi_{\text{propertyNo}}(\sigma_{\text{rooms=3}}(\text{PropertyForRent})))$

Aggregate: $\mathfrak{I}$

Usage: $\mathfrak{I}_{\text{AL}}(R)$

Applies the aggregate function list, AL, to the relation $R$ to define a relation over the aggregate list. AL contains one or more ( <aggregate_function> , <attibute> ) pairs.

Aggregate functions:

| | |
|---|---|
| COUNT | Returns the number of values in the associated attribute. |
| SUM | Returns the sum of the values in the associated attribute. |
| AVG | Returns the average of the values in the associated attribute. |
| MIN | Returns the smallest value in the associated attribute. |
| MAX | Returns the largest value in the associated attribute. |

Query: "How many property cost more than £350 per month to rent?"

$\rho_{r(\text{myCount})}(\mathfrak{I}_{\text{COUNT propertyNo}}(\sigma_{\text{rent>350}}(\text{PropertyForRent})))$

Query: "Find the minimum, maximum, and average staff salary"

$\rho_{r(\text{myMin,myMax,myAvg})}(\mathfrak{I}_{\text{MIN salary,MAX salary,AVG salary}}(\text{Staff}))$

Grouping:

Usage: ${}_{\text{GA}}\mathfrak{I}_{\text{AL}}(R)$

Groups the tuples of relation $R$ by the grouping attributes, GA.

Then applies the aggregate function list AL to define a new relation. AL contains one or more ( <aggregate_function> , <attibute> ) pairs.

The resulting relation contains the grouping attributes, GA, along with the results of each of the aggregate functions.

Simplified: find aggregation result for each different value in GA.

Query: "Find the number of staff working in each branch and the sum of their salaries"

$\rho_{r(\text{branchNo,myCount,mySum})}({}_{\text{branchNo}}\mathfrak{I}_{\text{COUNT staffNo,SUM salary}}(\text{Staff}))$

Lecture 3 — SQL 1: Defining & Modifying Tables

Database Containment Hierarchy

A Computer may have one or more **clusters**.

A cluster is a database server.

= a computer can run multiple database servers

Each cluster contains one or more **catalogs**.

Catalog is just another name for "database".

Each catalog consists of a set of **schemas**.

Schema is a namespace of tables, and security boundary.

A schema consists of tables, views, domains, assertions, collations, translations, and character sets. All have same owner.

Creating a Database

First, we need to create a schema

```
CREATE SCHEMA name;
CREATE DATABASE name;
```

If you want to create table in this schema, you need to tell MySQL to "enter" into this schema by typing:

```
USE name;
```

After that, if you create dateless, they will be created in this schema

SQL: Table Definition — Syntax of **`CREATE TABLE`**, Data types of SQL

Create Tables

```
CREATE TABLE name (
    col-name datatype [col-options],
    ⋮
    col-name datatype [col-options],
    [constraint-1],
    ⋮
    [constraint-n],
);
```

`[xxx]`: something optional.

Common Data Types (MySQL): `col-name` **`datatype`** `[col-options]`

| Data Type | Keyword |
|---|---|
| **Boolean** | **`BOOLEAN`** |
| **Character** | **`CHAR`**`(size)` **`VARCHAR`**`(size_limit)` **`TEXT`** **`TINYTEXT`** |
| **Number** | **`INTEGER`**`(size)` **`FLOAT`**`(size, d)` **`DOUBLE`**`(size, d)` |
| **Date** | **`DATE`** **`DATETIME`** **`TIME`** |

| | |
|---|---|
| **`CHAR`** | has fixed string length |
| **`VARCHAR`** | has variable string length |
| **`DATE`** | 1-oct-2015 |
| **`Time`** | 21:05:02 |
| **`FLOAT`**`(size,d)` | size := all significant figures, d := digits after decimal point |

Strings In SQL

Strings in SQL are surrounded by single quotes:

```
'I AM A STRING'
```

Single quotes with a string are doubled or escaped using \

```
'I''M A STRING'  'I\'M A STRING'  ''
```

- is an empty string

In MYSQL, double quotes also work (Not a standard)

Column Options: `col-name datatype [col-options]`

**NOT NULL**

Values of this column cannot be null.

**UNIQUE**

Each ball must be unique (candidate key on a single attribute)

**DEFAULT** value

Default value for this column if not specified by the user.

Does not work in MS Access.

**AUTO_INCREMENT** = baseValue

```
CREATE TABLE Persons (
      Id INT AUTO_INCREMENT,
      ...
```

A value (usually max(col)+1) is automatically inserted when the data is added.

You can also manually provide values to override this behavior

```
ALTER TABLE Persons AUTO_INCREMENT = 100;
```

Create Tables: Full Example

```
CREATE TABLE Persons (
      Id          INT UNIQUE NOT NULL,
      LastName    VARCHAR(255) NOT NULL,
      FirstName   VARCHAR(255),
      Age         INT,
      City        VARCHAR(255)
);

CREATE TABLE Branch (
      branchNo    VARCHAR(255) UNIQUE NOT NULL,
      Street      VARCHAR(255) NOT NULL,
      city        VARCHAR(255) NOT NULL,
      postCode    VARCHAR(255)
);
```

SQL: Constrains — Domain, Primary Key, unique key, foreign key

Domain Constraints

You can limit the possible values of an attribute by adding a **domain constraint**.

A domain constraint can be defined along with the column or separately:

```
CREATE TABLE Persons (
      id    INTEGER PRIMARY KEY,
      Name  VARCHAR(100) NOT NULL,
      Sex   CHAR NOT NULL CHECK (Sex IN ('M','F'))
);
```

Unfortunately, MySQL does not support domain constraints, these constraints will be ignored.

Constraints:

General Syntax:

**CONSTRAINT** name **TYPE details;**

If you don't provide a name, one will be generated

MySQL provides the following constraint types:

**PRIMARY KEY**

**UNIQUE**

**FOREIGN KEY**

**INDEX**

**UNIQUE**

Usage: **CONSTRAINT** name **UNIQUE** (col1, col2, ...);

Same effect as the one specified with column options but can be applied to multiple columns and make them one candidate key.

The following candidate keys are different

One candidate key (a, b, c):

Tuples (1, 2, 3) and (1, 2, 2) are allowed

Separate candidate keys (a) (b) (c):

Tuples (1, 2, 3) and (1, 2, 2) are NOT allowed

**PRIMARY KEY**

Usage: **CONSTRAINT** name **PRIMARY KEY** (col1, col2, ...);

**PRIMARY KEY** also automatically adds **UNIQUE** and **NOT NULL** to the relevant column definitions

Example:

```
CREATE TABLE Branch (
      branchNo    CHAR(4),
      Street      VARCHAR(100),
      city        VARCHAR(25),
      postCode    VARCHAR(7),
      CONSTRAINT branchPK PRIMARY KEY (branchNo)
);
```

**FOREIGN KEY**

Usage:

```
CONSTRAINT name
      FOREIGN KEY
            (col1, col2, ...)
      REFERENCES
      table-name
            (col1, col2, ...)
      [ON UPDATE ref_opt
       ON DELETE ref_opt]

      ref_opt: RESTRICT | CASCADE | SET NULL | SET DEFAULT
```

To apply a foreign key you need to provide:

The column which make up the foreign key

The referenced table

The columns which are referenced by the foreign key

Reference options can be optionally provided

Example:

```
CONSTRAINT branchFK
      FOREIGN KEY branchNo
      REFERENCES  Branch(branchNo)
```

Reference Options:

| | |
|---|---|
| **RESTRICT** | stop the user from table update. |
| **CASCADE** | let the changes flow on. |
| **SET NULL** | make referencing values null. |
| **SET DEFAULT** | make referencing values the default for their column, not available in MySQL |

Can be applied to one or both kinds of table updates:

**ON UPDATE**

**ON DELETE**

Deleting Tables

You can delete tables with the **DROP** keyword

```
DROP TABLE [IF EXISTS] table-name1, table-name2,...;
```

All tuples will be deleted as well.

Undoing is sometimes impossible.

Foreign Keys constraints will prevent **DROPS** under the default **RESTRICT** option

To overcome this, either remove the constraints or drop the tables in the correct order (referencing table first).

## SQL: Altering Tables — Keyword **ALTER**

**ALTER** — Change Tables:

**ALTER** is used to add, delete, or modify columns in an existing table.

Add column:

```
ALTER TABLE table-name ADD col-name datatype [col-options];
```

Drop column:

```
ALTER TABLE table-name DROP COLUMN col-name;
```

**ALTER** — Modify Columns:

Modify column name and definition:

```
ALTER TABLE table-name
      CHANGE COLUMN
      col-name new-col-name datatype [col-options];
```

Modify column definition only:

```
ALTER TABLE table-name
      MODIFY COLUMN
      col-name  datatype [col-options];
```

**ALTER** — Add Constraints:

Add a constraint:

```
ALTER TABLE table-name
      ADD CONSTANT name definition;
```

For instance:

```
ALTER TABLE branch
      ADD CONSTANT ck_branch UNIQUE (street);
```

**ALTER** — Remove Constraints:

Remove a constraint:

```
ALTER TABLE table-name
      DROP INDEX | FOREIGN KEY | PRIMARY KEY name;
```

For instance:

```
ALTER TABLE branch
      ADD CONSTANT ck_branch UNIQUE (street);
```

## SQL: Tuple Operations

**INSERT**:

Inserts rows into the database with the specified values:

```
INSERT INTO table-name (col1, col2, ...)
      VALUES (val1, val2, ...),
             ⋮
             (val1, val2, ...);
```

If a value is added to every column, the list can be implicit:

```
INSERT INTO table-name VALUES (val1, val2, ...);
```

But then the ordering of values must match the ordering of attributes in the table

Example:

```
INSERT INTO Employee (ID, Name, Salary)
      VALUES (2, 'Mary', 26000),
             (2, 'Max', 233333);
```

**UPDATE**:

Changes values in specified rows based on **WHERE** conditions

```
UPDATE table-name
    SET col1 = val1 [, col2 = val2 ...],
    [WHERE condition];
```

All rows where the condition is true have the columns set to the given values

If no condition is given all rows are changed.

Values are constraints or can be computed from columns.

Examples:

```
UPDATE Employee
    SET Salary = 15000,
        Name = 'Jane'
    WHERE ID = 4;
UPDATE Employee
    SET Salary = Salary * 1.05;
```

**DELETE**:

Removes all rows, or those which satisfies a condition

```
DELETE FROM table-name [WHERE condition];
```

If no condition is given then ALL rows are deleted.

Examples:

```
DELETE FROM Employee WHERE Salary > 20000;
DELETE FROM Employee;
```

Lecture 4 — SQL 2.1: **SELECT**

**SELECT**: Overview

```
SELECT [DISTINCT | ALL]
      col-list FROM table-names
      [WHERE condition]
      [ORDER BY col-list]
      [GROUP BY col-list]
      [HAVING condition]
```

Examples in This Lecture:

**Student**

| ***ID*** | **First** | **Last** |
|---|---|---|
| **S103** | John | Smith |
| **S104** | Mary | Jones |
| **S105** | Jane | Brown |
| **S106** | Mark | Jones |
| **S107** | John | Brown |

**Grade**

| ***ID*** | **Code** | **Mark** |
|---|---|---|
| **S103** | DBS | 72 |
| **S103** | IAI | 58 |
| **S104** | PR1 | 68 |
| **S104** | IAI | 65 |
| **S106** | PR2 | 43 |
| **S107** | PR1 | 76 |
| **S107** | PR2 | 60 |
| **S107** | IAI | 35 |

**Course**

| ***Code*** | **Title** |
|---|---|
| **DBS** | Database Systems |
| **PR1** | Programming 1 |
| **PR2** | Programming 2 |
| **IAI** | Introduction to AI |

**SELECT**

In the simplest form, **SELECT** is the SQL version of projection:

$\pi_{\text{attributes}}(R)$

```
SELECT col1[,col2...] FROM table-name;
```

Selection $\sigma_{\text{predicate}}(R)$ can be achieved by using a **WHERE** clause:

```
SELECT * FROM table-name
      WHERE predicate;
```

Star (*) := all columns of table

**DISTINCT** and **ALL**

Sometimes duplicate entries exists.

**DISTINCT** removes duplicates

**ALL** retains duplicates, used as default if neither is supplied.

These will work over multiple columns.

Examples:

```
SELECT ALL Last FROM Student;
      -> Last:[Smith, Jones, Brown, Jones, Brown]
SELECT DISTINCT Last FROM Student;
      -> Last:[Smith, Jones, Brown]
```

**WHERE** Clauses

A **WHERE** clause restricts rows that are returned

It takes the form of a condition

Only rows that satisfy the condition are returned.

Example Conditions

```
Mark < 40
First = 'John'
First <> 'John'
First = Last
(First = 'John') AND (Last = 'Smith')
(MARK < 40) OR (MARK > 70)
```

<> := NOT equal to

Examples:

```
SELECT * FROM Grade
      WHERE Mark >= 60;
      -> Grade:[  ID,  Code, Mark]
               {(S103,  DBS,   72),
                (S104,  PR1,   68),
                (S104,  IAI,   65),
                (S107,  PR1,   76),
                (S107,  PR2,   60)}
SELECT DISTINCT ID FROM Grade WHERE Mark >= 60;
      -> Grade:[  ID]
               {(S103),
                (S104),
                (S107)}
```

Queries:

"Find a list of the ID numbers and Marks for student who have passed (50+) in IAI"

```
SELECT ID, Mark FROM Grade
      WHERE (Code = 'IAI') AND (Mark >= 50);
```

"Find the combined list of the student IDs for both the IAI and PR2 module"

```
SELECT DISTINCT ID FROM Grade
      WHERE (Code = 'IAI') OR (CODE = 'PR2');
```

**SELECT** and Cartesian product

Cartesian product of two tables can be obtained by using:

```
SELECT * FROM Table1, Table2;
```

If same table column names exists ambiguity will occur, thus this can be resolved by referencing columns with the table name: `TableName.ColumnName`

Example:

```
SELECT First, Last, Mark
      FROM Student, Grade
      WHERE (Student.ID = Grade.ID) AND (Mark >= 40);
```

**SELECT** from Multiple Tables

**WHERE** clause is a key feature when selecting from multiple tables.

Unrelated combinations can be filtered out.

```
SELECT * FROM
       Student, Grade, Course
WHERE Student.ID = Grade.ID AND
       Course.Code = Grade.Code;
```

If same table column names exists ambiguity will occur, thus this can be resolved by referencing columns with the table name: `TableName.ColumnName`

| **Student** | | | |
|---|---|---|---|
| ***sID*** | nName | sAddress | sYear |

| **Module** | | |
|---|---|---|
| ***mCode*** | mCredits | mTitle |

| **Enrollment** | |
|---|---|
| ***sID*** | **mCode** |

Query: “Produce a list of all student names and all their enrollments (module codes)”

```
SELECT DISTINCT Student.nName FROM Student, Enrollment
WHERE Student.sID = Enrollment.sID;
```

Query: “Find a list of module titles being taken by the student named Harrison”

```
SELECT DISTINCT Module.mTitle
FROM Student, Module, Enrollment
WHERE (Student.sID = Enrollment.sID) AND
       (Enrollment.mCode = Module.mCode) AND
       Student.sName = 'Harrison';
```

Query: “Find a list of module codes and title for all modules currently being taken by first year students”

```
SELECT DISTINCT Module.mCode, Module.mTitle
FROM Student, Module, Enrollment
WHERE (Student.sID = Enrollment.sID) AND
       (Enrollment.mCode = Module.mCode) AND
       Student.sYear = 1;
```

Aliases

Aliases rename columns or tables

Make shorter, easier to use, more meaningful names while resolving ambiguity.

Two forms exists:

Column Alias

```
SELECT column [AS] new-col-name
```

Table Alias

```
SELECT * FROM table [AS] new-table-name
```

Alias Example:

| **Employee** | | **WorksIn** | |
|---|---|---|---|
| ***ID*** | Name | ***ID*** | Department |

```
SELECT E.ID AS empID, E.name, W.Department -- (3)
    FROM Employee E, WorksIn W             -- (1)
    WHERE E.ID = W.ID;                     -- (2)
```

Note: column aliases cannot be used in a **WHERE** clause.

Aliases and ‘Self-Joins’

Aliases can be used to copy a table, so that it can be combined with itself:

Query: “Find the names of all employees who work in the same department as Andy”

```
SELECT A.Name
    FROM Employee A, Employee B
    WHERE A.Dept = B.Dept AND B.Name = 'Andy';
```

Subqueries

A **SELECT** statement can be nested inside another query to form a subquery.

The results of the subquery are passed back to the containing query.

Query: “retrieve a list of name of people who are in Andy’s department”

```
SELECT Name FROM Employee WHERE Dept =
    (SELECT Dept FROM Employee
        WHERE Name = 'Andy');
```

Often a subquery will return a set of values rather than a single value.

We cannot directly compare a single value to a set doing so will result in an error.

Options for handling sets:

| | |
|---|---|
| **IN** | checks to see if value is in a set |
| **EXISTS** | checks to see if a set is empty |
| **ALL/ANY** | checks to see if a relationship holds for every/one member of a set |
| **NOT** | used with any of the above |

Handling Sets: **IN**

Using **IN** we can see if a given value is in a set of values.

```
SELECT columns FROM tables
    WHERE col IN set;
```

**NOT IN** checks to see if a given value is not in the set.

```
SELECT columns FROM tables
    WHERE col NOT IN set;
```

The set can be given explicitly or can be produced in a subquery.

Query: “List all Employee from Marketing or Sales department”

```
SELECT * FROM Employee
    WHERE Department IN ('Marketing', 'Sales');
```

Query: “List all employee who’s not a manager”

```
SELECT * FROM Employee
    WHERE Name NOT IN
        (SELECT Manager FROM Employee);
```

Handling Sets: **EXISTS**

Using **EXISTS** we can see whether there is at least one element in a given set

```
SELECT columns FROM tables
    WHERE EXISTS set;
```

**NOT EXISTS** is true if the set is empty.

```
SELECT columns FROM tables
    WHERE NOT EXISTS set;
```

The set is always given by a subquery.

Query: “Retrieve all the info for those employees who are also managers”

```
SELECT * FROM Employee AS E1
    WHERE EXISTS
        (SELECT * FROM Employee AS E2
            WHERE E1.Name = E2.Manager);
```

Handling Sets: **ANY** and **ALL**

**ANY** and **ALL** compare a single value to a set of values.
They are used with comparison operators like: `=, >, <, <>, >=, <=`
`val = ANY (set)` is true if there is at least one member of the set equal to the value.
Query: “Find the name(s) of the employee(s) who earn more than someone else”

```
SELECT Name FROM Employee
    WHERE Salary > ANY(SELECT Salary FROM Employee);
```

`val = ALL (set)` is true if all members of the set equal to the value.
Query: “Find the name(s) of the employee(s) who earn the highest salary”

```
SELECT Name FROM Employee
    WHERE Salary >= ALL(SELECT Salary FROM Employee);
```

Word Search using **LIKE**

Word Search is commonly used for searching product catalogues etc, searched using keywords/partial keywords.
We can use the **LIKE** keyword to perform string comparisons in queries.
**LIKE** is bot the same as ‘=’ because it allows wildcard characters/
It is NOT normally case sensitive.
Query: “search for ‘crypt’ in book names from a database of books”

```
SELECT * FROM books
    WHERE bookBame LIKE "%crypt%";
```

The ‘`%`’ character can represent any number of characters, including none.
The ‘_’ character represents exactly one character.
Search for a set of words:

**AND** can describe condition with all words present
**OR** can describe condition with any words present

Query: “search for ‘crypt’ or ‘cloud’ in book names from a database of books”

```
SELECT * FROM books
    WHERE bookBame LIKE "%crypt%" OR bookBame LIKE "%cloud%";
```

Date, Time, Datetime

3 different types for time column

| Type | Description | Example |
|---|---|---|
| **DATE** | A Day, Month and Year | '1981-12-16'<br>'81-12-16' |
| **TIME** | Hours, Minutes and Seconds | '15:24:39' |
| **DATETIME** | Combination of above | '1981-12-16 15:24:39' |

Timestamp
as **DATETIME**, usually used to display current date and time ('2020-11-16 15:24:39')
Usual conditions may be used on **WHERE** clauses, such as:

```
SELECT * FROM table-name WHERE date-of-event < '2020-01-01';
SELECT * FROM table-name WHERE date-of-event LIKE '2020-01-%';
```

## **JOIN**s

**JOIN**s can be used to combine tables in a **SELECT** query

**CROSS JOIN** Returns all pairs of row from `A` and `B`, the same as Cartesian product

**INNER JOIN** Returns pairs of rows satisfying a condition

**NATURAL JOIN** Returns pairs of rows with common values in identically named columns

**OUTER JOIN** Returns pair of rows satisfying a condition, BUT ALSO handles NULLS

## **CROSS JOIN**

Syntax:

```
SELECT * FROM A CROSS JOIN B;
```

(same as: `SELECT * FROM A, B;`)

Usually needs **WHERE** to filter out unrelated tuples.

## **INNER JOIN**

Syntax:

```
SELECT * FROM A INNER JOIN B ON condition;
```

Can also use a **USING** clause that will output rows with equal values in the specified columns:

```
SELECT * FROM A INNER JOIN B
    ON condition;
```

Usually needs **WHERE** to filter out unrelated tuples.

A single primary key row will be out put representing the equal values form both `A` and `B`, while combining corresponding fields.

## **NATURAL JOIN**

Syntax:

```
SELECT * FROM A NATURAL JOIN B;
```

A **NATURAL JOIN** is effectively a special case if ab **INNER JOIN** `B` where the **USING** clause has specified all identically named columns.

Same as the $A \bowtie B$ in relational algebra.

## **JOIN**s Vs. **WHERE** Clauses

**INNER/NATURAL JOIN** are not absolutely necessary.

You can obtain the same results by selecting from multiple tables and using appropriate **WHERE** clauses.

Should/Should not use then?

YES:

Lead to concise and elegant queries

**NATURAL JOIN**s are extremely common

NO:

Support for **JOIN**s can vary between DBMSs.

## **OUTER JOIN**

**OUTER JOIN**s are practical by replacing unknown value with **NULL**:

Syntax:

```
SELECT cols FROM table1 type OUTER JOIN table2 ON condition;
```

Where **type** is one of **LEFT/RIGHT/FULL**.

Query: “List all student’s enrollment status whether or not it exists”

```
SELECT * FROM
    Student LEFT OUTER JOIN Enrollment
    ON Student.ID= Enrollment.ID;
```

Query: “List all enrollment status of students regardless if the student exists”

```
SELECT * FROM
    Student RIGHT OUTER JOIN Enrollment
    ON Student.ID= Enrollment.ID;
```

Only **LEFT/RIGHT OUTER JOIN**s are supported in MySQL, a **FULL** join will become:

```
(SELECT * FROM
      Student LEFT OUTER JOIN Enrollment
      ON Student.ID= Enrollment.ID);
UNION
(SELECT * FROM
      Student RIGHT OUTER JOIN Enrollment
      ON Student.ID= Enrollment.ID);
```

Example:

| **Student** | | | | |
|---|---|---|---|---|
| ***ID*** | Name | aID | pID | Grad |
| **Phone** | | | | |
| ***pID*** | pNumber | pMobile | | |
| **Degree** | | | | |
| ***ID*** | Classification | | | |
| **Address** | | | | |
| ***aID*** | aStreet | aTown | aPostcode | |

Query: ”For student graduating in absentia, find a list of all student IDs, names, addresses, phone numbers and their final degree classifications.”

```
SELECT ID, Name, aStreet, aTown, aPostcode, pNumber, Classification
      FROM Student
            LEFT OUTER JOIN Phone ON Student.pID = Phone.pID
            LEFT OUTER JOIN Address ON Student.aID = Address.aID
            INNER JOIN Degree ON Student.ID = Degree.ID
      WHERE Grad = 'A';
```

**ORDER BY**

The **ORDER BY** clause sorts the results of a query

- You can sort in ascending(default) or defending order.
- Multiple columns can be given.
- Cannot order by a column which isn’t in the result.

```
SELECT columns FROM tables
            WHERE condition
            ORDER BY col1 [ASC|DESC][, col2 [ASC|DESC]...];
```

Arithmetic

As well as columns, a **SELECT** statement can also be used to

- Compute arithmetic expressions
- Evaluate functions

Aliases are often helpful to use when dealing with expressions or functions.

```
SELECT Mark / 100 FROM Grades;
SELECT Salary + Bonus FROM Employee;
SELECT 1.20 * Price AS 'Price inc. VAT' FROM Products;
```

Aggregate Functions

Aggregate functions compute summaries of data in a table.

Most aggregate functions (except **COUNT(*)**) work on a single column of numerical data

Aggregate functions:

| | |
|---|---|
| **COUNT** | The number of rows |
| **SUM** | The sum of the entries in the column |
| **AVG** | The average entry in a column |
| **MIN/MAX** | The minimum/maximum entries in a column |

Best to use with aliases.

**COUNT** Syntax

```
SELECT COUNT(*) AS Count FROM Grades;
SELECT COUNT(DISTINCT Code) AS Count FROM Grades;
```

**SUM, MIN/MAX, AVG** Syntax

```
SELECT SUM(Mark) AS Total FROM Grades;
SELECT MAX(Mark) AS Best FROM Grades;
SELECT MIN(Mark) AS Worst FROM Grades;
SELECT AVG(Mark) AS Mean FROM Grades;
```

Combining Aggregate Functions

Aggregate functions can be combined using arithmetic:

```
SELECT MAX(Mark) - MIN(Mark) AS Range FROM Grades;
```

Example:

| **Modules** | | |
|---|---|---|
| ***Code*** | Title | Credits |
| **Grades** | | |
| ***Name*** | Code | Mark |

```
SELECT
SUM(Mark * Credits) / SUM(Credits) AS Average_Weighted_Marks
FROM Grades, Modules
WHERE Grades.Code = Modules.Code AND Name = 'John';
```

**GROUP BY**

Sometimes we want to apply aggregate functions to groups of rows

Example: find the average mark of each student individually

The **GROUP BY** clause achieves this.

```
SELECT column_set1 FROM tables
    WHERE predicate
    GROUP BY column_set2;
```

Where every entry in `column_set2` should be in `column_set1`, be a constant, or be an aggregate function.

Query: “find the average mark of each student individually”

```
SELECT Name,AVG(Mark) AS Average
        FROM Grades
        GROUP BY Name;
```

**HAVING**

**HAVING** is like a **WHERE** clause, except that it only applies to the results of a **GROUP BY** query

It can be used to select groups which satisfies a given condition

```
SELECT Name,AVG(Mark) AS Average
        FROM Grades
        GROUP BY Name
        HAVING AVG(Mark) >= 40;
```

**SET** operations

**UNION**, **INTERSECT** and **EXCEPT**

These treats tables as sets and are the usual set operators of union, intersection and difference

They all combine the results from two **SELECT** statements

The results of the two **SELECT**s should have the same columns and corresponding data types.

**UNION** Example

"Find, in a single query, the average mark for each student and the average mark overall."

```
SELECT Name,AVG(Mark) AS Average
      FROM Grades
      GROUP BY Name
UNION
SELECT 'Total' AS Name, AVG(Mark) AS Average
      FROM Grades;
```

Missing Information

Sometimes we don't know what value an entry in a relation should have

We know that there is a value, but don't know what it is

There is no value at all that makes any sense

Two main methods have been proposed to deal with this

**NULL**s can be used as markers to show that information is missing

A default value can be used to represent the missing value

**NULL**

Represents a state for an attribute that is currently unknown or is not applicable for this tuple.

**NULL**s are a way to deal with incomplete or exceptional data

**NULL** is a place holder for missing or unknown value of an attribute. It is not itself a value.

E.g. A new staff is added but has not yet decided which branch he belongs to

Problems with **NULL**s

Problems extending relational algebra operation to **NULL**s:

Selection operation: if we check tuples for "Mark > 40" and for some tuple Mark is **NULL**, do we include it?

Comparing tuples in two relation: are two tuples **AND** the same or not?

Additional problems for SQL:

**NULL**s treated as duplicates?

Inclusion of **NULL**s in **COUNT**, **SUM** and **AVG**?

If yes, how?

Arithmetic operations behavior with argument **NULL**?

Theoretical Solutions

Use three-valued logic instead of classical two valued logic to evaluate conditions.

When there are no **NULL**s around, conditions evaluates to true or false, but if a null is involved, a condition might evaluate to the third value ('undefined', or 'unknown').

3-valued logic table:

If the conditions involves a Boolean combination, we evaluate it as follows:

| a | b | a OR b | a AND b | a == b |
|---|---|---|---|---|
| **TRUE** | **TRUE** | TRUE | TRUE | TRUE |
| **TRUE** | **FALSE** | TRUE | FALSE | FALSE |
| **TRUE** | **Unknown** | TRUE | Unknown | Unknown |
| **FALSE** | **TRUE** | TRUE | FALSE | FALSE |
| **FALSE** | **FALSE** | FALSE | FALSE | TRUE |
| **FALSE** | **Unknown** | Unknown | FALSE | Unknown |
| **Unknown** | **TRUE** | TRUE | Unknown | Unknown |
| **Unknown** | **FALSE** | Unknown | FALSE | Unknown |
| **Unknown** | **Unknown** | Unknown | Unknown | Unknown |

SQL `NULL`s in Conditions

```
SELECT * FROM Employee WHERE Salary > 15,000;
```

`WHERE` clause of SQL `SELECT` uses three-valued logic:

Only tuples where the condition evaluates true are returned.

`Employee`s where its `Salary > 15,000` evaluates to unknown is excluded.

SQL `NULL`s in Aggregation

```
SELECT
    AVG(Salary) AS Average,
    COUNT(Salary) AS Count,
    SUM(Salary) AS Sum
FROM Employee;
```

In aggregation, the `NULL` values will be ignored, only when using `COUNT(*)` then the `NULL`s will be counted.

SQL `NULL`s in `GROUP BY`

```
SELECT COUNT(Name) AS Count
FROM Employee
GROUP BY Salary;
```

`NULL`s are treated as equivalents in `GROUP BY` clauses.

SQL `NULL`s in `ORDER BY`

```
SELECT *
FROM Employee
ORDER BY Salary;
```

`NULL`s are considered and reported in `ORDER BY` clauses, `NULL`s are onsidered to be lower than any values.

Default Values

Default values are an alternative way to the use of `NULL`s

A value that makes non sense is chosen in normal circumstances.

```
age INT DEFAULT -1,
```

These will be treated as actual values.

Default values can have more meaning than `NULL`s:

None, Unknown, Not supplied, Not applicable

Not all default represent missing information. It can depend on the situation.

Default Values Example

Default values are

“Unknown” for Name

-1 for Weight and Quantity

```
CREATE TABLE Parts (
      ID INT NOT NULL,
      Name VARCHAR(255) DEFAULT "Unknown" NOT NULL,
      Weight INT DEFAULT -1,
      Quantity INT DEFAULT -1
)
```

-1 is used for Weight and Quantity as it is not sensible otherwise.

There are still problems, such as updating all quantity by incrementing 5

```
UPDATE Parts SET Quantity = Quantity + 5
```

Lecture 7 — Entity-Relationship Diagrams

Database Design

This lecture introduces the technique to design a database from a piece of written requirements

Need to consider

What is the databased going to be used for?

What tables, attributes, keys are needed?

Designing your database is important

Often results in a more efficient and simpler queries once the database has been created

May help reduce Data redundancy in the table.

Entity-Relationship Modeling

E/R Modeling is used for conceptual design

Entities: Objects or items of interest.

Attributes: Properties of an entity.

Relationships: Links between entities.

For example, in a University database we might have entities for Students, Modules and Lecturers.

Student might have attributes such as their ID, Name, and Course

Student could have relationship with Modules (enrollment) and Lecturers (tutor/tutee)

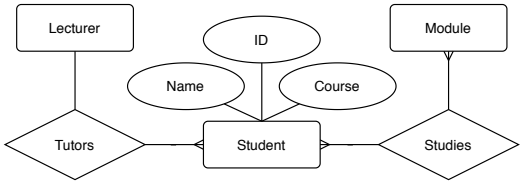
Entity-Relationship Diagrams

E/R Models are often represented as E/R diagrams that

Give a conceptual view of the data base

Are independent of the choice of DBMS

Can identify some problems in a design




Component 1: Entities

Entities represent objects or things of interest.

Physical things like students, lecturers, employees, products

More abstract things like modules, orders, courses, projects

Entity:

Is a general type or class, such as Lecturer or Module

Has instances of that particular type. E.g. DBI and IAI are isntances of Module

Has attributes (such as name, email address)

In E/R Diagrams, we represent entities as boxes with rounded corners.

Component 2: Attributes

Attributes are facts, aspects, properties, or details about an entity

Students have IDs, names, courses, addresses, …

Modules have codes, titles, credit weighs, levels, …

Attributes have:

A name, an associated entity, domains of possible values

For each instance of the associated entity a value from the attributes domain

In an E/R diagram attributes are drawn as ovals, each attribute is linked to its entity by a line, the name of the attribute is written in the oval.

Component 3: Relationships

A relationship is an association between two or more entities

Each Student takes several Modules.

Each Module is taught by a Lecturer.

Each Employee works for a single Department.

Relationships have

A name, a set of entities that participate in them, a degree: number of entities that participate (most have degree 2), and a cardinality ratio.

Cardinality Ratios

One to one (1:1)

Each lecturer has a unique office & office are single occupancy

One to many (1:M)

A lecturer may tutor many students, but each student has just one tutor

Many to many (M:M)

Each students take several modules, and each modules is taken by several students.

E/R Diagram relationships

Relationships are shown as links between two entities

The name is given in a diamond box

The ends of the link show cardinality.

Making E/R Models

To make an E/R model you need to identify:

Entities

Attributes

Relationships

Cardinality Ratios

We obtain theses from a problem description

General guidelines

Since entities are things or objects they are often nouns in the description.

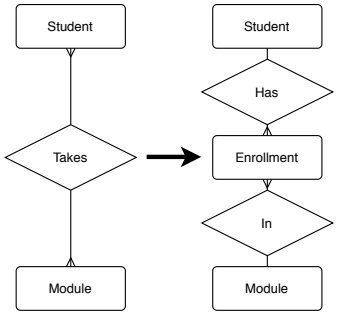
Attributes are facts or properties, and so are often nouns.

Verbs often describe relationships between entities.

Removing M:M Relationships

We can split many to many relation ship into two one to many relationships

An additional entity is created to represent the M:M relationship

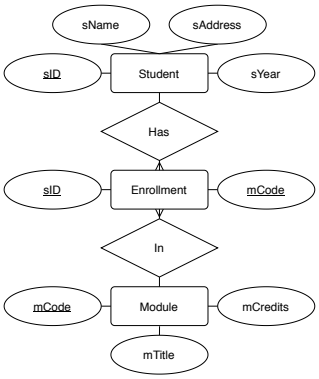



The Enrollment table:

Will have columns for the student ID and module code attributes

Will have a foreign key to Student for the ‘has’ relationship

Will have a foreign key to Module for the ‘in’ relationship



Entities and Attributes

- Sometimes it is hard to tell if something should be an entity or an attribute
  - They both represent objects or facts about the world
  - They are both often represented by nouns in description
- General Guidelines
  - Entities can have attributes but attributes have no smaller parts
  - Entities can have relationships between them, but an attribute belongs to a single entity.

One to one relationship

- We can merge the two entities that take part in a redundant relationship together
  - They become a single entity, the new entity had all the attributes of the old ones

Relationships as Foreign Keys

- 1:1 are usually not used, or can be treated as a special case of M:1
- M:1 are represented as a foreign key from the M-side to the 1
- M:M are split into two M:1 relationship

Lecture 7 — Normalization

What is Normalization?

Normalization is another way of creating accurate representation of the data, relationships between the data, and constraints on the data that is pertinent to the enterprise.
It is a design technique for producing a set of suitable relations that support the data requirements of an enterprise.

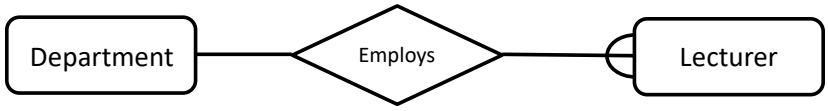
Suitable Relations

Characteristics of a suitable set of relations include:

The minimal number of attributes necessary to support the data requirements of the enterprise
Attributes with a close logical relationship are found in the same relation
Minimal redundancy with each attribute represented only once with the important exception of attributes that form all or part of foreign keys.

Minimal Redundancy

Given the example from E/R diagram, department and lecturer can be designed as:




Department

| dName | lecName |
|---|---|
| CS | Matt |
| CS | Andrew |
| EEE | Thomas |
| EEE | John |

Lecturer

| lecName |
|---|
| Matt |
| Andrew |
| Thomas |
| John |

Department

| dName |
|---|
| CS |
| EEE |

Lecturer

| lecName | dName |
|---|---|
| Matt | CS |
| Andrew | CS |
| Thomas | EEE |
| John | EEE |

The design on right is better since dName can be seen as key instead of duplicating.
However, this relationship is in reality 1 to 1, thus its better to just combine the two tables.

Functional Dependency - A method for finding relationships between attributes within a table

Functional Dependency

To find relationship between attributes within a table can be done by regrouping attributes based on their context and splitting the tables.

Functional Dependency (FD):

If $\mathrm{A}$ and $\mathrm{B}$ are attribute set of relation $R$, $\mathrm{B}$ is functionally dependent on $\mathrm{A}$ (**denoted** $\mathrm{A} \rightarrow \mathrm{B}$), if each value of $\mathrm{A}$ in $R$ is associated with exactly one value of $\mathrm{B}$ in $R$. $\mathrm{A}$ is called **determinant**.

Ex. $\mathrm{LecID} \rightarrow \mathrm{LName}$ (Reads Lecture ID determines Lecture Name)

However we have a bug problem here, the FD below is also true:

$$\mathrm{LecID}, \mathrm{LName}, \mathrm{LEmail} \rightarrow \mathrm{LName}, \mathrm{LEmail}$$

But, $(\mathrm{LecID}, \mathrm{LName}, \mathrm{LEmail})$ is not a good primary key.

Full Functional dependency

Full functional dependency indicates that if $\mathrm{A}$ and $\mathrm{B}$ are two sets of attributes of a relation, $\mathrm{B}$ is fully functionally dependent on $\mathrm{A}$, if $\mathrm{B}$ is functionally dependent on $\mathrm{A}$, but not on any proper subset of $\mathrm{A}$.

In other words, determinants should have the minimal number of attributes necessary to maintain the functional dependency with the attribute(s) on the right hand-side.

Partial Functional Dependency

Partial FDs:

A FD, $A \rightarrow B$ , is a partial FD if some attribute $A$ can be removed and the FD still holds.

Formally, there is some proper subset of $A$, $C \subset A$ such that $C \rightarrow B$.

Determinant in Full/Partial FDs

Determinants in Full FDs can become candidate keys if we split the table.
Determinants in partial FDs can only become super keys.

FDs in Normalization

We only care about full FDs in normalization.

Partial FDs results in super keys, if you use a foreign key to refer to a super key in another table, you will need extra columns in the referencing table, rendering it unecessary.

More about Determinants

The determinants has a M:1 or 1:1 relationship with other attributes in FDs.
Two staffs may have the same name:

Thus, pone staff name can be associated with more than one staff member. (M:1)

Two staffs may have the same position:

Thus, one position may be associated with more than one staff number. (M:1)

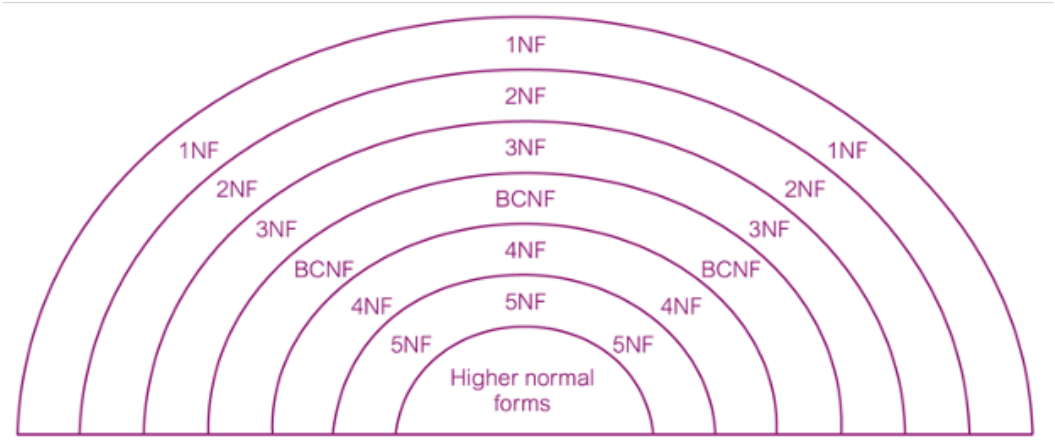
In ER Modeling, M:1 relationship between tables will become foreign keys.
We can infer that, for attributes that have a M:1 relationships, if they belong to the same context they will be grouped into one relation, otherwise they will be split into two tables and lines with a foreign key

Transitive Dependency

Transitive dependency describes a condition where $A$, $B$, and $C$ are attributes of a relation such that if $A \rightarrow B$ and $B \rightarrow C$, then $C$ is transitively dependent on $A$ via $B$ ( provided that $A$ is not functional dependent on $B$ or $C$ ).

Normal Forms — How can we use FDs and TDs to redesign tables.

The process of Normalization



Normalization: Some Notes

In relational database, we always try to assign tables with primary keys (or at least candidate keys.

Why? Because relational databases are about relations.

Attributes within one context forums an entity (table).

Primary keys make referencing possible.

Attributes connecting entities become foreign keys.

Foreign keys form these connections.

Can I insist that no unique keys are used?

Then its beyond the topic of this module :-)

Remember, we are learning one of the possible design decisions of managing data, don't limit your imagination

First Normal Form

In most definitions of the relational model

All values should be in first normal form (1NF):

If all data values are atomic

Unnormalized Form (UNF)

Unnormalized relation

Problems With UNF

To update a particular value in all attribute as a list of values, then you need to update all lists of values in attribute.

To delete a particular value from the list of values in an attribute will also require you to manually traverse through all values.

A new value cannot be added without information of the primary key.

Method 1:

1. Remove the repeating group by entering appropriate data into the empty columns fo trows containing the repeating data ( also called 'flattening' the table).
2. Assign a new primary/unique ket to the new table.

Method 2:

1. Identify primary/unique key
2. Place the repeating data along with a copy of the original (unique) key attributes into a separate relation.

Problems in 1NF

Adding new values to fields without FK is impossible.

If a set of non FK is modified, then the change must be made to all FK sets.

If a set of FK is deleted then the associated non FK sets will be permanently lost.

Second Normal Form

Definition of Second normal form 2NF:

A relation is in second normal form (2NF) if it is in 1NF and

No non-key attribute is partially dependent on the primary key

In other words, no $\mathrm{C} \rightarrow \mathrm{B}$ where $\mathrm{C}$ is a strict subset of a primary key and $\mathrm{B}$ is a non-key attribute.

Example: Let $(\mathrm{A}, \mathrm{C})$ be the primary key and $\mathrm{B}$ is a non-key attribute

Hence $\mathrm{A}, \mathrm{C} \rightarrow \mathrm{B}$, and if $\mathrm{C} \rightarrow \mathrm{B}$ or $\mathrm{A} \rightarrow \mathrm{B}$ then a relation is not in 2NF.

1NF to 2NF

Identify the primary key(s) for the 1NF relation

Identify the functional dependencies in the relation

If partial dependencies exist on the primary key, remove them by placing attributes of the corresponding full FD in a new relation and leave its determinant in the original table

For example:

If: relation $(A, B, C)$ where $A, B \rightarrow C$ and $B \rightarrow C$

Then: make new relation $(A, B)$ and $(B, C)$
For example:
If for 1NF table $(\text{Module}, \text{Dept}, \text{Lecturer}, \text{Text})$
$\{\text{Module}, \text{Text}\} \rightarrow \{\text{Lecturer}, \text{Dept}\}$ and $\{\text{Module}\} \rightarrow \{\text{Lecturer}, \text{Dept}\}$ then
$\rightarrow$ 2NFa $(\text{Module}, \text{Dept}, \text{Lecturer})$ and 2NFb $(\text{Module}, \text{Text})$

Problem in 2NF

We cannot add a lecturer who is not assigned with any module. Because the module is the primary key.
By deleting a module, we looses the information about the Lecturer forever.
To change the department for a Lecturer, we need to manually change multiple rows.

Third Normal Form (3NF)

Based on the concept of transitive dependency
A relation that is 1NF and 2NF and in which no non-key attribute is transitively dependent on the primary key.

2NF to 3NF

Identify the primary key in the 2NF relation.
Identify functional dependencies in the relation
If transitive dependencies exist on the primary key, remove them by placing them in a new relation along with a copy of their determinant.
Example

If taking 2NF relation $(\text{Module}, \text{Dept}, \text{Lecturer})$
Where $\{\text{Module}\} \rightarrow \{\text{Lecturer}\}$ and $\{\text{Lecturer}\} \rightarrow \{\text{Dept}\}$so theres a transitive FD from the primary ket $\{\text{Module}\}$to $\{\text{Dept}\}$
Then make relations:

$(\text{Module}, \text{Lecturer})$ and $(\text{Lecturer}, \text{Dept})$