

Database Architectures and Models

A *Database* is “a collection of data arranged for ease and speed of search and retrieval”.

A *Database System* consists of not only the data, but also the software, hardware and users associated with it. Such a system allows users to store, update, retrieve, organise and protect their data.

A *Database Management System (DBMS)* is the software that controls the information. It provides users with a *Data Definition Language (DDL)*, a *Data Manipulation Language (DML)*, and a *Data Control Language (DCL)*. Often these are actually all the same language. A DBMS provides persistence (storage, backup), concurrency (multiple users), integrity, security and data independence. The *Data Dictionary* holds metadata about the data in the database, e.g., descriptions of database objects, access information, schemas and mappings, the dictionary itself.

File Based Systems

- Data is stored in files
- Each file has a specific format
- Programs that use these files depend on knowledge about that format
- Disadvantages:
 - o No standards
 - o Data duplication
 - o Data dependence
 - o No query generation
 - o No provision for security, recovery, concurrency, etc.

Relational Systems

- Data stored in *tuples* or *records* in a *relation* or *table*
- Based on a sound mathematical theory

The ANSI/SPARC Architecture [American National Standards Institute / Standards Planning and Requirements Committee]

- Proposed a framework for databases (1975)
- Three levels:
 - o Internal level – for system designers
 - o Conceptual level – for database designers and administrators
 - o External level – for users
- Internal level deals with physical storage of data, the structure of records on disk. Used by database system programmers

I.e., an internal schema:

```
RECORD EMP
LENGTH 44
HEADER: BYTE (5)
        OFFSET = 5
SALARY: FULLWORD
        OFFSET = 30
DEPT:  BYTE (10)
        OFFSET: 34
```

- Conceptual level deals with the overall organisation of the data using abstraction to remove unnecessary internal detail (i.e., the SQL coding)
- External level provides a view of the database tailored to a user, i.e.,
Payroll:

```
String name
double Salary
```
- *Mappings* translate information from one level to the next. These mappings provide data independence so changes to the internal level shouldn't affect the conceptual level (physical independence) and conceptual level changes shouldn't affect the external levels (logical independence).

Revision Notes

The Relational Model

Data stored in *relations* (tables); Each relation has a *scheme* (heading); The scheme defines the relation's *attributes* (columns); Data takes the form of *tuples* (rows).

More formally, a scheme is a set of attributes; a tuple assigns a value to each attribute in its scheme (i.e. a set of attribute-value pairs); a relation is a set of tuples with the same scheme.

Since relations are *sets* of tuples, the tuples of a relation are *unique* and *unordered*. The number of tuples in a relation is called the *cardinality* of the relation. Similarly, since schemes are sets of attributes, the attributes of a relation are also *unique* and *unordered*. The number of attributes in a relation is the relation's *degree*. The *domain* lists the possible values for each attribute.

Keys

- A set of attributes in a relation is called a *Candidate Key* IFF:
 - o Every tuple in this set has a unique value (uniqueness)
 - o No proper subset of this set has the uniqueness property (minimality)
- One candidate key is usually chosen to identify tuples. This is called the *Primary Key*. Primary Key values must not contain nulls
- *Foreign Keys* are used to link data between relations. A set of attributes in the *referencing* relation is a Foreign Key if its value matches a candidate key in the *referenced* relation or is null – i.e., if it maps each tuple in the first relation to *at most* one tuple in the second relation.

Referencing Integrity

- When relations are updated, referential integrity can be violated. Usually occurs when a referenced tuple is updated or deleted.
- Options:
 - o Restrict – stops any action that violates integrity
 - o Cascade – allows changes to flow through, e.g.:

Department

DID	DName
13 16	Marketing
14	Accounts
15	Personnel

Employee

EID	EName	DID
15	John Smith	13 16
16	Mary Brown	14
17	Mark Jones	13 16
18	Jane Smith	NULL

- o Nullify – makes problem values null.

Relational Algebra

- Just like 'normal' algebra, relational algebra has a set of operations which take relations as input and return relations as results. Some operators have restrictions on them.
- Since operators take and return relations, they can be chained together, again, just like in 'normal' algebra, i.e. $(3 + 2) * 7$. This is called *Closure*.
- Operators include:
 - o *Select* - returns tuples satisfying a given condition
 - o *Project* - returns the attributes specified
 - o *Union, Intersection, Difference* - as in set theory
- Example: To find the module codes taken by John Smith:

- 1) Select those tuples where First = 'John'
- 2) Select those tuples where Last = 'Smith'
- 3) Take the intersection of (1) and (2)
- 4) Project (3) over *Module*.

Enrolment

First	Last	Module
John	Smith	G52DBS
John	Smith	G5BADS
Mary	Smith	G5AHOC
John	Brown	G53RDB

Revision Notes

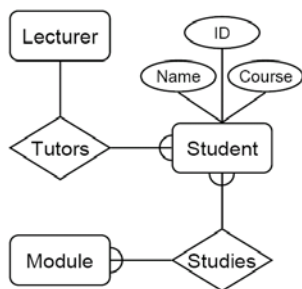
Naming Conventions

- A consistent naming convention helps to clarify the structure of the database.
- Assign each table a unique prefix, so in table, *Student*, the name attribute is called stuName.
- Give foreign keys the same name as their reference.

Database Design

- Conceptual design: a model independent of DBMS
- Logical design: create the database in a DBMS
- Physical design: how is the database stored

Conceptual Design often uses Entity-Relationship modelling. An ER diagram shows the entities, attributes and relationships of a database.



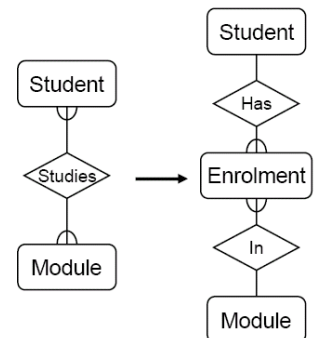
- Entities are drawn as boxes with rounded corners
- Attributes are drawn as ovals and joined to their associated entity.
- Relationships are represented as diamond boxes and join the entities they relate. The joining lines show the cardinality of the relationship:



More on Relationships

Cardinality Ratios:

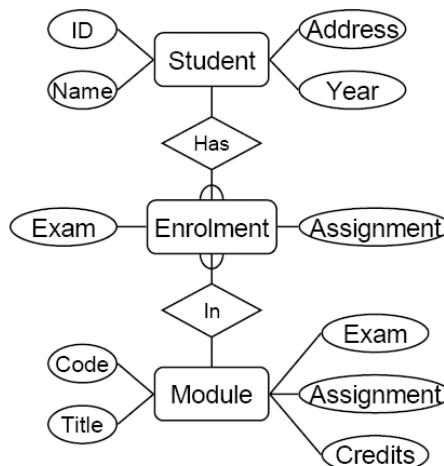
- One to one (1:1), e.g. each lecturer has a unique office.
 - One to many (1:M), e.g. a lecturer may tutor many students, but each student has just one tutor.
 - Many to many (M:M), e.g. each student takes several modules, and each module is taken by several students.
- Many to many relationships are difficult to represent in a database, and so is split into two one to many relationships (see left).



Some relationships between entities, A and B, might be redundant if:

- It is a 1:1 relationship between A and B
- Every A is related to a B and every B is related to an A
- Redundant relationships can be merged together

Another example E/R diagram:





SQL (Structured Query Language)

- Strings surrounded by 'single quotes'. An empty string is "".
- Datatypes include INT, REAL, CHAR (length), VARCHAR (max length), DATE

Create Table

```
CREATE TABLE <name> (
    <column definitions>
    <...>
    <constraints>
    <...>
)
```

The Column definitions are defined as

```
<col-name> <type> [NULL|NOT NULL] [DEFAULT <val>] [constraints]
```

Constraints are defined as

```
CONSTRAINT <name> <type> <details>
```

where type could be PRIMARY KEY or FOREIGN KEY etc., and details are the columns which form the constraint. A Primary Key includes unique and not null constraints. Foreign keys add REFERENCES <table> <attribute>.

Example:

```
CREATE TABLE Enrolment (
    stuID INT NOT NULL,
    modCode VARCHAR(6) NOT NULL,

    CONSTRAINT enrPK PRIMARY KEY (stuID, modCode),
    CONSTRAINT enrStu FOREIGN KEY (stuID) REFERENCES Student (stuID),
    CONSTRAINT enrMod FOREIGN KEY (modCode) REFERENCES Module (modCode)
)
```

Auto-incrementing Fields:

- MS Access and mySQL differ slightly in their implementation:
- mySQL: CREATE TABLE Auto (
 - ID INT AUTO_INCREMENT, ← modifies the integer type
 - ...
- MS Access: CREATE TABLE Auto (
 - ID AUTOINCREMENT, ← its own datatype
 - ...
- When inserting data into a table with an auto-incrementing field, do not specify a value - let the DBMS calculate it.

Deleting Tables

DROP TABLE [IF EXISTS] <name> will delete a table and any information in it.

Altering Tables

ALTER TABLE can:

- Add a new column - ALTER TABLE <name> ADD COLUMN <column definition>
- Remove an existing column - ALTER TABLE <name> DROP COLUMN <col name>
- Add a new constraint - ALTER TABLE <name> ADD CONSTRAINT <details>
- Remove an existing constraint - ALTER TABLE <name> DROP CONSTRAINT <def>

Revision Notes

The Data Dictionary

The Data Dictionary holds information about tables, columns, access, etc. It is information about information, i.e. *metadata*. It is accessed in SQL using `SHOW` commands:

- `SHOW TABLES`
- `SHOW COLUMNS FROM <table name>` or `DESCRIBE <table name>`
This shows what columns are in the table, their type and details such as keys, defaults, null settings etc.
- Many parts of a relational DBMS use the data dictionary, for example:
 - o The query processor uses it to validate queries
 - o It stores information about locks for concurrency
 - o It stores information about users for security
 - o It stores information about how the database is physically stored on disk for retrieval

Insert

```
INSERT INTO <table>
(col1, col2, ...)
VALUES
(val1, val2, ...)
```

Number of columns must be equal to number of values. If you are adding values into *every* column, it is not necessary to list the columns.

Update

```
UPDATE <table> SET
col1 = val1,
col2 = val2,
...
[WHERE <condition>]
```

If no condition is given, *all rows will be changed!*

Delete

```
DELETE FROM <table>
[WHERE <condition>]
```

Again, if no condition is given, all rows will be deleted.

Select

Queries a set of tables and returns results as a table.

```
SELECT <columns> FROM <table>
```

`<columns>` can be a single column, a comma-separated list of columns or a `*` for all columns.

Tip: Before doing a `DELETE` with condition, test it out using `SELECT`!

```
SELECT [DISTINCT|ALL] <columns> FROM <table>
```

using 'distinct' shows only unique rows. 'all' is the default value.

```
SELECT * FROM <table>
WHERE <condition>
```

multiple conditions possible, e.g. `(First = 'John') AND (Last = 'Smith')`

```
SELECT * FROM Texts
WHERE Title LIKE '%vis%'
```

`%` is a wildcard character (Access uses `*`) that matches any string

Select can be used on multiple tables (`SELECT * from table1, table2`). If no conditions are specified, *each row of the second table is given for every row in the first*. To reference attributes of particular tables, use the `Table.column` syntax, e.g.:

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

Revision Notes

Joins

- Combine tables
- A CROSS JOIN B: returns all pairs of rows from A and B

```
SELECT * tableA CROSS JOIN tableB
```

 is the same as

```
SELECT * from tableA, tableB
```
- A NATURAL JOIN B: returns pairs of rows with common values (in columns with the same name)

```
SELECT * tableA NATURAL JOIN tableB
```

 is the same as

```
SELECT * from tableA, tableB WHERE (tableA.col1 = tableB.col1)
AND (tableA.col2 = tableB.col2)
...
```
- A INNER JOIN B: returns pairs of rows satisfying a given condition

```
SELECT * tableA INNER JOIN tableB ON <condition>
```

 is the same as

```
SELECT * from tableA, tableB WHERE <condition>
```
- Joins vs. Where clauses:
Joins are good in that they lead to concise queries. Natural joins are very common. However, support for JOINS varies among SQL dialects.

Aliases

- In the context of a query, rename columns or tables by using the keyword AS.
- Example, using two Tables, one called Employee, the other WorksIn:

```
SELECT E.ID AS empID,
       E.Name
       W.Dept
FROM Employee AS E
       WorksIn AS W
WHERE E.ID = W.ID
```
- This query returns the table (empID, Name, Dept)
- Aliases can also be used to make an effective copy of a table so it can be combined with itself:

```
SELECT A.Name FROM Employee AS A,
       Employee AS B
WHERE A.Dept=B.Dept
AND B.Name='Andy'
```
- This query returns the names of employees working in the same department as Andy.

Subqueries¹

- 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.
- Example:

```
SELECT Name FROM Employee
WHERE Dept = (SELECT Dept FROM Employee
              WHERE Name='Andy')
```
- This returns the names of employees in the same department as Andy.

Boolean Operators

- (NOT) IN – checks to see if a value is in the set or not

```
SELECT * FROM Employee
WHERE Name NOT IN (SELECT Manager FROM Employee)
```
- This returns the names of employees who are not managers
- (NOT) EXISTS – checks to see if the set is empty or not

```
SELECT * FROM Employee AS E1
WHERE EXISTS
      (SELECT * FROM Employee AS E2
       WHERE E1.Name = E2.Manager)
```
- This returns the names of all the managers.

¹ Subqueries *are not supported* by the version of MySQL currently running on CSiT machines.

Revision Notes

- ALL/ANY – checks to see if a relationship holds for all/any member of the set

```
SELECT Name FROM Employee
WHERE Salary >= ALL (SELECT Salary
                     FROM Employee)
```
- Returns the name of the highest earning employee.

```
SELECT Name FROM Employee
WHERE Salary > ANY (SELECT Salary
                   FROM Employee)
```
- Returns the names all but the lowest earning employee.

Word Indexes

- Used for searching by keywords
- A word index keeps:
 - o A table of items to be searched
 - o A table of keywords
 - o A linking table saying which keywords belong to which items.
- Example:
 Book catalogue containing:
 - o “Cryptonomicon” by Neil Stephenson
 - o “Applied Cryptography” by Bruce Schneier

<u>Items</u>		<u>Keywords</u>		<u>ItemKey</u>	
itmID	itmTitle	keyID	keyWord	itmID	keyID
1	Cryptonomicon	1	Cryptonomicon	1	1
2	Applied Cryptography	2	Applied	2	2
		3	Cryptopgrahy	2	3

- The SQL query is therefore of the following form:

```
SELECT * FROM Items
WHERE itmID IN
           (SELECT itmID FROM ItemKey
            WHERE keyID IN
              (SELECT keyID FROM Keywords
               WHERE keyWord LIKE `crypt%`))
```
- This would return the items that contain words starting crypt-
- Word index searches can be combined with AND or OR clauses.

Ordering

SELECT <columns> FROM <tables> WHERE <conditions> ORDER BY <col> [ASC | DESC]
 Can sort by multiple columns, i.e. SELECT * FROM Grades ORDER BY Code ASC, Mark DESC

Constants & Arithmetic

E.g.,

```
SELECT Mark/100 FROM Grades
SELECT Salary * Bonus FROM Employee
SELECT 1.175 * Price FROM Products
```

Aggregate Functions (Count, Sum, Avg, Max, Min) and Group By

- Except Count, these work on a single column of data
- Use as Alias to name the result
 E.g.,

```
SELECT COUNT(*) AS Count FROM myTable
SELECT SUM(Mark) AS Total FROM Marks
```



Revision Notes

To apply aggregate functions to *groups* of rows, use Group By. E.g., to find the average mark for each Student, `SELECT AVG(Mark) AS Average FROM Marks GROUP BY Name.`

You can group by multiple columns, i.e.,

```
SELECT TOTAL(Sales) AS Total FROM Store GROUP BY Department, Month
```

Having

This is like a WHERE clause, but it refers to the results of a GROUP BY query, e.g.,

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

N.B., WHERE refers to rows of a table, and so cannot use aggregate functions, and HAVING refers to groups of rows and so cannot use columns which are not in the GROUP BY. Think of a query being processed in the order: Combine tables → Where → Group By & Aggregates → Column selection → Having → Order By.

Union, Intersect & Except

Select queries can be joined with these operators in the same way as sets. E.g., find the average mark of each student, and the overall average in a single query:

```
SELECT Name, AVG(Mark) AS Average FROM Table GROUP BY Name
UNION
SELECT 'Overall' AS Name, AVG(Mark) FROM Table
```

A Big SQL Example – Examiners’ Report

A single query to:

- List of students and their average mark.
- For first and second years, the average is for that year.
- For finalists, the average is 40% the second year and 60% the third year
- Results sorted by year then average mark (desc) then surname, first name, ID
- Take into account the number of credits a module is worth

Tables:

Student – ID, First, Last, Year

Grade – ID, Code, Mark, YearTaken

Module – Code, Title, Credits

The basic query:

```
SELECT Last, First, Student.ID, Year, <some maths> AS Average
FROM Student, Grade, Module
WHERE Student.ID = Grade.ID AND Module.Code = Grade.Code
```

The Maths:

For the second and third years:

```
SUM(Mark * Credits) / 120
```

For the third years, we can exploit the property that 40% is 2*0.2 and 60% is 3*0.2:

```
SUM( (0.2 * YearTaken) * (Mark * Credits) ) / 120
```

Conditions:

For the First & Second Years:

```
WHERE YearTaken = Year AND Year IN (1,2)
```

For the Finalists:

```
WHERE Year = 3 and YearTaken IN (2, 3)
```

Ordering and Grouping:

```
GROUP BY Year, Student.ID, First, Last
```

```
ORDER BY Year DESC, AverageMark DESC, Last, First, Student.ID
```



Revision Notes

The final query:

```
((SELECT Last, First, Student.ID, Year, SUM(Mark * Credits) / 120 AS Average
FROM Student, Grade, Module
WHERE Student.ID = Grade.ID AND Grade.Code = Module.Code
AND YearTaken = Year AND Year IN (1,2))
UNION
(SELECT Last, First, Student.ID, Year,
SUM( (0.2 * YearTaken) * (Mark * Credits) ) / 120 AS Average
FROM Student, Grade, Module
WHERE Student.ID = Grade.ID AND Grade.Code = Module.Code
AND Year = 3 AND YearTaken IN (2,3)))
GROUP BY Year, Student.ID, First, Last
ORDER BY Year DESC, AverageMark DESC, Last, First, Student.ID;
```

Extending SQL

SQL is used to create, control, and query a database. It provides a set of commands to do so, but does not have loops, if-then conditionals, variables, or other common language features.

Solutions include extending SQL, allowing programming languages to work with SQL, embedded SQL and dynamic SQL.

Extending SQL - SQL plus programming features

- PL/SQL (Oracle)

- o Has a block structure with BEGIN (and others) and END acting like { and } in JAVA or C/C++
- o Provides variables, loop constructs, etc.
- o Basic structure:

```
[ DECLARE
-- Variables ]
BEGIN
-- Commands
[ EXCEPTION
-- Error handling ]
END;
```

o Example - A Parity Table:

```
CREATE TABLE Parity (
    num INT,
    parity CHAR(4)
);
BEGIN
FOR x IN 1..10 LOOP
IF MOD(x,2)=0 THEN
INSERT INTO Parity
VALUES(x, 'EVEN');
ELSE
INSERT INTO Parity
VALUES(x, 'ODD');
END IF;
END LOOP;
END;
```

Revision Notes

- Often we want to loop through the results of a SELECT statement. A *cursor* is used to do this. You declare them in the first block by giving the SELECT statement and can then loop over the results, i.e.

```
DECLARE
CURSOR <cursor>
IS <select_stmt>
BEGIN
-- Other statements
FOR <row> IN <cursor> LOOP
-- Do stuff with <row>
END LOOP;
-- Other statements
END;
```

- Worked Example:

```
DECLARE
CURSOR MCur IS
SELECT * FROM Marks WHERE name='smx';
Passes INT := 0;
Soft INT := 0;
Hard INT := 0;
AvgMark REAL := 0.0;
Prog VARCHAR(8) = '';

BEGIN
FOR Row IN MCur LOOP
IF (Row.Mark < 30) THEN
Hard := Hard + Row.Credits;
ELSIF (Row.Mark < 40) THEN
Soft := Soft + Row.Credits;
ELSE
Passes := Passes + Row.Credits;
END IF;
AvgMark := AvgMark + Row.Mark*Row.Credits/120.0;
END LOOP;

IF (Passes = 120)
Prog = 'Progress';
ELSEIF ((Passes >= 100) AND (AvgMark >= 50))
Prog = 'Progress';
ELSEIF ((Passes >= 80) AND (AvgMark >= 40)
AND (Soft = 0))
Prog = 'Progress';
ELSE
Prog = 'Resit';

END;
```

Embedded SQL

- SQL statements are embedded directly in a host language. Some special keywords (often EXEC SQL) are used to identify SQL statements. Variables in the host language are prefixed with colons and are declared to SQL first. Statements which return multiple rows use cursors to step through the results.