Advances in Data Management
Principles of Database Systems, A.Poulovassilis

1 Review of Database Concepts

What is a database ?

A collection of **data** which pertains to a specific aspect of the **real world** and which can be **shared** by users.

Features of a database :

- "users" include application programs, end-users and database administrators
- exists independently of particular users
- persists on secondary storage
- is organised for efficient retrieval and update of data
- includes **meta data** i.e. information about the format and nature of its data

What is a database management system (DBMS) ?

This is a set of programs for **creating**, **accessing** and **maintaining** a database.

A DBMS typically provides tools for:

- **creating** the database
- **interacting** with the database
- **controlling access** to the database
- controlling the **integrity** of data entered into the database
- **recovering** from system failures
- supporting correct **concurrent** access to the database by multiple users

A **database system** consists of a database and DBMS.

Why are DBMSs needed ?

The first DBMSs emerged in the 1960s (IDS, IMS).

Before DBMSs, persistent data was stored in a number of files rather than in a single centralised repository.

This has a number of undesirable consequences:

1. Different applications design their own files according to their own requirements. This encourages:
   - **redundancy** of stored data, leading to increased data storage costs and data management costs;
   - **inconsistency** of data between applications;
   - **ad hoc file design**, since no there is no overall control of the design process.
2. Unnecessary **dependencies** between programs and data, for example a change in the **format** of a file record means that all programs using that file must be amended and recompiled.
3. **Global optimisation** of data access routines is difficult because data is spread over a number files and there is no integrated source information about them.

In order to overcome these problems, the **3-schema architecture** for databases was defined by the ANSI/SPARC Study Group on Database Management Systems, 1975, 1978.

This architecture consists of:

1. A **physical** database and an associated **physical schema** that describes the actual organisation of the data in the database.
2. A **conceptual** database and an associated **conceptual schema** that describes the database contents in terms of some **data model**.
3. A number **external** schemas or **views**, each describing that part of the conceptual database which is of interest to a particular user or group of users.

A data model imposes a set of **inherent constraints** on the data e.g.

- in the relational model, data is structured into tables of atomic values
- in OO models, data is structured into classes of objects
- in XML, data is structured into trees

The 3-schema architecture provides **physical data independence** between the physical and conceptual schemas i.e. changes to the organisation of the data do not affect application programs using only constructs at the conceptual or external level.

It also provides **logical data independence** between the conceptual and external schemas i.e. changes to the conceptual level do not affect application programs using views provided these views are still available.

A **data definition language** (DDL) is used for defining the three levels of schema.

A **data manipulation language** (DML) is used for inserting, retrieving and updating the actual data. The DML provides a set of operations appropriate for the particular data model.

In modern DBMSs, DDL and DML functionality is typically provided within *one* language e.g. SQL.

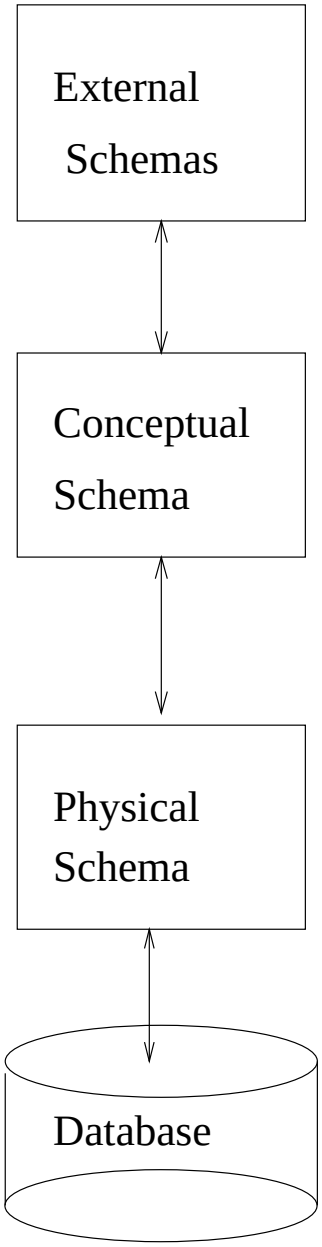

Figure 1: ANSI/SPARC 3-schema architecture

Example — A University Database

Conceptual schema:

- Students(sid: VARCHAR(10), name: VARCHAR(30), age:NUMERIC(2))
- Courses (cid: CHAR(6), cname: VARCHAR(30), units:NUMERIC(1))
- Enrolled(sid: VARCHAR(10), cid: CHAR(6), examMark:NUMERIC(3))

Here, the underlining indicates that the attribute forms part of the **key** of a relation — see below.

Physical schema:

- hash index on sid values of Students
- B+ tree index on cid values of Courses
- bitmap index on units values of Courses

External schemas:

- CourseNums(cid:CHAR(6), enrollment:NUMERIC(3))
- StudentAvg(sid:VARCHAR(10), avg:NUMERIC(5,2))

2 Overview of DBMS Functionality

People who use Database Systems

People use database systems according to a number of roles. Sometimes one particular person may have more than one role.

(i) Database Administrator (DBA). Ultimately responsible for the database resource: authorising access to it, monitoring its integrity, tuning its performance.

(ii) Database Designer. Design of the physical and conceptual schemas, in consultation with system analysts, developers and end-users.

(iii) System Analyst. Design of systems accessing and updating the database, in consultation with end-users. Design of external schemas.

(iv) Developer. Implementation, testing and maintenance of systems.

(v) End-User. Querying, updating, and generating reports from the database.

Main Objectives of DBMSs

- Controlling the Redundancy of data
- Data Structuring and Efficient Access

- Simultaneous access to the data by many users
- Providing appropriate interfaces for the different users
- Ensuring the Security of the data
- Ensuring the Integrity of the data
- Providing facilities for Backup & Recovery of data

We briefly discuss each of these below.

(i) Controlling the Redundancy of Data

If all data is under centralised control, users are automatically aware of what data is already present and are discouraged from **duplication of effort** in design and data acquisition.

Generally, each logical data item will appear in the database **only once** hence storage and update costs minimised.

This also eliminates potential **inconsistency** between multiple copies of the same logical data item.

However, duplication of data is sometimes necessary or advantageous. The point is that the DBMS allows the DBA to **control** this.

(ii) Data Structuring and Efficient Access

The data stored in the database will typically consist of **entities**, their **attributes**, and **relationships** between them. The DBMS should be able to

- represent complex inter-relationships between entities
- retrieve and update related entities efficiently

Since the data is under centralised control, the DBA can design the database so as to achieve good performance of common retrieval and update requests.

(iii) Simultaneous access to the data by many users

If all data is to be central it needs to be accessible by multiple users **simultaneously**. The DBMS must therefore provide some form of **concurrency control** which schedules retrieval and update requests to the same data item so as

- not to cause **physical corruption** of the database
- give each database user the illusion that they are the **only** user

(iv) Providing appropriate interfaces for the different users

For designers and developers: database languages incorporating DDL/DML functionality e.g. SQL, XQuery.

For end-users: forms-based interfaces, natural language interfaces, visual interfaces.

For DBAs: utilities for database loading, reorganisation, logging, recovery, backup, ...

(v) Ensuring the Security of the Data

A DBMS should provide **security and authorisation** facilities to

- ensure that the database is used only via the proper interfaces
- assign **passwords** to users who are allowed to use the database
- assign **authorisation** codes to users showing what parts of the database they are allowed to access and what operations they are allowed to perform on this data

(vi) Ensuring the Integrity of the Data

As mentioned earlier, the particular data model adopted by the DBMS imposes some **inherent constraints** on the database.

In addition, the DBMS provides an **integrity sub-system** which

- allows specification of any additional integrity constraints which must be satisfied by the data
- automatically enforces these constraints during database updates

Categories of such constraints include:

- **referential** integrity constraints, which require that multiple occurrences of the same logical data item are consistent with each other; e.g. in SQL: FOREIGN KEY ... REFERENCES ...
- **value** constraints, which require that data items take a particular value or range of values, possibly depending on the current values of other data items; e.g. in SQL: NOT NULL attributes; CHECKs on attribute values; stand-alone ASSERTIONs
- **structural** integrity constraints requiring that relationships exist or have certain cardinalities; e.g. in SQL: UNIQUE attributes.

(vii) Providing facilities for Backup & Recovery of Data

The constraints discussed in (vi) above were **semantic integrity** constraints on the database.

The **physical integrity** of the database must also be ensured in the event of **systems failure** e.g. if there is a power outage whilst the database is in use, or if an applications program updating the database aborts.

Hence the need for **logging** of database operations and for **database recovery** utilities.

Also in the event of **media failure** such as disk failure, fire, theft, loss ... Hence the need for **database backup** utilities.

3 Relational DBMS

Relational databases are the dominant market database technology, with products such as DB2, Oracle, SQL Server.

The Relational Data Model

Based on the mathematical concept of a **relation**, which is a subset of the cartesian product of a number of sets $D_1, D_2, \ldots, D_n$.

The members of a relation are **n-tuples** of the form $(d_1, d_2, \ldots, d_n)$, where $d_i \in D_i$ for $1 \leq i \leq n$.

In the relational model, relations can be viewed as **tables**:

- Each **row** of the table is a tuple.
- The columns have names, called **attributes**.
- Each **column** consists of the values of that attribute for each tuple.
- These values are drawn from a specified set, termed a **domain**.
- The set of attributes and their domains is called the **scheme** of the relation.

The scheme of a relation R with attributes $A_1, A_2, ..., A_n$ having domains $D_1, D_2, \ldots, D_n$, respectively, is typically written as:

$$R(A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$$

For example:

- Students(<u>sid</u>: VARCHAR(10), name: VARCHAR(30), age:NUMERIC(2))
- Courses (<u>cid</u>: CHAR(6), cname: VARCHAR(30), units:NUMERIC(1))
- Enrolled(<u>sid</u>: VARCHAR(10), <u>cid</u>: CHAR(6), examMark:NUMERIC(3))

An **instance** of $R(A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$ is a set of n-tuples. For example, the following 3 tables are instances of Students, Courses and Enrolled:

Students		
sid	name	age
5000	Mary Brown	25
6000	Joe Smith	30
7000	Jane White	30
8000	John Black	20
9000	Mary Brown	45

Courses		
cid	cname	units
BSCC++	C++ Programming	1
BSCDB1	Databases I	1
BSCDB2	Databases II	2
BSCOPS	Operating Systems	2
BSCHCI	Human Computer Interaction	3
BSCINT	Intelligent Systems	3

Enrolled		
sid	cid	examMark
5000	BSCDB1	35
5000	BSCOPS	55
5000	BSCHCI	60
6000	BSCDB2	38
6000	BSCINT	70
7000	BSCC++	35
7000	BSCOPS	45
7000	BSCDB1	40
8000	BSCDB1	75
8000	BSCDB2	80
8000	BSCINT	68
8000	BSCHCI	50
9000	BSCOPS	55

Relations have sets of one or more attributes which serve as **keys** that uniquely identify rows.

A set of attributes K is a **key** for a relation R if:

(a) in every instance of R, no two tuples agree on all the attributes of K

(b) K is minimal i.e. no subset of it has property (a)

For example, for the Students relation, the set of attributes {sid} has properties (a) and (b), but the set of attributes {sid, name} has property (a) but not property (b) (because property (a) holds for its subset {sid}).

The Relational Algebra

The Relational Algebra was one of two original DML's for the relational data model (the other was the Relational Calculus).

The relational algebra comprises 5 basic operations:

1. **union**, $R \cup S$, where
   (a) R and S must have the same no. of columns and the same domains
   (b) any duplicates in the output are eliminated (because the relational algebra is set-based)

2. **difference**, $R - S$, where (a) again applies (why is a check for duplicates unnecessary ?)

3. **product**, $R \times S$.

   Consists of all possible concatenations of a tuple from R and a tuple from S (i.e. forms the cartesian product of R and S and "flattens" each pair of tuples into a single tuple of atomic values)

4. **projection**, $\pi_{B_1,\ldots,B_m} R$, where R is a relation with scheme $R(A_1, \ldots, A_n)$ and $\{B_1, \ldots, B_m\} \subseteq \{A_1, \ldots, A_n\}$.

   Deletes columns not in $\{B_1, \ldots, B_m\}$ and eliminates any duplicates.

5. **selection**, $\sigma_C(R)$, where R is a relation with scheme $R(A_1, \ldots, A_n)$ and C is a **condition** containing:

   (a) variables, denoting components of tuples; these may be of the form *$i*, denoting a column number, or attribute names;

   (b) constants, denoting values of tuple components;

   (c) arithmetic comparison operators $=, \neq, >, <, \leq, \geq$ between tuple components and constants;

   (d) boolean connectives AND, OR, NOT.

   Returns the set of tuples of R which satisfy condition C.

The Relational Algebra is **closed** i.e. every operation returns a relation. So operations can be arbitrarily **nested** into expressions.

Other useful operations:

- **intersection**, $R \cap S$. (Equivalent to $R - (R - S)$.)
- **theta-join**, $R \bowtie_C S$. Returns the subset of $R \times S$ that satisfies the condition C on the attributes of R and S. (Can be expressed using product and selection.)
- **natural join**, $R \bowtie S$. Returns the pairs of tuples from R and S that agree on the attributes that R and S have in common, eliminating one copy of these attributes. (Can be expressed using product, selection and projection.)
- **equi-join**. Special case of the theta-join in which C consists only of equalities.
- **rename**, $\rho(N(i \to A, j \to B, \ldots), e)$

  Names the relation returned by the relational algebra expression e to N, and renames its i^{th} column to A, its j^{th} column to B etc.

Some Example Relational Algebra Queries over the University Database

1. Find the ID of each course: $\pi_{cid}\ Courses$
2. Find the ID and name of each course: $\pi_{cid,cname}\ Courses$
3. Find the ID and name of each 2-unit course: $\pi_{cid,cname}\ (\sigma_{unit=2} Courses)$

4. Find the ID and name of each course worth more than 1 unit: $\pi_{cid,cname}\ (\sigma_{unit>1}Courses)$
5. Find the ID of each student enrolled on course BSCDB1: $\pi_{sid}\ (\sigma_{cid='BSCDB1'}Enrolled)$
6. Find the name of each student enrolled on course BSCDB1:
   $\pi_{name}\ ((\sigma_{cid='BSCDB1'}Enrolled) \bowtie Students)$
7. Find the id of each student enrolled on a course worth more than 2 units:
   $\pi_{sid}\ ((\sigma_{unit>2}Courses) \bowtie Enrolled)$
8. Find the name of each student enrolled on a course worth more than 2 units:
   $\pi_{name}\ ((\sigma_{unit>2}Courses) \bowtie Enrolled \bowtie Students)$
9. Find the name of each student and the name of each course where the student gained a mark of less than 40:
   $\pi_{name,cname}\ ((\sigma_{examMark<40}Enrolled) \bowtie Courses \bowtie Students)$
10. Find the name of each student and the name of each course where the student gained a mark of between 38 and 42, or between 68 and 72:
    $\pi_{name,cname}\ ((\sigma_{(examMark\geq 38\ AND\ examMark\leq 42)\ OR\ (examMark\geq 68\ AND\ examMark\leq 72)}Enrolled) \bowtie Courses \bowtie Students)$

    or, equivalently:

    $\pi_{name,cname}\ ((\sigma_{examMark\geq 38\ AND\ examMark\leq 42}Enrolled) \bowtie Courses \bowtie Students)\ \cup$
    $\pi_{name,cname}\ ((\sigma_{examMark\geq 68\ AND\ examMark\leq 72}Enrolled) \bowtie Courses \bowtie Students)$
11. Find the ID of all students who have the same name as some other student:
    $\pi_{S1.sid}\ (\rho(S1(1 \to S1.sid, 2 \to S1.name), Students) \bowtie_{S1.name=S2.name\ AND\ S1.sid\neq S2.sid}$
    $\rho(S2(1 \to S2.sid, 2 \to S2.name), Students))$

Exercises Given the relations Students, Courses and Enrolled above, write relational algebra queries which:

1. Find the name of each student who is enrolled on either course BSCDB1 or course BSCDB2.
2. Find the id of each student who is enrolled on both course BSCDB1 and course BSCDB2.

SQL

The relational algebra is useful for expressing alternative query plans during the query optimisation process.

However, it is not really suitable as a query language for end-users and developers, where queries should be *declarative* i.e. stating **what** needs to be computed rather than **how** it should be computed.

It also does not support grouping and aggregation operations, which are very useful for summarising data.

This led to the development of SQL (Structured Query Language), sometimes pronounced “sequel”, as the standard relational query language.

The syntax of the basic SQL query is

$$\begin{array}{rl} \text{SELECT} & [\text{DISTINCT}]\ A_1, A_2, \ldots, A_n \\ \text{FROM} & R_1, R_2, \ldots, R_m \\ [\text{WHERE} & C] \end{array}$$

- $R_1, R_2, ..., R_m$ are relation names
- $A_1, A_2, ..., A_n$ are attributes of these relations
- C is a **condition** composed of:
  - attribute names
  - constants (numbers, strings)
  - comparisons $=, \neq, >, <, \leq, \geq$ of attribute names and constants
  - boolean connectives AND, OR, NOT

The meaning of the above query is:

- consider all possible combinations of tuples $t_1, t_2, \ldots, t_m$ where $t_1 \in R_1$, $t_2 \in R_2$, ..., $t_m \in R_m$ (there are $|R_1| \times |R_2| \times ... \times |R_m|$ possible combinations)
- retain only those combinations that satisfy the condition C, if there is one (i.e. a selection)
- for each such combination, retain only the values of the attributes $A_1, \ldots, A_n$ (i.e. a projection)
- if DISTINCT is specified, eliminate duplicate values

Of course, this will often be a very inefficient evaluation strategy — it is the job of the DBMS's query optimiser to find more efficient ones.

DISTINCT is an optional keyword that indicates that duplicates should be eliminated from the result of the query — the default is that duplicates are not eliminated.

So SQL generally manipulates **bags** rather than **sets** (bags are unordered collections that may contain duplicates).

Some Example SQL Queries over the University Database

1. Find the ID of each course:

```
SELECT cid
FROM   Courses
```

2. Find all the details of each course:

```
SELECT cid, cname, units
FROM   Courses
```

or:

```
SELECT *
FROM   Courses
```

3. Find the ID and name of each 2-unit course:

```
SELECT cid, cname
FROM   Courses
WHERE  units=2
```

4. Find the ID and name of each course worth more than 1 unit:

```
SELECT cid, cname
FROM   Courses
WHERE  units > 1
```

5. Find the ID of each student enrolled on course BSCDB1:

```
SELECT sid
FROM   Enrolled
WHERE  cid='BSCDB1'
```

6. Find the name of each student enrolled on course BSCDB1:

```
SELECT name
FROM   Enrolled, Students
WHERE  cid='BSCDB1' AND Enrolled.sid = Students.sid
```

7. Find the id of each student enrolled on a course worth more than 2 units:

```
SELECT DISTINCT sid
FROM   Enrolled, Courses
WHERE  units > 2 AND Enrolled.cid = Courses.cid
```

8. Find the name of each student enrolled on a course worth more than 2 units:

```
SELECT DISTINCT name
FROM   Students, Enrolled, Courses
WHERE  units > 2 AND Enrolled.cid = Courses.cid AND
       Enrolled.sid = Students.sid
```

or (equivalently):

```
SELECT name
FROM   Students
WHERE  sid IN
       (SELECT sid
        FROM   Enrolled, Courses
        WHERE  units > 2 AND
               Enrolled.cid = Courses.cid)
```

9. Find the name of each student not enrolled on any course worth more than 2 units:

```
SELECT name
FROM   Students
WHERE  sid NOT IN
      (SELECT sid
       FROM Enrolled, Courses
       WHERE  units > 2 AND
              Enrolled.cid = Courses.cid)
```

or (equivalently):

```
SELECT name
FROM   Students
WHERE  NOT EXISTS
      (SELECT *
       FROM Enrolled, Courses
       WHERE  units > 2 AND
              Enrolled.cid = Courses.cid AND
              Enrolled.sid = Students.sid)
```

This is an example of a *correlated* subquery, whereas the previous, equivalent, query has a non-correlated subquery.

10. Find the name of each student and the name of each course where the student gained a mark of less than 40:

```
SELECT name, cname
FROM   Students, Courses, Enrolled
WHERE  Enrolled.sid = Students.sid AND
       Enrolled.cid = Courses.cid AND
       Enrolled.examMark < 40
```

11. Find the name of each student and the name of each course where the student gained a mark of between 38 and 42, or between 68 and 72:

```
SELECT name, cname
FROM   Students, Courses, Enrolled
WHERE  Enrolled.sid = Students.sid AND
       Enrolled.cid = Courses.cid AND
       ((Enrolled.examMark >= 38 AND
         Enrolled.examMark <= 42) OR
        (Enrolled.examMark >= 68 AND
         Enrolled.examMark <= 72))
```

12. Find the ID of all students who have the same name as some other student:

```
SELECT S1.sid
FROM   Students S1, Students S2
WHERE  S1.name = S2.name AND
       S1.sid != S2.sid
```

Note the use of *aliases* for the same table, Students.

Exercises Given the relations Students, Courses and Enrolled above, write SQL queries which:

1. Find the name of each student who is enrolled on either course BSCDB1 or course BSCDB2.
2. Find the name of each student who is enrolled on both course BSCDB1 and course BSCDB2.

Some More Examples of SQL Queries over the University Database

SQL also suppports grouping (GROUP BY) and aggregation operations (SUM, COUNT, MAX, MIN, AVG) e.g.:

1. Find how many students there are:

   ```
   SELECT COUNT(*)
   FROM   Students
   ```

   or:

   ```
   SELECT COUNT(sid)
   FROM   Students
   ```

2. Find the youngest age of any student:

   ```
   SELECT MIN(age)
   FROM   Students
   ```

3. Find the oldest age of any student:

   ```
   SELECT MAX(age)
   FROM   Students
   ```

4. Find the total exam marks for student 5000:

   ```
   SELECT SUM(examMark)
   FROM   Enrolled
   WHERE  sid='5000'
   ```

5. Find the average exam mark gained by student 5000:

   ```
   SELECT AVG(examMark)
   FROM   Enrolled
   WHERE  sid='5000'
   ```

6. Find the average exam mark gained by student 5000 in courses worth more than 1 unit:

```
SELECT AVG(examMark)
FROM   Enrolled, Courses
WHERE  sid='5000' AND
       Courses.cid = Enrolled.cid AND
       Courses.units > 1
```

7. Find how many students sat each course:

```
SELECT   cid, COUNT(sid)
FROM     Enrolled
GROUP BY cid
```

8. Find the average mark gained by each student and order the results in ascending order of the students' ids:

```
SELECT   sid, AVG(examMark)
FROM     Enrolled
GROUP BY sid
ORDER BY sid
```

9. Find the average mark gained by each student and order the results in descending order of the students' ids:

```
SELECT   sid, AVG(examMark)
FROM     Enrolled
GROUP BY sid
ORDER BY sid DESC
```

10. Find the average mark gained by each student and order the results in alphabetical order of the students' names:

```
SELECT   Endolled.sid, name, AVG(examMark)
FROM     Enrolled, Students
WHERE    Enrolled.sid = Students.sid
GROUP BY Enrolled.sid, name
ORDER BY name
```

11. Find the average mark gained by each student and order the results in increasing order of these averages:

```
SELECT   sid, AVG(examMark)
FROM     Enrolled
GROUP BY sid
ORDER BY AVG(examMark)
```

12. Same, but order the results in decreasing order of the averages:

```
SELECT   sid, AVG(examMark)
FROM     Enrolled
GROUP BY sid
ORDER BY AVG(examMark) DESC
```

SQL incorporates many other features, such as:

- statements for updating relations (INSERT INTO, DELETE FROM, UPDATE)
- schema definition and update statements (CREATE TABLE, DROP TABLE, ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, DROP VIEW)
- granting/revoking privileges to users (GRANT, REVOKE)
- integrity constraint definition statements (PRIMARY KEY, FOREIGN KEY/REFERENCES, UNIQUE, NOT NULL, CHECK)
- trigger definition statements (CREATE TRIGGER)

This makes SQL a full DDL and DML for the relational model.

There have been several successive SQL standards, e.g.:

- ANSI/ISO SQL, 1989
- SQL2 or SQL-92, 1992
  - outer joins - left, right, full
  - integrity constraints
  - assertions (constraints over more than one table)
- SQL3 or SQL-99
  - recursion
  - triggers
  - extended assertion functionality
- SQL 2003 — introduced XML-related features
- SQL 2011 — enhanced support for temporal data
- SQL 2016 — support for JSON data; pattern-matching on table rows

All major vendors' versions of SQL include ANSI/ISO SQL. Most conform to much of SQL2 and also support features from SQL3 and later versions.

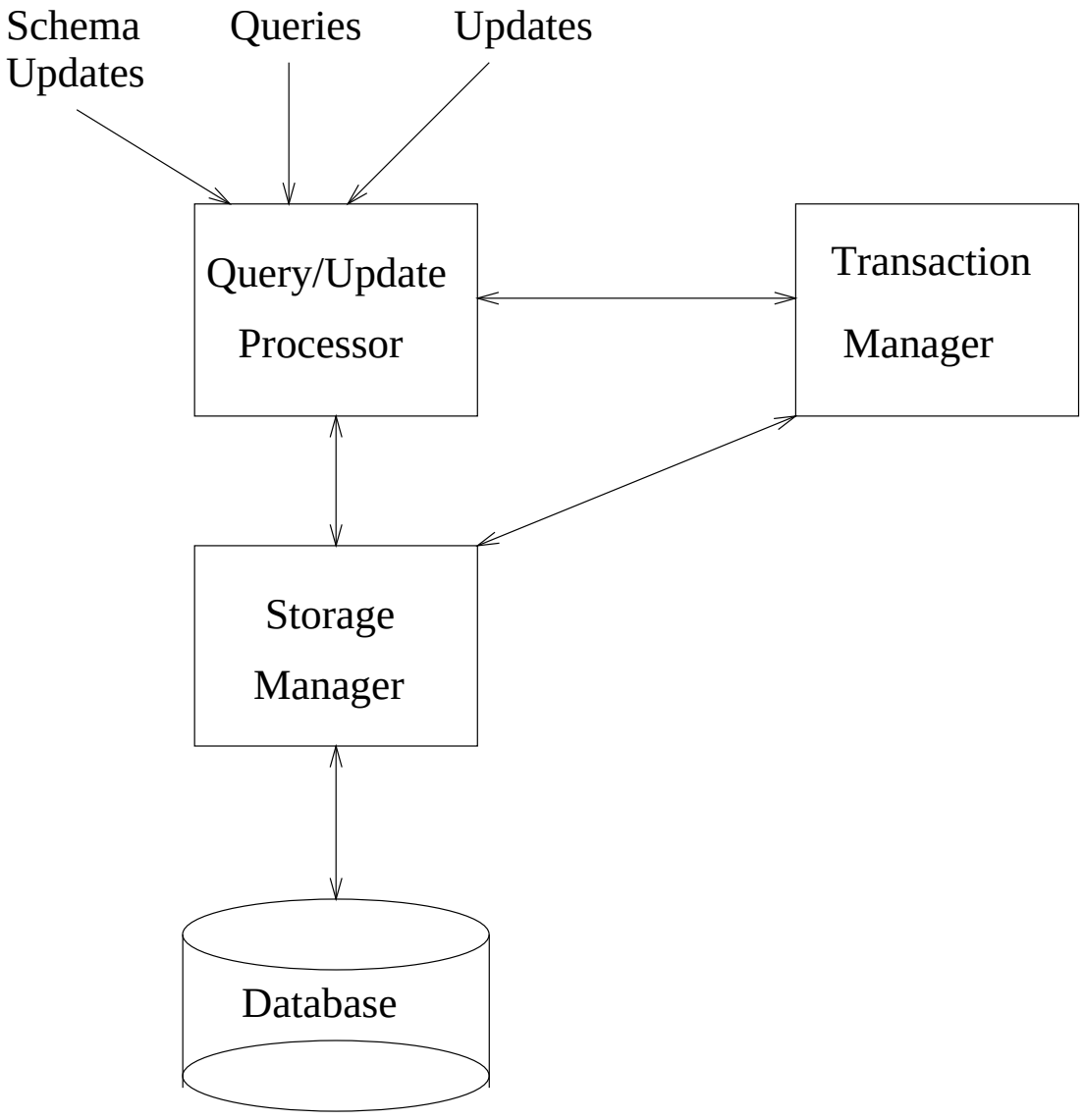

Figure 2: Major components of a DBMS

4 DBMS Architecture

(i) The Database

This consists of:

- files storing the data and indexes to that data
- meta data e.g. the physical, conceptual and external schemas
- system files e.g. authorisation tables, lock tables, logs, backup files

(ii) The Storage Manager

This performs retrieval and update operations on the database, as requested by the Transaction Manager or Query/Update Processor.

It consists of two main components:

- the **file manager** manages the reading and writing of pages onto disk;
- the **buffer manager** manages the part of the database that is currently held in main memory; it requests pages from disk, via the file manager, and returns updated pages to the file manager for writing back to disk

(iii) The Query/Update Processor

This comprises a query optimiser, a query/update evaluator, and a set of data structures and data access methods.

The evaluator processes DDL/DML statements by translating them into a sequence of data retrieval or update requests.

There may be many alternative translations for a given query — these are known as **query plans**.

The task of selecting the best, or at least a good, query plan is known as **query optimisation**.

For example, consider the query

```
SELECT sid
FROM   Enrolled, Courses
WHERE  units > 2 AND Enrolled.cid = Courses.cid
```

Three alternative query plans might be generated:

1.
```
for each tuple from Enrolled
      for each tuple from Courses
          if Courses.units > 2 and Enrolled.cid = Courses.cid
          then output Enrolled.sid
```

2.
```
for each tuple from Courses {
      retrieve all tuples in Enrolled such that Enrolled.cid = Courses.cid;
          if Courses.units > 2
          then output all Enrolled.sid values
  }
```

3.
```
for each tuple from Courses such that units > 2 {
      retrieve all tuples in Enrolled such that Enrolled.cid = Courses.cid;
          output all Enrolled.sid values
  }
```

The DBMS knows the performance of different file organisations, indexes, and algorithms — this is a set of formulae.

The DBMS also maintains statistics about actual file sizes and data distributions — these are parameters in the cost formulae.

These parameters are fed into the cost formulae in order to obtain an estimate of the cost of each query plan.

The cheapest plan is selected for execution.

(iv) The Transaction Manager

A **transaction** is a sequence of queries and/or updates submitted by a user to the database, which must appear to the user as having been executed consecutively in that sequence.

It is generally important for fast throughput of transactions to keep CPU utilisation high by working on several transactions at the same time.

While this is happening, the transaction manager must ensure that the **ACID** properties of transactions are preserved:

- **Atomicity:** either all or none of a transaction is executed.
- **Consistency:** if a transaction would be consistent if executed on its own (i.e. it would leave the database satisfying all the stated integrity constraints), then it should be consistent if executed concurrently with other transactions.
- **Isolation:** it must appear to users that transactions are being executed one at a time, even though they are interleaved for efficiency purposes.
- **Durability:** if a transaction has **committed**, its effect must not be lost if a systems failure occurs.

The two major aspects of the Transaction Manager (TM) that achieve the ACID properties are:

- **logging and recovery** — guarantees A and D, and
- **concurrency control** — guarantees C and I.

5 Object-Oriented DBMS

Commercial object-oriented DBMS have emerged during the last 15 years or so, with products such as ObjectStore, Versant, db4o.

Reasons for the development of OO DBMS include:

- The limited data types supported by the relational model (only sets of tuples of atomic values).

  These are OK for traditional record-based applications but not for applications that handle more complex objects e.g. computer aided design, computer aided software engineering, or multimedia information.

- The emergence of object-oriented programming languages such as C++ which have a more natural afinity with OO data models than with the relational data model.

There are no ANSI or ISO standards for OO databases.

However two very influential industry consortia, the Object Management Group (**OMG**) and the Object Database Management Group (**ODMG**) proposed OO database standards that have been implemented by the major OO database vendors.

The OMG proposed an OO data model, **IDL**, as one component of its **CORBA** (Common Object Request Broker Architecture) standard.

The ODMG proposed an extension to IDL, called **ODL**, and an OO query language called **OQL** as part of its ODMG-93 standard.

The following is an example of an ODL schema:

```
interface Sailor
    (extent Sailors
     key    sid) {
    attribute string sid;
    attribute string sname;
    attribute integer rating;
    attribute real age;
    relationship Set <Boat> reserves;
    integer noOfBoatsReserved(); }

interface Boat
    (extent Boats
     key    bid) {
    attribute string bid;
    attribute string name;
    attribute integer colour;
    relationship Set <Sailor> reservedBy
                 inverse Sailor::reserves; }
```

The syntax of OQL is based on that of SQL. The following is an example of an OQL query:

```
SELECT s.sname
FROM   Sailors s, s.reserves b
WHERE  b.bid="103"
```

Note the use of the **path expression** `s.reserves` which returns a set of `Boat` objects.

OQL extends SQL2 and supports some of the SQL3 features, such as structured types (sets, bags, lists arrays), reference types (pointers), path expressions, Abstract Data Types and inheritance.

6 Object-Relational DBMS

Due to

- the relatively poor performance of early OO technology
- and the plethora of applications that used relational (or older) technology

the major relational DBMS vendors extended their DBMS with OO features such as

- stored procedures and functions
- user-defined data types
- classes, including both attributes and methods (the latter being written in a 4GL or in an external PL for example)

7 XML DBMS

The rapid growth of the World Wide Web drove the need for a common data format for data exchange between applications, leading to the advent of XML.

The widespread adoption of XML resulted in a new trend: storing data directly as XML, rather than translating into/out of XML from some other data model.

This might be within "native" XML DBMSs, that is DBMSs that are specifically developed for storing XML documents e.g. eXist, Xindice, NATIX.

Or within relational or OO DBMS that have been extended to handle XML data alongside other structured data.

Native and extended DBMS support query languages such as XPath and XQuery for manipulating their XML data.

Relational DBMS that have been extended with XML capabilities typically support SQL/XML, a combined SQL+XQuery language that makes it possible to manipulate XML data and relational data within the same query.

8 Earlier DBMSs

The Network (CODASYL) Data Model

Conists of two principal elements:

- Record types: individual records are uniquely identified by system-internal identifiers (c.f. object identifiers).
- Sets: these correspond to 1-M relationships between 2 distinct record types.

A record type can be traversed. The children records of a parent record can be traversed. Parents are accessible from their children.

The Hierarchical Data Model

Record types, as in the network model.

Parent/child relationships between record types — again 1-M relationships, but are not named and can only be traversed from parent to child.

Parent/child relationships must form a forest of trees. Children records are stored clustered with their parent record.

Other hierarchical views of the data can be constructed to support other traversal pathways.

9 Trends in DBMS

There are three major directions of developments and advances in DBMS technology:

The degree of distribution and heterogeneity of the DBMS:

- distributed databases
- heterogeneous databases
- interdependent databases e.g. data warehouse architectures
- autonomous databases e.g. P2P architectures

DBMS performance:

- cost-based query optimisation
- parallel database architectures
- multi-tier architectures
- high-performance transaction processing

The variety of information managed by the DBMS e.g.

- stored procedures and functions
- triggers
- multimedia data
- spatio-temporal data
- XML and other semi-structured data
- rules and other "semantic" capabilities (e.g. RDF/OWL storage, querying, inferencing)

We will be discussing some of these topics on the ADM module in the coming weeks.

Not covered in detail in ADM, but covered in other modules of the department, are several other topics relating to data-intensive computing:

- data mining and knowledge discovery from large datasets; MapReduce, Pig, Hive, Spark in the context of data warehouse architectures; stream data processing — see Data Warehousing and Data Mining;
- distributed processing of 'big data', including MapReduce, Hadoop, Spark — see Cloud Computing;
- data analytics over large-scale datasets — see Big Data Analytics using R;
- Web languages such as HTML, XHTML, XML, JSON, XPath, XSLT (though we do look at XML and - optionally XPath - in ADM) — see Internet and Web Technologies;
- multi-tier applications development — see Component Based Software Development.

Coursework Requirements for the Advances in Data Management course

There will be several Lab Sessions during the course - attendance at these is compulsory.

Four pieces of coursework will be set during the course, on:

1. database programming using Oracle PL/SQL
2. query optimisation for SQL queries (no programming involved)
3. ontology-based data integration and the SPAQL query language
4. database programming using Neo4J or MongoDB

Your three best coursework marks will contribute (with equal weighting) to the 10% coursework component of the overall ADM assessment; the written exam will contribute 90% to the overall ADM assessment. Thus, in practice, you may choose to complete only three of the four ADM courseworks.

There will be further homework set during the course. **You are also strongly urged to do this homework**, as preparation for the exam.

Feedback on all homework and coursework will be given as the course progresses.

We draw students' attention to the MSc programme handbooks which state the penalties that apply in respect of coursework that is submitted late.

Homework 1

1. Given the relations

   Boats (bid:CHAR(10),name:CHAR(20),colour:CHAR(10))
   Sailors (sid:CHAR(10),sname:CHAR(20),rating:NUMERIC(2),age:NUMERIC(2))
   Reserves (sid:CHAR(10),bid:CHAR(10),day:DATE)

   write relational algebra queries that give the following:

   (a) The names of sailors who've reserved boat 103.
   (b) The names of sailors who've reserved a red boat.
   (c) The names of sailors who've reserved a red boat or a green boat.
   (d) The names of sailors who've reserved a red boat and a green boat.

2. Express the same queries in SQL.