Course 2. SQL-Queries

Layered Approach to Database Implementation

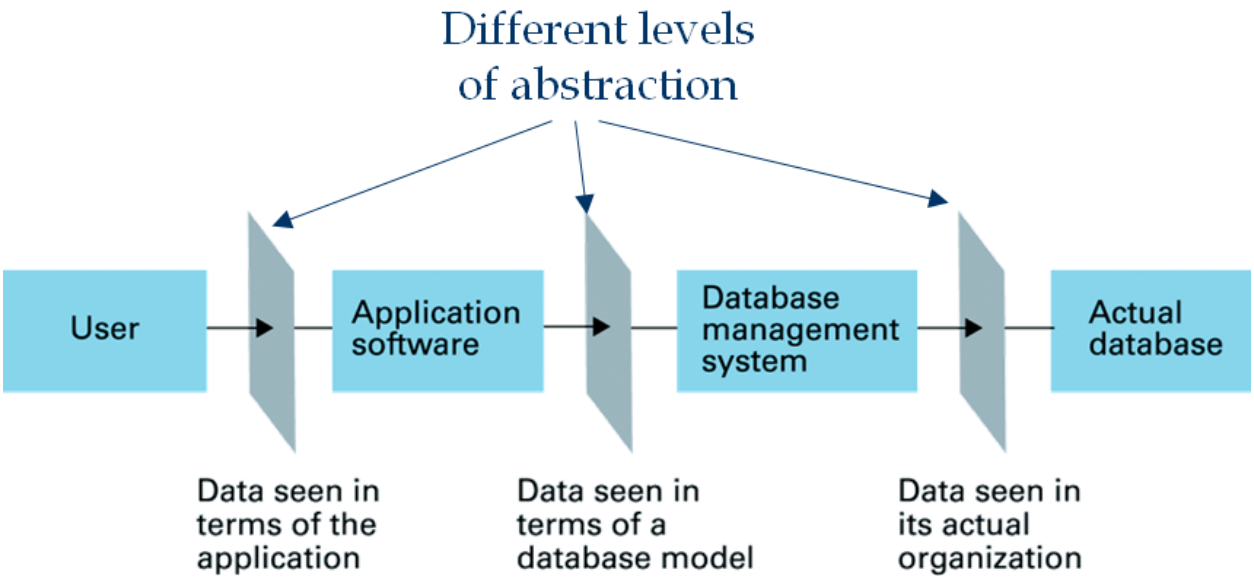

Figure 2.1.

Levels of Abstraction:

- Views describe how users see the data
- Conceptual schema defines logical structure
- Physical schema describes the files and indexes used

Example: ***Faculty*** database
Conceptual schema:
Students(sid:string, name:string, email:string, age:integer, gr:integer)
Courses(cid: string, cname: string, credits:integer)
Exams(sid:string, cid:string, grade:integer)

Physical schema:
Relations stored as unordered files.
Index on first column of Students.

External Schema (View):
Course_info(cid:string, enrollment:integer)

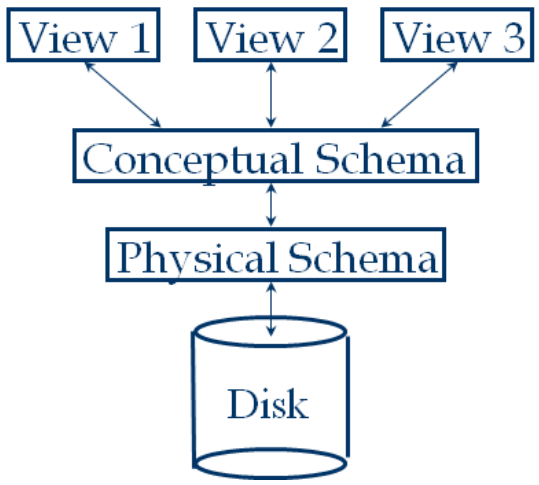

Figure 2.2.

One of the most important benefits of using a DBMS is that the applications are isolated from

how data is structured and stored.

- *Logical data independence*: Protection from changes in *logical* structure of data.
- *Physical data independence*: Protection from changes in *physical* structure of data.

Queries in a DBMS

For the sample Faculty Database, here are some questions that users may ask:

- "*What is the name of the student with sid 2833*"?
- "*What is the salary of the professor who teaches the course with cid Alg100*"?
- "*How many students are enrolled in course Alg100*"?

Such questions involving data stored in a DBMS are called ***queries*****.** A DBMS provides a specialized language, called the ***query language***, in which queries can be posed. Each query language is composed by:

- *Data Definition Language (DDL)*
  - Used to define the conceptual and internal schemas;
  - Includes constraint definition language (CDL) for describing conditions that database instances must satisfy;
  - Includes storage definition language (SDL) to influence layout of physical schema (some DBMSs);
- *Data Manipulation Language (DML)*
  - Used to describe operations on the instances of a database
  - Procedural DML (how) vs. declarative DML (what).

Query Languages for Relational DBs

SQL (Structured Query Language):

SELECT *name* FROM *Students* WHERE *age* > 20

Algebra:

$\pi_{name}(\sigma_{age > 20}\ (Students))$

Domain Calculus:

$\{<X> | \exists V\ \exists Y\ \exists Z\ \exists T : Students(V, X, Y, Z, T) \wedge Z>20\}$

T-uple Calculus:

$\{X | \exists Y : Y \in Students \wedge Y.age > 20 \wedge X.name = Y.name\}$

Databases related actors

- System Analyst: Designs entity-relationship diagram

- Database Designer: Designs logical /physical schemas
- Application Programmer
- Database Administrator
    - Handles security and authorization
    - Data availability, crash recovery
    - Database tuning as needs evolve
- System Administrator
- End Users (Naive / Sophisticated end users)

Relational Query Languages

A major strength of the relational model is that it supports simple, powerful *querying* of data. Queries can be written intuitively, and the DBMS is responsible for efficient evaluation.

- The key: precise semantics for relational queries.
- Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change.

Structured Query Language (SQL) became de-facto standard for query languages based on the relational data model. It was developed by IBM (system R) in the 1970. It was a need for a standard since it is used by many vendors

SQL Standards:

- SQL-86
- SQL-89 (minor revision)
- SQL-92 (major revision) - *1,120 pages*
- SQL-99 (major extensions) - *2,084 pages*
- SQL-2003 (SQL/XML – new section) - *3,606 pages*
- SQL-2008
- SQL- 2011

SQL Levels

Data-definition language (DDL):

- Create / destroy / alter *relations* and *views*.
- Define *integrity constraints* (IC's).

Data-manipulation language (DML)

- Allows users to pose queries

- Insert /delete / modify (update) tuples.

Access Control:

- Can grant / revoke the right to access and manipulate tables (relations / views).

SQL DML Subset

SELECT statement

A very simple example of using SELECT statement is:

```
SELECT  *
FROM  Students S
WHERE  S.age = 21
```

which returns all 21 years old students form ***Students*** table:

To find just names and email addresses, we should replace the first line with:

```
SELECT S.name, S.email
```

A very simple SQL query looks like:

```
SELECT [DISTINCT]target-list
FROM relation-list
WHERE qualification
```

Where:

- ***relation-list*** is a list of relation names (possibly with a range-variable after each name);
- ***target-list*** is a list of attributes of relations in relation-list;
- ***qualification*** contains logical expressions having comparisons (Attr op const or Attr1 op Attr2, where op is one of $<, >, =, \leq, \geq, \neq$) combined with logical operators AND, OR and NOT;
- ***DISTINCT*** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!

Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:

- Compute the cross-product of relation-list;
- Discard resulting tuples if they fail qualifications;
- Delete attributes that are not in target-list;
- If DISTINCT is specified, eliminate duplicate rows.

Range variables really needed only if the same relation appears twice in the FROM clause.

It is good style, however, to always use range variables.

So, we can write the same query in two distinct ways:

```
SELECT S.name, E.cid
FROM   Students S, Enrolled E
WHERE  S.sid=E.sid AND E.grade=10
```

or

```
SELECT name, cid
FROM   Students, Enrolled
WHERE  Students.sid=Enrolled.sid
       AND grade=10
```

The following query illustrates the use of arithmetic expressions and string pattern matching: *Find triples (of ages of students and two fields defined by expressions) for students whose names begin and end with B and contain at least three characters.*

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM   Students S
WHERE  S.name LIKE 'B_%B'
```

Observations:

- Note that AS and = are two ways to name fields in result.
- LIKE operator is used for string matching.
- `_' stands for any one character
- `%' stands for 0 or more arbitrary characters.

Join Queries

Students

sid	*name*	*email*	*age*	*gr*
1234	John	j@cs.ro	21	331
1235	Smith	s@cs.ro	22	331
1236	Anne	a@cs.ro	21	332

Courses

cid	*cname*	*credits*
Alg1	Algorithms1	7
DB1	Databases1	6
DB2	Databases2	6

Enrolled

sid	*cid*	*grade*
1234	Alg1	9
1235	Alg1	10
1237	DB2	9

Figure 2.2. Sample tables

Join variant	Sample query	Result

INNER JOIN	`SELECT S.name, C.cname`
	`FROM Students S`
	`INNER JOIN Enrolled E`
	`ON S.sid = E.sid`
	`INNER JOIN Courses C`
	`ON E.cid = C.cid`
		name | *cname*
		John | Algorithms1
		Smith | Algorithms1

LEFT OUTER JOIN	`SELECT S.name, C.cname`
	`FROM Students S`
	`LEFT OUTER JOIN Enrolled E`
	`ON S.sid = E.sid,`
	`LEFT OUTER JOIN Courses C`
	`ON E.cid = C.cid`
		name | *cname*
		John | Algorithms1
		Smith | Algorithms1
		Anne | **NULL**

RIGHT OUTER JOIN	`SELECT S.name, C.cname`
	`FROM Students S`
	`RIGHT OUTER JOIN Enrolled E`
	`ON S.sid = E.sid,`
	`INNER JOIN Courses C`
	`ON E.cid = C.cid`
		name | *cname*
		John | Algorithms1
		Smith | Algorithms1
		NULL | Databases2

FULL OUTER JOIN	`SELECT S.name, C.cname`
	`FROM Students S`
	`FULL OUTER JOIN Enrolled E`
	`ON S.sid = E.sid,`
	`FULL OUTER JOIN Courses C`
	`ON E.cid = C.cid`
		name | *cname*
		John | Algorithms1
		Smith | Algorithms1
		NULL | Databases2
		NULL | Databases1
		Anne | **NULL**

NULL values

Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable*. SQL provides a special value *null* for such situations.

The presence of *null* complicates many issues. E.g.:

- Special operators needed to check if value is/is not *null*.
- Is *rating>8* true or false when *rating* is equal to *null*? What about AND, OR and NOT connectives?

Solution: we need a 3-valued logic (**true**, **false** and ***unknown***). Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.). New operators (in particular *outer joins*) are possible/needed.

Aggregate Operators

Most used aggregate operators are (A is a table field name):

- `COUNT (*)`
- `COUNT ( [DISTINCT] A)`
- `SUM ( [DISTINCT] A)`
- `AVG ( [DISTINCT] A)`
- `MAX (A)`
- `MIN (A)`

Sample: *Get the total number of students*

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

Sample: *Get age average of group 311*

```
SELECT  AVG (S.age)
FROM  Students S
WHERE  S.gr=311
```

Sample: *Find how many groups have assigned at least one student named Bob*

```
SELECT COUNT (DISTINCT S.gr)
FROM  Students S
WHERE S.name='Bob'
```

Sample: *Find the names of oldest students*

```
SELECT S.name
FROM Students S
WHERE S.age = ANY
      (SELECT MAX(S2.age)
       FROM  Students S2)
```