Introduction to Database Systems

2023-Fall

3. User Interfaces and SQL Language

User interface of DBMS

- A DBMS must offer some interfaces to support user to access database, including:
  - Query Languages
  - Interface and maintaining tools (GUI)
  - APIs
  - Class Library
- Query Languages
  - Formal Query Language
  - Tabular Query Language
  - Graphic Query Language
  - Limited Natural Language Query Language

Example of Tabular Query Language

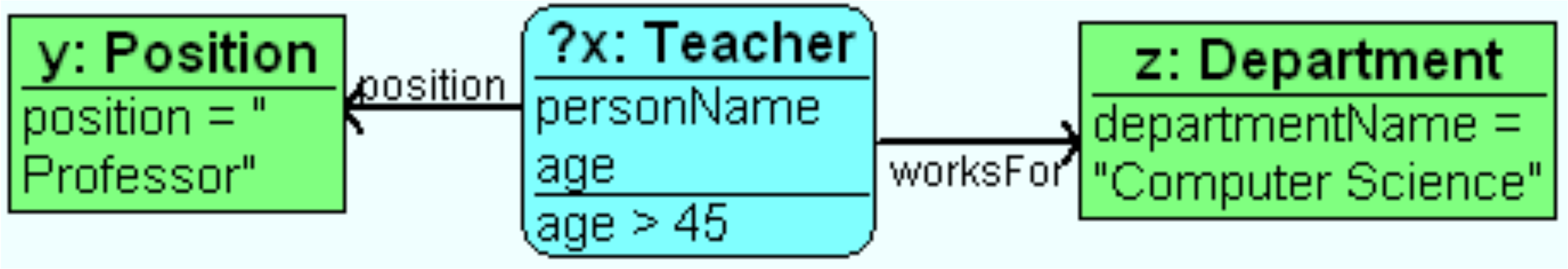
- **Find the names of all students in the department of Info. Science**

Student	Sno	Sname	Ssex	Sage	Sdept
		P.T			IS

PRINT Domain Variables Conditions

Example of Graphic Query Language

Find all Teachers, which have position=”Professor” and which have age>”45” and which work for department=”Computer Science”



User interface of DBMS

- A DBMS must offer some interfaces to support user to access database, including:
  - Query Languages
  - Interface and maintaining tools (GUI)
  - APIs
  - Class Library
- Query Languages
  - Formal Query Language
  - Tabular Query Language
  - Graphic Query Language
  - Limited Natural Language Query Language

Relational Query Languages

- Query languages: Allow manipulation and ***retrieval of data*** from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages **!=** programming languages!
  - QLs not expected to be “Turing complete”.
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.

Formal Relational Query Languages

- Two mathematical Query Languages form the basis for "**real**" languages (e.g. SQL), and for implementation:
  - ***Relational Algebra***: More operational, very useful for representing execution plans.
  - ***Relational Calculus***: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.)
- The most successful relational database language --- SQL (Structured Query Language; Standard Query Language(1986); Now SQL: 2016.)

SQL Language

SQL Roots

- Developed @IBM Research in the 1970s
  - System R project
  - Vs. Berkeley’s Quel language (Ingres project)
- Commercialized/Popularized in the 1980s
  - "Intergalactic Dataspeak”
  - IBM beaten to market by a startup called Oracle

SQL’s Persistence

- Over 40 years old!
- Questioned repeatedly
  - 90’s: Object-Oriented DBMS (OQL, etc.)
  - 2000’s: XML (Xquery, Xpath, XSLT)
  - 2010’s: NoSQL & MapReduce
- SQL keeps re-emerging as the standard
  - Even Hadoop, Spark etc. mostly used via SQL
  - May not be perfect, but it is useful

SQL Pros and Cons

- Declarative!
  - Say *what* you want, not *how* to get it
- Implemented widely
  - With varying levels of efficiency, completeness
- Constrained
  - Not targeted at Turing-complete tasks
- General-purpose and feature-rich
  - many years of added features
  - extensible: callouts to other languages, data sources

SQL Language

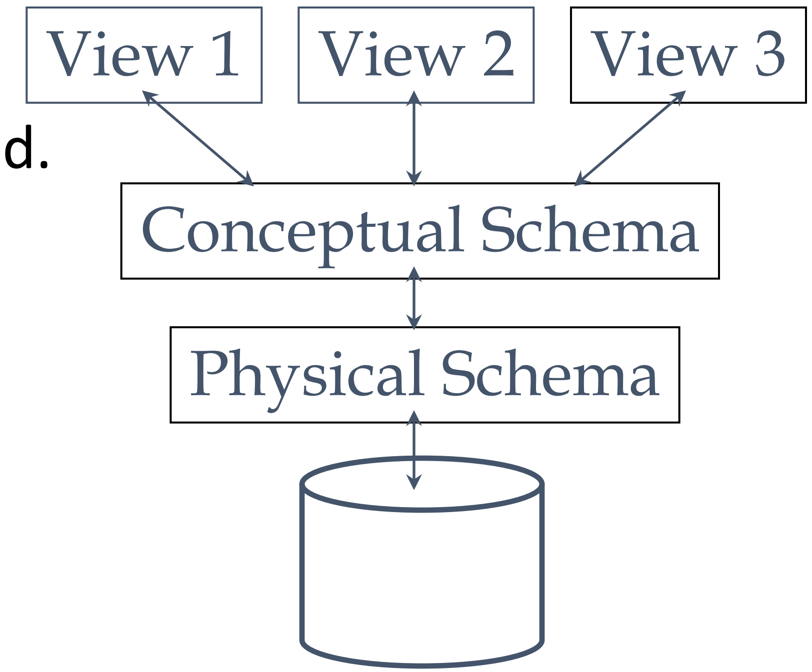
- It can be divided into four parts according to functions.
  - ***Data Definition Language (DDL),*** used to define, delete, or alter data schema.
  - Query Language (QL), used to retrieve data
  - ***Data Manipulation Language (DML),*** used to insert, delete, or update data.
  - Data Control Language (DCL), used to control user's access authority to data.
- RDBMS responsible for efficient evaluation.
  - Choose and run algorithms for declarative queries
    - Choice of algorithm must not affect query answer.

Important terms and concepts

- Base table
- View
- Data type supported
- NULL
- UNIQUE
- DEFAULT
- PRIMARY KEY
- FOREIGN KEY
- CHECK (Integration Constraint)

Levels of Abstraction: ANSI-SPARC Architecture

- Many *views*, single *conceptual (logical) schema* and *physical schema*.
  - Views describe how users see the data.
  - Conceptual schema defines logical structure
  - Physical schema describes the files and indexes used.



☛ *Schemas are defined using DDL; data is modified/queried using DML.*

Example Instances

- We will use these instances of the Sailors, Reserves and Boats relations in our examples.

R1

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

B1

<u>bid</u>	<u>bname</u>	<u>color</u>
101	tiger	red
103	lion	green
105	hero	blue

S1

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

The SQL DDL: Sailors

```
CREATE TABLE Sailors (
    sid INTEGER,
    sname CHAR(20),
    rating INTEGER,
    age FLOAT

    PRIMARY KEY (sid));
```

sid	**sname**	**rating**	**age**
1	Fred	7	22
2	Jim	2	39
3	Nancy	8	27

The SQL DDL: Boats

```
CREATE TABLE Boats (
  bid INTEGER,
  bname CHAR (20),
  color CHAR(10),
  PRIMARY KEY (bid));
```

bid	**bname**	**color**
101	Nina	red
102	Pinta	blue
103	Santa Maria	red

The SQL DDL: Reserves

```
CREATE TABLE Reserves (
  sid INTEGER,
  bid INTEGER,
  day DATE,
  PRIMARY KEY (sid, bid, day)
  FOREIGN KEY (sid) REFERENCES Sailors,
  FOREIGN KEY (bid) REFERENCES Boats);
```

sid	bid	day
1	102	9/12
2	102	9/13

Updates to tables

- **Insert**
  - **insert into** *instructor* **values** ('10211', 'Smith', 'Biology', 66000);
- **Delete**
  - Remove all tuples from the *student* relation
    - **delete from** *student*
- **Drop Table**
  - **drop table** *r*
- **Alter**
  - **alter table** *r* **add** *A D*
    - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
    - All exiting tuples in the relation are assigned *null* as the value for the new attribute.
  - **alter table** *r* **drop** *A*
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases.

DML 1

The SQL DML

Sailors

<u>sid</u>	sname	rating	age
1	Fred	7	22
2	Jim	2	39
3	Nancy	8	27

- Find all 27-year-old sailors:

```
SELECT *
FROM  Sailors AS S
WHERE S.age=27;
```

- To find just names and rating, replace the first line to:

```
SELECT S.sname, S.rating
```

Basic Single-Table Queries

SELECT [*DISTINCT*] <*column expression list*>
FROM <*single table*>
[**WHERE** <*predicate*>]

- Simplest version is straightforward
  - Produce all tuples in the table that satisfy the predicate
  - Output the expressions in the SELECT list
  - Expression can be a column reference, or an arithmetic expression over column refs

SELECT DISTINCT

SELECT [DISTINCT] <*column expression list*>
FROM <*single table*>
[WHERE <*predicate*>]

SELECT DISTINCT S.name, S.gpa
FROM students S
WHERE S.dept = 'CS'

- DISTINCT specifies removal of duplicate rows before output
- Can refer to the students table as “S”, this is called an alias

A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND bid=103
```

OR

```
SELECT  sname
FROM    Sailors, Reserves
WHERE   Sailors.sid=Reserves.sid
        AND bid=103
```

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

Expressions and Strings

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

- Illustrates use of arithmetic expressions and string pattern matching: *Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.*
- AS and = are two ways to name fields in result.
- LIKE is used for string matching. '_' stands for any one character and '%' stands for 0 or more arbitrary characters.

Tuple Relational Calculus

- *Query* has the form:

  { t[<*attribute list*>] | P(t)}

- t is called *tuple variable*.

✓*Answer* includes all tuples t<*attribute list*> that make the *formula* P(t) be *true*.

✓Example query: Find all sailors’ name whose rating above 7 and younger than 50;

  $\{ t[\ N\ ] \mid t \in \mathit{Sailors} \wedge t.T>7 \wedge t.A<50 \}$

ORDER BY

```
SELECT S.name, S.gpa, S.age*2 AS a2
FROM Students S
WHERE S.dept = 'CS'
ORDER BY S.gpa, S.name, a2;
```

- ORDER BY clause specifies output to be sorted
  - ***Lexicographic*** ordering
- Obviously must refer to columns in the output
  - Note the AS clause for naming output columns!

ORDER BY, Pt. 2

```
SELECT  S.name, S.gpa, S.age*2 AS a2
FROM Students S
WHERE S.dept = 'CS'
ORDER BY S.gpa DESC, S.name ASC, a2;
```

- Ascending order by default, but can be overridden
  - DESC flag for descending, ASC for ascending
  - Can mix and match, lexicographically

LIMIT

```
SELECT  S.name, S.gpa, S.age*2 AS a2
FROM Students S
WHERE S.dept = 'CS'
ORDER BY S.gpa DESC, S.name ASC, a2;
LIMIT 3 ;
```

- Only produces the first <integer> output rows
- Typically used with ORDER BY
  - Otherwise the output is *non-deterministic*
  - Not a “pure” declarative construct in that case – output set depends on algorithm for query processing

Aggregate Operators

- Significant extension of relational algebra.
  - COUNT (*)
  - COUNT ([DISTINCT] A)
  - SUM ([DISTINCT] A)
  - AVG ([DISTINCT] A)
  - MAX (A)
  - MIN (A)
- *A* is single column

Aggregates

SELECT [DISTINCT] **AVG**(S.gpa)
FROM Students S
WHERE S.dept = 'CS'

- Before producing output, compute a summary (a.k.a. an *aggregate*) of some arithmetic expression
- Produces **1 row** of output
  - with one column in this case
- Other aggregates: SUM, COUNT, MAX, MIN

Examples of Aggregate Operators

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

```
SELECT  COUNT (DISTINCT S.rating)
FROM  Sailors S
WHERE S.sname='Bob'
```

```
SELECT  AVG (S.age)
FROM  Sailors S
WHERE  S.rating=10
```

```
SELECT  AVG (DISTINCT S.age)
FROM  Sailors S
WHERE  S.rating=10
```

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.rating= (SELECT  MAX(S2.rating)
                  FROM  Sailors S2)
```

Find the name and age of the oldest sailor(s)

- The first query is illegal! (We'll look into the reason a bit later, when we discuss GROUP BY.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

```
SELECT  S.sname, MAX (S.age)
FROM  Sailors S
```

```
SELECT  S.sname, S.age
FROM  Sailors S
WHERE  S.age =
          (SELECT  MAX (S2.age)
           FROM  Sailors S2)
```

```
SELECT  S.sname, S.age
FROM  Sailors S
WHERE  (SELECT  MAX (S2.age)
           FROM  Sailors S2)
           = S.age
```

Motivation for Grouping

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Consider: *Find the age of the youngest sailor for each rating level.*
  - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

For i = 1, 2, ... , 10:

```
SELECT  MIN (S.age)
FROM  Sailors S
WHERE  S.rating = i
```

GROUP BY

SELECT [DISTINCT] **AVG**(S.gpa), S.dept
FROM Students S
GROUP BY S.dept

- Partition table into groups with same GROUP BY column values
  - Can group by ***a list of columns***
- Produce an aggregate result per group
  - Cardinality of output = # of distinct group values
- Note: can put grouping columns in SELECT list

HAVING

SELECT [DISTINCT] **AVG**(S.gpa), S.dept
FROM Students S
GROUP BY S.dept
HAVING COUNT(*) > 2

- The HAVING predicate ***filters*** groups
- HAVING is applied ***after*** grouping and aggregation
  - Hence can contain anything that could go in the SELECT list
  - I.e. aggs or **GROUP BY** columns
- HAVING can only be used in aggregate queries
- It's an optional clause

Putting it all together

```
SELECT S.dept, AVG(S.gpa), COUNT(*)
FROM Students S
WHERE S.gender = 'F'
GROUP BY S.dept
HAVING COUNT(*) >= 2
ORDER BY S.dept;
```

Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors

```
SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 <u>such</u> sailors.

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

rating	minage
3	25.5
7	35.0
8	25.5

DISTINCT Aggregates

Are these the same or different?

```
SELECT COUNT(DISTINCT S.name)
FROM Students S
WHERE S.dept = 'CS';
```

```
SELECT DISTINCT COUNT(S.name)
FROM Students S
WHERE S.dept = 'CS';
```

What Is This Asking For?

```
SELECT S.name, AVG(S.gpa)
FROM Students S
GROUP BY S.dept;
```

SQL DML:
General Single-Table Queries

SELECT [DISTINCT] <*column expression list*>
FROM <*single table*>
[**WHERE** <*predicate*>]
[**GROUP BY** <*column list*>
[**HAVING** <*predicate*>]]
[**ORDER BY** <*column list*>]
[**LIMIT** <integer>];

Summary

- Relational model has **well-defined query semantics**
- Modern SQL extends “pure” relational model
  (some extra goodies for duplicate row, non-atomic types… more in next lecture)
- Typically, many ways to write a query
  - DBMS figures out a fast way to execute a query, regardless of how it is written.