

# CPSC 304

## **Introduction to Database Systems**

---

Structured Query Language (SQL)

Textbook Reference  
Database Management Systems: Chapter 5

# Databases: the continuing saga

---



When last we left databases...

- We had decided they were great things
- We knew how to conceptually model them in ER diagrams
- We knew how to logically model them in the relational model
- We knew how to normalize our database relations
- We could formally specify queries in Relational Algebra

Now: how do most people write queries? SQL!

# Learning Goals

---



- Given the schemas of a relation, create SQL queries using: SELECT, FROM, WHERE, EXISTS, NOT EXISTS, UNIQUE, NOT UNIQUE, ANY, ALL, DISTINCT, GROUP BY and HAVING.
- Show that there are alternative ways of coding SQL queries to yield the same result. Determine whether or not two SQL queries are equivalent.
- Given a SQL query and table schemas and instances, compute the query result.
- Translate a query between SQL and RA.
- Comment on the relative expressive power of SQL and RA.
- Explain the purpose of NULL values and justify their use. Also describe the difficulties added by having nulls.
- Create and modify table schemas and views in SQL.
- Explain the role and advantages of embedding SQL in application programs.
- Write SQL for a small-to-medium sized programming application that requires database access.
- Identify the pros and cons of using general table constraints (e.g., CONSTRAINT, CHECK) and triggers in databases.

# Coming up in SQL...

---

- Data Definition Language (reminder)
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Modification of the Database
- Views
- Integrity Constraints
- Putting SQL to work in an application

# The SQL Query Language

---

- Need for a standard since relational queries are used by many vendors
- Consists of several parts:
  - Data Definition Language (DDL)  
(a blast from the past (Chapter 3))
  - Data Manipulation Language (DML)
    - Data Query
    - Data Modification

# Creating Tables in SQL(DDL) Revisited

- A SQL relation is defined using the **create table** command:

**create table**  $r$  ( $A_1 D_1, A_2 D_2, \dots, A_n D_n,$   
                  (integrity-constraint<sub>1</sub>),  
                  ...,  
                  (integrity-constraint<sub>k</sub>))

- *Integrity constraints can be:*

- *primary and candidate keys*
- *foreign keys*

- Example:

```
CREATE TABLE Student
(sid    CHAR(20),
 name  CHAR(20),
 address CHAR(20),
 phone CHAR(8),
 major  CHAR(4),
PRIMARY KEY (sid))
```

# Domain Types in SQL

## Reference Sheet

---

- **char(*n*)**. Fixed length character string with length *n*.
- **varchar(*n*)**. Variable length character strings, with maximum length *n*.
- **int**. Integer (machine-dependent).
- **smallint**. Small integer (machine-dependent).
- **numeric(*p*,*d*)**. Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point.
- **real, double precision**. Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(*n*)**. Floating point number, with user-specified precision of at least *n* digits.
- Null values are allowed in all the domain types.  
To prohibit null values declare attribute to be **not null**
- **create domain** in SQL-92 and 99 creates user-defined domain types  
**create domain *person-name* char(20) not null**

# Date/Time Types in SQL

## Reference Sheet

---

- **date.** Dates, containing a (4 digit) year, month and date
  - E.g. **date** '2001-7-27'
- **time.** Time of day, in hours, minutes and seconds.
  - E.g. **time** '09:00:30'      **time** '09:00:30.75'
- **timestamp:** date plus time of day
  - E.g. **timestamp** '2001-7-27 09:00:30.75'
- **Interval:** period of time
  - E.g. Interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values
- Relational DBMS offer a variety of functions to
  - extract values of individual fields from date/time/timestamp
  - convert strings to dates and vice versa
  - For instance in Oracle (date is a timestamp):
    - TO\_CHAR( date, format)
    - TO\_DATE( string, format)
    - format looks like: 'DD-Mon-YY HH:MI.SS'



# Running Example (should look familiar)

---

Movie(MovieID, Title, Year)

StarsIn(MovieID, StarID, Character)

MovieStar(StarID, Name, Gender)

# Basic SQL Query

- SQL is based on set and relational operations
- A typical SQL query has the form:

**SELECT**  $A_1, A_2, \dots, A_n$   
**FROM**  $r_1, r_2, \dots, r_m$   
**WHERE**  $P$

SELECT	<i>target-list</i>
FROM	<i>relation-list</i>
WHERE	<i>qualification</i>

- $A_i$ s represent attributes
  - $r_i$ s represent relations
  - $P$  is a predicate.
- $\pi \rightarrow$  SELECT clause  
 $\sigma \rightarrow$  WHERE clause  
 $\bowtie \rightarrow$  FROM and WHERE clause
- The result of a SQL query is a table (relation)
  - By default, duplicates are not eliminated in SQL relations, which are **bags** or **multisets** and not sets
  - Let's compare to relational algebra...

# Basic SQL/RA Comparison example 1

---

- Find the titles of movies

# Basic SQL/RA Comparison example 1

---

- Find the titles of movies

$\pi_{\text{Title}}(\text{Movie})$

- In SQL,  $\pi$  is in the SELECT clause
- Select only a subset of the attributes

```
SELECT Title  
FROM   Movie
```

- Note duplication can happen!
  - You can get the same value multiple times

# Basic SQL/RA Comparison example 1

---

- You can also refer to an attribute by (relation name).(attribute name)

```
SELECT  Movie.Title  
FROM    Movie
```

- Since we are only working with one relation, you don't need to specify where the attribute comes from
  - This is useful when you have multiple relations that share the same attribute name

## In SQL, $\sigma$ is in *WHERE* clause

---

```
SELECT *  
FROM   Movie  
WHERE  Year > 1939
```

You can use:

- attribute names of the relation(s) used in the FROM.

- comparison operators: =, <>, <, >, <=, >=

- apply arithmetic operations: rating\*2

- operations on strings (e.g., "||" for concatenation).

- Lexicographic order on strings.

- Pattern matching: s LIKE p

- Special stuff for comparing dates and times.

## Basic SQL/RA Comparison example 2

---

Find female movie stars

## Basic SQL/RA Comparison example 2

---

Find female movie stars

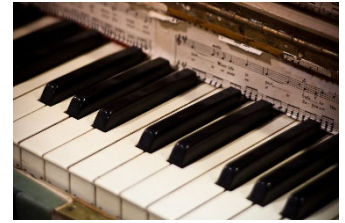
$\sigma_{\text{Gender} = \text{'female'}} \text{MovieStar}$

```
SELECT *  
FROM   MovieStar  
WHERE  Gender='female'
```



# Selection & Projection – together forever in SQL

---



We can put these together:

- What are the names of female movie stars?
- What are the titles of movies from prior to 1939?

# Selection example (dates)

---

reserves

SID	BID	Day
22	101	2010-10-10
22	102	2010-10-10
22	103	2010-10-08
22	104	2010-07-10
31	102	2010-11-10
31	103	2010-11-06
31	104	2010-11-12
58	102	2010-11-08
58	103	2010-11-12

```
SELECT *  
FROM reserves  
WHERE day < DATE'2010-11-01'
```

SID	BID	Day
22	101	2010-10-10
22	102	2010-10-10
22	103	2010-10-08
22	104	2010-07-10

## Basic SQL/RA comparison example 3

---

- Find the person names and character names of those who have been in movies
- In order to do this we need to use joins.  
How can we do joins in SQL?
  - $\pi \rightarrow$  SELECT clause
  - $\sigma \rightarrow$  WHERE clause
  - $\bowtie \rightarrow$  FROM and WHERE clause

# Joins in SQL

---

```
SELECT Character, Name  
FROM StarsIn s, MovieStar m  
WHERE s.StarID = m.StarID
```

- Cross product specified by FROM clause
- Can alias relations (e.g., “StarsIn s”)
- Conditions specified in WHERE clause

# So how does a typical SQL query relate to relational algebra then?

---

SQL:

**SELECT**  $A_1, A_2, \dots, A_n$   
**FROM**  $r_1, r_2, \dots, r_m$   
**WHERE**  $P$

Is approximately equal to  
Relational algebra

$$\pi_{A_1, A_2, \dots, A_n}(\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

Difference? Duplicates.  
Remove them? **Distinct**

# Using DISTINCT

---

- Find the names of actors who have been in at least one movie

```
SELECT  DISTINCT Name
FROM    StarsIn S, MovieStar M
WHERE   S.StarID = M.StarID
```

- Would removing DISTINCT from this query make a difference?
- Note: on the exams, if we ask for a general question like “find all the names”, we expect duplicates to be removed. When in doubt, keep the DISTINCT rather than reasoning through if you need it!

# Join Example

---

- Find the names of all movie stars who have been in a movie

# Join Example

- Find the names of all movie stars who have been in a movie

SELECT Name

FROM StarsIn S, MovieStar M

WHERE S.StarID = M.StarID

Is this totally correct?

StarID	Name	Gender
1	Harrison Ford	Male
2	Vivian Leigh	Female
3	Judy Garland	Female

MovieID	StarID	Character
1	1	Han Solo
4	1	Indiana Jones
2	2	Scarlett O'Hara
3	3	Dorothy Gale

Harrison Ford will appear twice



# Join Example

---

- Find the names of all movie stars who have been in a movie

```
SELECT Name  
FROM StarsIn S, MovieStar M  
WHERE S.StarID = M.StarID
```

Is this totally correct?

```
SELECT DISTINCT Name  
FROM StarsIn S, MovieStar M  
WHERE S.StarID = M.StarID
```

What if two movie stars  
had the same name?

- What if I run the following query?

```
SELECT DISTINCT StarID, Name  
FROM StarsIn S, MovieStar M  
WHERE S.StarID = M.StarID
```

Error: Column StarID  
is ambiguous

# Renaming Attributes in Result

---

- SQL allows renaming relations and attributes using the **as** clause:  
*old-name as new-name*
- Example: Find the title of movies and the IDs of all actors in them, and rename “StarID” to “ID”

```
SELECT  Title, StarID AS ID
FROM    StarsIn S, Movie M
WHERE   M.MovieID = S.MovieID
```

# Congratulations:

## You know select-project-join queries

---

- Very common subset to talk about
  - You saw it in tutorial
- Can do many (but not all) useful things

SQL is *declarative*, not procedural  
how do we know? Lets see what  
procedural would look like...

# Conceptual Procedural Evaluation Strategy

---

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

# Example of Conceptual Procedural Evaluation

```
SELECT Name  
FROM MovieStar M, StarsIn S  
WHERE S.StarID = M.StarID AND MovieID = 276
```

join

selection

MovieStar X StarsIn

(StarID)	Name	Gender	MovieID	(StarID)	Character
1273	Nathalie Portman	Female	272	1269	Leigh Anne Touhy
1273	Nathalie Portman	Female	273	1270	Mary
1273	Nathalie Portman	Female	274	1271	King George VI
1273	Nathalie Portman	Female	276	1273	Nina Sayers
...	...	...	...	...	...

# New Students Example

---

- Class(name,meets\_at,room,fid)
- Student(snum,sname,major,standing,age)
- Enrolled(snum,cname)
- Faculty(fid,fname,deptid)

# Class Table

Name	Meets_at	Room	FID
Data Structures	MWF 10	R128	489456522
Database Systems	MWF 12:30-1:45	1320 DCL	142519864
Operating System Design	TuTh 12-1:20	20 AVW	489456522
Archaeology of the Incas	MWF 3-4:15	R128	248965255
Aviation Accident Investigation	TuTh 1-2:50	Q3	011564812
Air Quality Engineering	TuTh 10:30-11:45	R15	011564812
Introductory Latin	MWF 3-4:15	R12	248965255
American Political Parties	TuTh 2-3:15	20 AVW	619023588
Social Cognition	Tu 6:30-8:40	R15	159542516
Perception	MTuWTh 3	Q3	489221823
Multivariate Analysis	TuTh 2-3:15	R15	090873519
Patent Law	F 1-2:50	R128	090873519
Urban Economics	MWF 11	20 AVW	489221823
Organic Chemistry	TuTh 12:30-1:45	R12	489221823
Marketing Research	MW 10-11:15	1320 DCL	489221823
Seminar in American Art	M 4	R15	489221823
Orbital Mechanics	MWF 8 1320	DCL	011564812
Dairy Herd Management	TuTh 12:30-1:45	R128	356187925
Communication Networks	MW 9:30-10:45	20 AVW	141582651
Optical Electronics	TuTh 12:30-1:45	R15	254099823
Introduction to Math	TuTh 8-9:30	R128	489221823

# Student Table

SNUM	SNAME	MAJOR	ST	AGE
51135593	Maria White	English	SR	21
60839453	Charles Harris	Architecture	SR	22
99354543	Susan Martin	Law	JR	20
112348546	Joseph Thompson	Computer Science	SO	19
115987938	Christopher Garcia	Computer Science	JR	20
132977562	Angela Martinez	History	SR	20
269734834	Thomas Robinson	Psychology	SO	18
280158572	Margaret Clark	Animal Science	FR	18
301221823	Juan Rodriguez	Psychology	JR	20
318548912	Dorthy Lewis	Finance	FR	18
320874981	Daniel Lee	Electrical Engineering	FR	17
322654189	Lisa Walker	Computer Science	SO	17
348121549	Paul Hall	Computer Science	JR	18
351565322	Nancy Allen	Accounting	JR	19
451519864	Mark Young	Finance	FR	18
455798411	Luis Hernandez	Electrical Engineering	FR	17
462156489	Donald King	Mechanical Engineering	SO	19
550156548	George Wright	Education	SR	21
552455318	Ana Lopez	Computer Engineering	SR	19
556784565	Kenneth Hill	Civil Engineering	SR	21
567354612	Karen Scott	Computer Engineering	FR	18
573284895	Steven Green	Kinesiology	SO	19
574489456	Betty Adams	Economics	JR	20
578875478	Edward Baker	Veterinary Medicine	SR	21



# Enrolled Table

**SNUM**

**CNAME**

112348546	Database Systems
115987938	Database Systems
348121549	Database Systems
322654189	Database Systems
552455318	Database Systems
455798411	Operating System Design
552455318	Operating System Design
567354612	Operating System Design
112348546	Operating System Design
115987938	Operating System Design
322654189	Operating System Design
567354612	Data Structures
552455318	Communication Networks
455798411	Optical Electronics
455798411	Organic Chemistry
301221823	Perception
301221823	Social Cognition
301221823	American Political Parties
556784565	Air Quality Engineering
99354543	Patent Law
574489456	Urban Economics

# Faculty Table

FID	FNAME	DEPTID
142519864	I. Teach	20
242518965	James Smith	68
141582651	Mary Johnson	20
011564812	John Williams	68
254099823	Patricia Jones	68
356187925	Robert Brown	12
489456522	Linda Davis	20
287321212	Michael Miller	12
248965255	Barbara Wilson	12
159542516	William Moore	33
090873519	Elizabeth Taylor	11
486512566	David Anderson	20
619023588	Jennifer Thomas	11
489221823	Richard Jackson	33
548977562	Ulysses Teach	20

# Running Examples

---

Movie(MovieID, Title, Year)  
StarsIn(MovieID, StarID, Character)  
MovieStar(StarID, Name, Gender)

Student(snum, sname, major, standing, age)  
Class(name, meets\_at, room, fid)  
Enrolled(snum, cname)  
Faculty(fid, fname, deptid)

# What kinds of queries can you answer so far?

---

- Find the student ids of those who have taken a course named “Database Systems”
- Find the names of all classes taught by Elizabeth Taylor

# What kinds of queries can you answer so far?

Find the departments that have more than one faculty member (express not equal by “<>”)

```
SELECT DISTINCT f1.deptid
FROM faculty f1, faculty f2
WHERE f1.fid <> f2.fid AND
      f1.deptid = f2.deptid
```

f1

<u>fid</u>	fname	Deptid
90873519	Elizabeth Taylor	11
619023588	Jennifer Thomas	11
...	...	...

That is why  
renaming is  
important

f2

<u>fid</u>	fname	Deptid
90873519	Elizabeth Taylor	11
619023588	Jennifer Thomas	11
...	...	...

A good example for using the same table twice in a query

# What kinds of queries can you answer so far?

Find the departments that have more than one faculty member (express not equal by “<>”)

```
SELECT DISTINCT f1.deptid
FROM faculty f1, faculty f2
WHERE f1.fid <> f2.fid AND
      f1.deptid = f2.deptid
```

f1

<u>fid</u>	fname	Deptid
90873519	Elizabeth Taylor	11
619023588	Jennifer Thomas	11
...	...	...

That is why  
renaming is  
important

f2

<u>fid</u>	fname	Deptid
90873519	Elizabeth Taylor	11
619023588	Jennifer Thomas	11
...	...	...

A good example for using the same table twice in a query

Do I need Distinct?

# String comparisons

---

What are the student ids of those who have taken a course with “System” in the name?

# A string walks into a bar...

---

```
SELECT DISTINCT snum  
FROM   enrolled  
WHERE  cname LIKE '%System%'
```

- **LIKE** is used for string matching:
  - ‘**\_**’ stands for any one character and
  - ‘**%**’ stands for 0 or more arbitrary characters.
- SQL supports string operations such as
  - concatenation (using “||”)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

Clicker question: do we need DISTINCT?

**a. Yes** b. No



# Ordering of Tuples

---

- List in alphabetic order the names of actors who were in a movie in 1939

SELECT distinct Name

FROM Movie, StarsIn, MovieStar

WHERE Movie.MovieID = StarsIn.MovieID and StarsIn.StarID  
= MovieStar.StarID and year = 1939

ORDER BY Name

Order is specified by:

- **desc** for descending order
- **asc** for ascending order (default)
- E.g. **order by Name desc**
- You can order within order: for example, ... “ORDER BY Year, Name” would first order by Year, then Name within years

# Set Operations

---

- **union**, **intersect**, and **except** correspond to the relational algebra operations  $\cup$ ,  $\cap$ ,  $-$ .
- **Each automatically eliminates duplicates;**  
To retain all duplicates use the corresponding multiset versions:  
**union all**, **intersect all** and **except all**.
- Suppose a tuple occurs  $m$  times in  $r$  and  $n$  times in  $s$ , then, it occurs:
  - $m + n$  times in  $r$  **union all**  $s$
  - $\min(m, n)$  times in  $r$  **intersect all**  $s$
  - $\max(0, m - n)$  times in  $r$  **except all**  $s$

Find IDs of MovieStars who've been in a  
movie in 1944 *or* 1974

---

# Set Operations: Intersect

---

Example: Find IDs of stars who have been in a movie in 1944 and 1974.

- **INTERSECT:** Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- In SQL/92, but some systems don't support it.

# Rewriting INTERSECT with Joins

---

- Example: Find IDs of stars who have been in a movie in 1944 and 1974 without using **INTERSECT**.

# Set Operations: EXCEPT

---

- Find the sids of all students who took Operating System Design but did not take Database Systems

## But what about...

---

- Select the IDs of all students who have not taken “Operating System Design”

# Motivating Example for Nested Queries

---

*Find ids and names of female stars who have been in movie with ID 28:*



# Nested Queries

---

- A very powerful feature of SQL:

```
SELECT   $A_1, A_2, \dots, A_n$   
FROM     $R_1, R_2, \dots, R_m$   
WHERE   condition
```

- A nested query is a query that has another query embedded with it.
  - A **SELECT**, **FROM**, **WHERE**, or **HAVING** clause can itself contain an SQL query!
  - Being part of the **WHERE** clause is the most common

# Nested Queries (IN/Not IN)

---

*Find ids and names of stars who have been in movie with ID 28:*

# Nested Queries (IN/Not IN)

---

*Find ids and names of female stars who have been in movie with ID 28:*

```
SELECT M.StarID, M.Name  There's also NOT IN
FROM   MovieStar M
WHERE  M.Gender = 'female' AND
      M.StarID IN (SELECT S.StarID
                  FROM   StarsIn S
                  WHERE  MovieID=28)
```

- To find stars who have *not* been in movie 28, use **NOT IN**.
- To understand nested query semantics, think of a nested loops evaluation:
  - For each *MovieStar* tuple, check the qualification by computing the subquery.

# Nested Queries (IN/Not IN)

*Find ids and names of female stars who have been in movie with ID 28:*

```
SELECT M.StarID, M.Name
FROM   MovieStar M
WHERE  M.Gender = 'female' AND
       M.StarID IN (SELECT S.StarID
                    FROM   StarsIn S
                    WHERE  MovieID=28)
```

- In this example inner query does not depend on the outer query so it could be computed just once.
- Think of this as a function that has no parameters.

```
SELECT S.StarID
FROM   StarsIn S
WHERE  MovieID=28
```

StarID
1026
1027

```
SELECT M.StarID, M.Name
FROM   MovieStar M
WHERE  M.Gender = 'female' AND
       M.StarID IN
       (1026,1027)
```

## Rewriting EXCEPT Queries Using In

- Using nested queries, find the sids of all students who took Operating System Design but did not take Database Systems

# Rewriting INTERSECT Queries Using IN

---

*Find IDs of stars who have been in movies in 1944 and 1974*

# Nested Queries with Correlation

## Same idea, subtle difference

---

*Find names of stars who have been in movie w/ ID 28:*

```
SELECT M.Name
FROM   MovieStar M
WHERE  EXISTS (SELECT *
                FROM   StarsIn S
                WHERE  MovieID=28 AND S.StarID = M.StarID)
```

- **EXISTS:** *returns true if the set is not empty.*
- **UNIQUE:** *returns true if there are no duplicates.*
- Illustrates why, in general, subquery must be re-computed for each StarsIn tuple.

# SQL EXISTS Condition

---

- The SQL EXISTS condition is used in combination with a subquery and is considered to be met, if the subquery returns at least one row. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement.
- We can also use NOT EXISTS



# SQL EXISTS Condition

---

- Using the EXISTS/ NOT EXISTS operations and correlated queries, find the name and age of the oldest student(s)

# SQL EXISTS Condition

---

- Using the EXISTS/ NOT EXISTS operations and correlated queries, find the name and age of the oldest student(s)

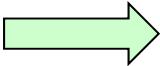
```
SELECT sname, age
FROM student s2
WHERE NOT EXISTS(SELECT *
                  FROM student s1
                  WHERE s1.age > s2.age)
```

# SQL EXISTS

```
SELECT sname, age
FROM student s2
WHERE NOT EXISTS(SELECT *
                  FROM student s1
                  WHERE s1.age > s2.age)
```

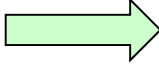


Does there exist a tuple in s1 such that the age of the s1 tuple is greater than the age of the tuple in s2?

Student s2



snum	Name	...
...	...	...
...	...	...
...	...	...

Student s1



snum	Name	...
...	...	...
...	...	...
...	...	...

# More on Set-Comparison Operators

---

- We've already seen **IN** and **EXISTS**. Can also use **NOT IN**, **NOT EXISTS**.
- Also available: **op ANY**, **op ALL**, where **op** is one of: **>**, **<**, **=**, **<=**, **>=**, **<>**
- Find movies made after "Fargo"

# More on Set-Comparison Operators

---

- We've already seen **IN** and **EXISTS**. Can also use **NOT IN**, **NOT EXISTS**.
- Also available: **op ANY**, **op ALL**, where **op** is one of: **>**, **<**, **=**, **<=**, **>=**, **<>**
- Find movies made after "Fargo"

```
SELECT *  
FROM Movie  
WHERE year > ANY (SELECT year  
                  FROM Movie  
                  WHERE Title = 'Fargo')
```

Just returning one column

If we have multiple movies named Fargo then we can use ALL instead of ANY

# Example

---

- Using the any or all operations, find the name and age of the oldest student(s)

# Reminder: Division A/B

*A*

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

*B1*

pno
p2

*B2*

pno
p2
p4

*B3*

pno
p1
p2
p4

*A/B1*

sno
s1
s2
s3
s4

*A/B2*

sno
s1
s4

*A/B3*

sno
s1

# Division in SQL

Find students who've  
taken all classes.

(method 1)

```
SELECT sname
FROM Student S
WHERE NOT EXISTS
    ((SELECT C.name
      FROM Class C)
    EXCEPT
    (SELECT E.cname
      FROM Enrolled E
      WHERE E.snum=S.snum))
```

All classes  
Classes  
taken by S

The hard way (without EXCEPT): (method 2)

```
SELECT sname
FROM Student S
WHERE NOT EXISTS (SELECT C.name
                  FROM Class C
                  WHERE NOT EXISTS (SELECT E.snum
                                    FROM Enrolled E
                                    WHERE C.name=E.cname
                                    AND E.snum=S.snum))
```

Method 2:

*select Student S such that ...  
there is no Class C...*

*which is not taken by S*



# Division in SQL

## (method 1- use EXCEPT)

---

Find students who've taken all classes.

```
SELECT sname
FROM   Student S
WHERE NOT EXISTS
      ((SELECT C.name    All classes
        FROM   Class C)
      EXCEPT
      (SELECT E.cname
        FROM   Enrolled E
        WHERE E.snum=S.snum))
```

# Division in SQL

## (method 1- use EXCEPT)

---

Find students who've taken all classes.

```
SELECT sname
FROM   Student S
WHERE NOT EXISTS
      ((SELECT C.name
        FROM   Class C)
      EXCEPT
      (SELECT E.cname
        FROM   Enrolled E
        WHERE E.snum=S.snum))
```

All classes  
taken by S

# Division in SQL

## (method 1- use EXCEPT)

---

Find students who've taken all classes.

```
SELECT sname  
FROM Student S  
WHERE NOT EXISTS
```

```
    ((SELECT C.name  
       FROM Class C)  
     EXCEPT  
     (SELECT E.cname  
       FROM Enrolled E  
       WHERE E.snum=S.snum))
```

All classes  
that have  
not been  
taken by S

# Division in SQL

## (method 1- use EXCEPT)

---

Find students who've taken all classes.

```
SELECT sname  
FROM Student S  
WHERE NOT EXISTS
```

```
    ((SELECT C.name  
       FROM Class C)  
     EXCEPT  
     (SELECT E.cname  
       FROM Enrolled E  
       WHERE E.snum=S.snum))
```

Only true if  
there is no  
class that has  
not been  
taken by S  
(i.e., S must  
have taken all  
the classes)

# Division in SQL

## (method 2- without using EXCEPT)

Find students who've taken all classes.

```
SELECT sname
FROM   Student S
WHERE NOT EXISTS (SELECT C.name
                  FROM   Class C
                  WHERE  NOT EXISTS (SELECT E.snum
                                    FROM   Enrolled E
                                    WHERE  C.name=E.cname
                                    AND E.snum=S.snum))
```

Returns a result if student  
S is enrolled in class C

# Division in SQL

## (method 2- without using EXCEPT)

Find students who've taken all classes.

```
SELECT sname
FROM   Student S
WHERE NOT EXISTS (SELECT C.name
                  FROM   Class C
                  WHERE  NOT EXISTS (SELECT E.snum
                                    FROM   Enrolled E
                                    WHERE  C.name=E.cname
                                    AND E.snum=S.snum))
```

Only true if student S has  
never been enrolled in  
class C.

# Division in SQL

## (method 2- without using EXCEPT)

Find students who've taken all classes.

```
SELECT sname
FROM   Student S
WHERE NOT EXISTS (SELECT C.name
                  FROM   Class C
                  WHERE  NOT EXISTS (SELECT E.snum
                                    FROM   Enrolled E
                                    WHERE  C.name=E.cname
                                    AND E.snum=S.snum))
```

Find the classes that student  
S has not enrolled in.

# Division in SQL

## (method 2- without using EXCEPT)

Find students who've taken all classes.

```
SELECT sname
FROM   Student S
WHERE  NOT EXISTS (SELECT C.name
                   FROM   Class C
                   WHERE  NOT EXISTS (SELECT E.snum
                                     FROM   Enrolled E
                                     WHERE  C.name=E.cname
                                     AND E.snum=S.snum))
```

Only true if there is no class that student S has never been enrolled in (i.e., student S has been enrolled in all the classes).



# You're Now Leaving the World of Relational Algebra

---

- You now have many ways of asking relational algebra queries
  - For this class, you should be able write queries using all of the different concepts that we've discussed & know the terms used
  - In general, use whatever seems easiest, unless the question specifically asks you to use a specific method.
  - Sometimes the query optimizer may do poorly, and you'll need to try a different version, but we'll ignore that for this class.

# Mind the gap

---

- But there's more you might want to know!
- E.g., “find the average age of students”
- There are extensions of Relational Algebra that cover these topics
  - We won't cover them
- We will cover them in SQL

# Aggregate Operators

---

- These functions operate on the multiset of values of a column of a relation, and return a value

**AVG:** average value

**MIN:** minimum value

**MAX:** maximum value

**SUM:** sum of values

**COUNT:** number of values

- The following versions eliminate duplicates before applying the operation to attribute A:

**COUNT ( DISTINCT A)**

**SUM ( DISTINCT A)**

**AVG ( DISTINCT A)**

```
SELECT count(distinct s.snum)
FROM enrolled e, Student S
WHERE e.snum = s.snum
```

```
SELECT count(s.snum)
FROM enrolled e, Student S
WHERE e.snum = s.snum
```

# Aggregate Operators: Examples

---

# students

```
SELECT COUNT(*)  
FROM Student
```

Find name and age of  
the oldest student(s)

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

Finding average age  
of SR students

```
SELECT AVG (age)  
FROM Student  
WHERE standing='SR'
```

# Aggregation examples

---

- Find the minimum student age

```
SELECT min(age)  
FROM student;
```

- How many students have taken a class with “Database” in the title

```
SELECT count(distinct snum)  
FROM enrolled  
WHERE cname like '%Database%'
```

# GROUP BY and HAVING

---

- Divide tuples into groups and apply aggregate operations to each group.
- Example: *Find the age of the youngest student for each major.*

For $i =$ 'Computer Science',	SELECT MIN (age)
'Civil Engineering' ...	FROM Student
	WHERE major = $i$

## ■ Problem:

We don't know how many majors exist, not to mention this is not good practice

# Grouping Examples

*Find the age of the youngest student who is at least 19, for each major*

```
SELECT    major, MIN(age)
FROM      Student
WHERE     age >= 19
GROUP BY  major
```

Snum	Major	Age
115987938	Computer Science	20
112348546	Computer Science	19
280158572	Animal Science	18
351565322	Accounting	19
556784565	Civil Engineering	21
...	...	...

Major	Age
Computer Science	19
Accounting	19
Civil Engineering	21
...	...

No Animal Science

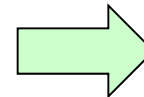
# Grouping Examples with Having

*Find the age of the youngest student who is at least 19, for each major with at least 2 such students*

```
SELECT    major, MIN(age)
FROM      Student
WHERE     age >= 19
GROUP BY  major
HAVING    COUNT(*) > 1
```

Snum	Major	Age
115987938	Computer Science	20
112348546	Computer Science	19
280158572	Animal Science	18
351565322	Accounting	19
556784565	Civil Engineering	21
...	...	...

Major	Age
Computer Science	20
Computer Science	19
Accounting	19
Civil Engineering	21
...	...



Major	
Computer Science	19



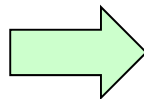
# And there are rules

*Find the age of the youngest student who is at least 19, for each major with at least 2 such students*

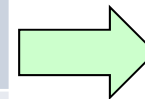
```
SELECT    major, MIN(age)
FROM      Student
WHERE     age >= 19
GROUP BY  major
HAVING    COUNT(*) > 1
```

- Would it make sense if I select age instead of MIN(age)?
- *Would it make sense if I select snum to be returned?*
- *Would it make sense if I select major to be returned?*

Major	Age
Computer Science	20
Computer Science	19
Accounting	19
Civil Engineering	21
...	...



Major	Age
Computer Science	20
Computer Science	19



Major	Age
Computer Science	19

# GROUP BY and HAVING (cont)

SELECT	[DISTINCT] <i>target-list</i>
FROM	<i>relation-list</i>
WHERE	<i>qualification</i>
GROUP BY	<i>grouping-list</i>
HAVING	<i>group-qualification</i>
ORDER BY	<i>target-list</i>

- The *target-list* contains
  - (i) attribute names
  - (ii) terms with aggregate operations (e.g., MIN (S.age)).
- Attributes in (i) must also be in *grouping-list*.
  - each answer tuple corresponds to a *group*,
  - *group* = a set of tuples with same value for all attributes in *grouping-list*
  - selected attributes must have a single value per group.
- Attributes in *group-qualification* are either in *grouping-list* or are arguments to an aggregate operator.

# Conceptual Evaluation of a Query

---

1. compute the cross-product of *relation-list* from
2. keep only tuples that satisfy *qualification* where
3. partition the remaining tuples into groups by the value of attributes in *grouping-list* group by
4. keep only the groups that satisfy *group-qualification* (expressions in *group-qualification* must have a *single value per group!*) having
5. delete fields that are not in *target-list* select
6. generate one answer tuple per qualifying group.

# GROUP BY and HAVING (cont)

---

- Example1: *For each class, find the age of the youngest student who has enrolled in this class:*

```
SELECT    cname, MIN(age)
FROM      Student S, Enrolled E
WHERE     S.snum= E.snum
GROUP BY  cname
```

- Example2: *For each course with more than 1 enrollment, find the age of the youngest student who has taken this class:*

```
SELECT    cname, MIN(age)
FROM      Student S, Enrolled E
WHERE     S.snum = E.snum
GROUP BY  cname
HAVING    COUNT(*) > 1      ← per group qualification!
```

# Groupies of your very own

---

- Find the average age for each standing (e.g., Freshman)
- Find the deptID and # of faculty members for each department having an id > 20

## Grouping Examples (cont')

---

*Find the age of the youngest student with age  $> 18$ , for each major with at least 2 students(of age  $> 18$ )*

## Grouping Examples (cont')

---

Find the age of the youngest student with age  $> 18$ , for each major for which the average age of the students who are  $> 18$  is higher than the average age of all students across all majors.

# Grouping Examples (cont')

---

*Find the age of the youngest student with age > 18, for each major with at least 2 students(of any age)*



# Grouping Examples (cont')

---

*Find those majors for which their average age is the minimum over all majors*

~~SELECT major, avg(age)  
FROM student S  
GROUP BY major  
HAVING min(avg(age))~~

- **WRONG, cannot use nested aggregation**

- One solution would be to use subquery in the FROM Clause

```
SELECT Temp.major, Temp.average  
FROM (SELECT S.major, AVG(S.age) as average  
      FROM Student S  
      GROUP BY S.major) AS Temp
```

A bit ugly

```
WHERE Temp.average in (SELECT MIN(Temp.average) FROM Temp)
```

# Grouping Examples (cont')

---

*Find those majors for which their average age is the minimum over all majors*

~~SELECT major, avg(age)  
FROM student S  
GROUP BY major  
HAVING min(avg(age))~~

- **WRONG, cannot use nested aggregation**

- Another would be to use subquery with ALL in HAVING

```
SELECT major, avg(age)
FROM student S
GROUP BY major
HAVING avg(age) <= all (SELECT AVG(S.age)
                        FROM Student S
                        GROUP BY S.major)
```

Easiest method  
would be to use  
Views

# What are views

---

- Relations that are defined with a create table statement exist in the physical layer
  - do not change unless explicitly told so
- Virtual views do not physically exist, they are defined by expression over the tables.
  - Can be queries (most of the time) as if they were tables.

# Why use views?

---

- Hide some data from users
- Make some queries easier
- Modularity of database
  - When not specified exactly based on tables.

Example: UBC has one table for students. Should the CS Department be able to update CS students info? Yes, Biology students? NO

Create a view for CS to only be able to update CS students

# Defining and using Views

---

- Create View <view name><attributes in view>  
As <view definition>
  - View definition is defined in SQL
  - From now on we can use the view almost as if it is just a normal table
- View  $V(R_1, \dots, R_n)$
- query  $Q$  involving  $V$ 
  - Conceptually
    - $V(R_1, \dots, R_n)$  is used to evaluate  $Q$
  - In reality
    - The evaluation is performed over  $R_1, \dots, R_n$

# Defining and using Views

---

- Example: Suppose tables

Course(Course#,title,dept)

Enrolled(Course#,sid,mark)

```
CREATE VIEW CourseWithFails(dept, course#, mark) AS
SELECT  C.dept, C.course#, mark
FROM    Course C, Enrolled E
WHERE   C.course# = E.course# AND mark<50
```

This view gives the dept, course#, and marks for those courses where someone failed

# Views and Security

---

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
- Given CourseWithFails, but not Course or Enrolled, we can find the course in which some students failed, but we can't find the students who failed.

Course(Course#, title, dept)  
Enrolled(Course#, sid, mark)  
VIEW CourseWithFails(dept, course#, mark)

# View Updates

---

- View updates must occur at the base tables.
  - Ambiguous
  - Difficult

CourseWithFails(dept, course#,  
mark)

Course(Course#, title, dept)  
Enrolled(Course#, sid, mark)

- DBMS's restrict view updates only to some simple views on single tables (called updatable views)



# View Deletes

---

- Drop View <view name>
  - Dropping a view does not affect any tuples of the in the underlying relation.
- How to handle DROP TABLE if there's a view on the table?
- DROP TABLE command has options to prevent a table from being dropped if views are defined on it:
  - DROP TABLE Student RESTRICT
    - drops the table, unless there is a view on it
  - DROP TABLE Student CASCADE
    - drops the table, and recursively drops any view referencing it

# The Beauty of Views

---

*Find those majors for which their average age is the minimum over all majors*

With views:

Create View Temp(major, average) as

```
SELECT      S.major, AVG(S.age) AS average
FROM        Student S
GROUP BY    S.major;
```

```
SELECT major, average
```

```
FROM Temp
```

```
WHERE average = (SELECT MIN(average) FROM Temp)
```

Without views:

```
SELECT Temp.major, Temp.average
```

```
FROM(SELECT S.major, AVG(S.age) as average
```

```
FROM Student S
```

```
GROUP BY S.major) AS Temp
```

```
WHERE Temp.average in (SELECT MIN(Temp.average) FROM Temp)
```

A bit ugly

# Null Values

---

- Tuples may have a null value, denoted by *null*, for some of their attributes
- Value *null* signifies an unknown value or that a value does not exist.
- The predicate **IS NULL** ( **IS NOT NULL** ) can be used to check for null values.
  - E.g. *Find all student names whose age is not known.*  

```
SELECT name  
FROM Student  
WHERE age IS NULL
```
- The result of any arithmetic expression involving *null* is *null*
  - E.g.  $5 + \text{null}$  returns *null*.

# Null Values and Three Valued Logic

- null requires a 3-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*, (*unknown* **or** *false*) = *unknown*  
(*unknown* **or** *unknown*) = *unknown*
  - AND: (*true* **and** *unknown*) = *unknown*, (*false* **and** *unknown*) = *false*,  
(*unknown* **and** *unknown*) = *unknown*
  - NOT: (**not** *unknown*) = *unknown*
  - “*P* is **unknown**” evaluates to true if predicate *P* evaluates to *unknown*
- Any comparison with *null* returns *unknown*
  - E.g. *5 < null* or *null <> null* or *null = null*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*
- All aggregate operations except **count(\*)** ignore tuples with null values on the aggregated attributes.

SELECT count(\*)  
FROM class

SELECT count(fid)  
FROM class

## Previously...

---

```
SELECT DISTINCT cname  
FROM Enrolled e, Student s  
WHERE e.snum = s.snum
```

# Natural Join

---

- The SQL NATURAL JOIN is a type of EQUI JOIN and is structured in such a way that, columns with same name of associate tables will appear once only.
- Natural Join : Guidelines
  - The associated tables have one or more pairs of identically named columns.
  - The columns must be the same data type.
  - Don't use ON clause in a natural join.

```
SELECT *  
FROM student s natural join enrolled e
```

- Natural join of tables with no pairs of identically named columns will return the cross product of the two tables.

```
SELECT *  
FROM student s natural join class c
```

# More fun with joins examples

R		S	
A	B	B	C
1	2	2	4
3	3	4	6

Natural  
Inner Join

A	B	C
1	2	4

Natural  
Left outer Join

A	B	C
1	2	4
3	3	Null

Natural  
Right outer Join

A	B	C
1	2	4
Null	4	6

Natural  
outer Join

A	B	C
1	2	4
3	3	Null
Null	4	6

Outer join (without the Natural) will use the key word ON for specifying The condition of the join.

Outer join not implemented in MYSQL  
Outer join is implemented in Oracle

# More fun with joins

---

- What happens if I execute query:  
SELECT \*  
FROM student s, enrolled e  
WHERE s.snum = e.snum
- To get *all* students, you need an *outer join*
- There are several special joins declared in the *FROM* clause:
  - Inner join – default: only include matches
  - Left outer join – include all tuples from left hand relation
  - Right outer join – include all tuples from right hand relation
  - Full outer join – include all tuples from both relations
- Orthogonal: can have natural join (as in relational algebra)

Example: SELECT \*

FROM Student S NATURAL LEFT OUTER JOIN Enrolled E



# Clicker outer join question

- Given:  
Compute:  
SELECT R.A, R.B, S.B, S.C, S.D  
FROM R FULL OUTER JOIN S  
ON (R.A > S.B AND R.B = S.C)

R(A,B)		S(B,C,D)		
A	B	B	C	D
1	2	2	4	6
3	4	4	6	8
5	6	4	7	9

- Which of the following tuples of R or S is dangling (and therefore needs to be padded in the outer join)?
  - A. (1,2) of R
  - B. (3,4) of R
  - C. (2,4,6) of S
  - D. All of the above
  - E. None of the above

# Database Manipulation

## Insertion redux

---

- Can insert a single tuple using:

```
INSERT INTO Student  
VALUES (53688, 'Smith', '222 W.15th ave', 333-4444, MATH)
```

or

```
INSERT INTO Student (sid, name, address, phone, major)  
VALUES (53688, 'Smith', '222 W.15th ave', 333-4444, MATH)
```

- Add a tuple to student with null address and phone:

```
INSERT INTO Student (sid, name, address, phone, major)  
VALUES (33388, 'Chan', null, null, CPSC)
```

# Database Manipulation

## Insertion redux (cont)

---

- Can add values selected from another table
- Enroll student 51135593 into every class taught by faculty 90873519

```
INSERT INTO Enrolled  
SELECT 51135593, name  
FROM Class  
WHERE fid = 90873519
```

The SELECT-FROM-WHERE statement is fully evaluated before any of its results are inserted or deleted.

# Database Manipulation

## Deletion

---

- Note that only whole tuples are deleted.
- Can delete all tuples satisfying some condition (e.g., name = Smith):

```
DELETE FROM Student  
WHERE name = 'Smith'
```

- The WHERE clause can contain nested queries

# Database Manipulation

## Updates

---

- Increase the age of all students by 2 (should not be more than 100)
- Need to write two updates:

```
UPDATE Student
SET      age = 100
WHERE    age >= 98
```

```
UPDATE Student
SET age = age + 2
WHERE age < 98
```

- Is the order important?

A: Yes

B: No

# Integrity Constraints (Review)

---

- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are disallowed.
  - Can ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be  $< 200$ )
- Types of IC's:
  - domain constraints,
  - primary key constraints,
  - foreign key constraints,
  - general constraints

# General Constraints: Check

---

- We can specify constraints over a single table using table constraints, which have the form

## Check conditional-expression

```
CREATE TABLE Student
( snum INTEGER,
  sname CHAR(32),
  major CHAR(32),
  standing CHAR(2)
  age REAL,
  PRIMARY KEY (snum),
  CHECK ( age >= 10
        AND age < 100 );
```

Check constraints are checked when tuples are inserted or modified

# General Constraints: Check

---

- Constraints can be named
- Can use subqueries to express constraint
- Table constraints are associated with a single table, although the conditional expression in the check clause can refer to other tables

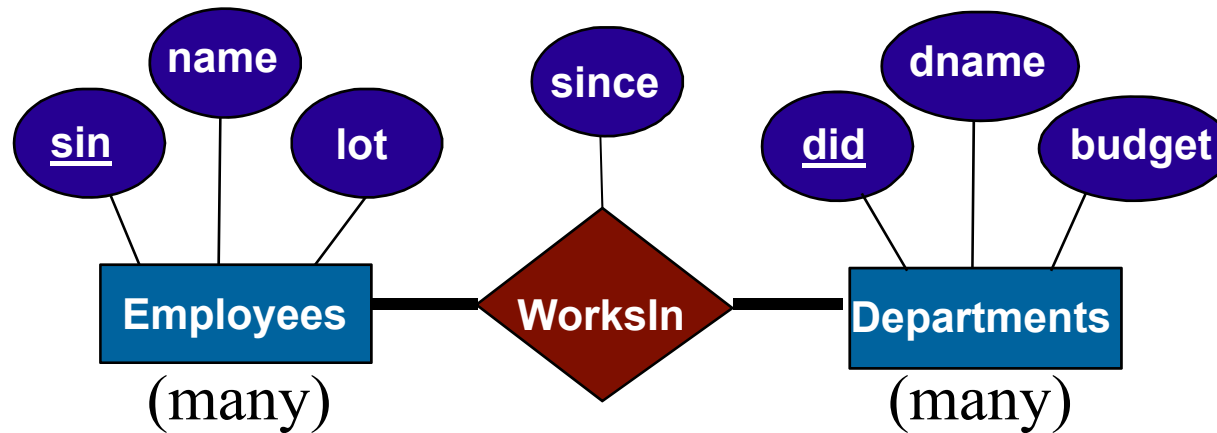
```
CREATE TABLE Enrolled
( snum  INTEGER,
  cname  CHAR(32),
  PRIMARY KEY (snum, cname),
  CONSTRAINT noR15
  CHECK (`R15' <>
        ( SELECT c.room
          FROM   class c
          WHERE  c.name=cname)));
```

No one can be  
enrolled in a class,  
which is held in R15



# Constraints over Multiple Relations: Remember this one?

---



- We couldn't express  
“every employee works in a department and every department has some employee in it”?
- Neither foreign-key nor not-null constraints in **Works\_In** can do that.
- Assertions to the rescue!

# Constraints Over Multiple Relations

---

- Cannot be defined in one table.
- Are defined as ASSERTIONS which are not associated with any table
- Example: *Every MovieStar needs to star in at least one Movie*

```
CREATE ASSERTION totalEmployment
CHECK
( NOT EXISTS ((SELECT StarID FROM MovieStar)
              EXCEPT
              (SELECT StarID FROM StarsIn))));
```

# Constraints Over Multiple Relations

---

- Example: Write an assertion to enforce every student to be registered in at least one course.

```
CREATE ASSERTION Checkregistry  
CHECK  
( NOT EXISTS ((SELECT snum FROM student)  
               EXCEPT  
               (SELECT snum FROM enrolled))));
```

# Triggers

---

- Trigger : a procedure that starts automatically if specified changes occur to the DBMS
- Active Database: a database with triggers
- A trigger has three parts:
  1. Event (activates the trigger)
  2. Condition (tests whether the trigger should run)
  3. Action (procedure executed when trigger runs)
- Database vendors did not wait for trigger standards! So trigger format depends on the DBMS
- **NOTE: triggers may cause cascading effects.**  
**Good way to shoot yourself in the foot**

Useful for project  
Not tested on exams

# That's nice. But how do we code with SQL?

---

- Direct SQL is rarely used: usually, SQL is embedded in some application code.
- We need some method to reference SQL statements.
- But: there is an *impedance mismatch* problem.
  - Structures in databases  $\leftrightarrow$  structures in programming languages
- Many things can be explained with the impedance mismatch.

# The Impedance Mismatch Problem

---

The host language manipulates variables, values, pointers SQL manipulates relations.

There is no construct in the host language for manipulating relations. See

[https://en.wikipedia.org/wiki/Object-relational\\_impedance\\_mismatch](https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch)

## Why not use only one language?

- Forgetting SQL: “we can quickly dispense with this idea” [Ullman & Widom, pg. 363].
- SQL cannot do everything that the host language can do.

# Database APIs

---

Rather than modify compiler, add library with database calls (API)

- Special standardized interface: procedures/objects
- Passes SQL strings from language, presents result sets in a language-friendly way – solves that impedance mismatch
- Microsoft's *ODBC* is a C/C++ standard on Windows
- Sun's *JDBC* a Java equivalent
- API's are DBMS-neutral
  - a “driver” traps the calls and translates them into DBMS-specific code

# A glimpse into your possible future:

## JDBC

---

- JDBC supports a variety of features for querying and updating data, and for retrieving query results
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with the database:
  - Open a connection
  - Create a “statement” object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors



# SQL API in Java (JDBC)

---

```
Connection con = // connect
    DriverManager.getConnection(url, "login", "pass");
Statement stmt = con.createStatement(); // set up stmt
String query = "SELECT sname, age FROM Student";
ResultSet rs = stmt.executeQuery(query);
try { // handle exceptions
    // loop through result tuples
    while (rs.next()) {
        String s = rs.getString("sname");
        Int n = rs.getFloat("age");
        System.out.println(s + "    " + n);
    }
} catch(SQLException ex) {
    System.out.println(ex.getMessage ()
        + ex.getSQLState () + ex.getErrorCode ());
}
```

# Summary

---

- SQL was an important factor in the early acceptance of the relational model; more natural than earlier, procedural query languages.
- Relationally complete; in fact, significantly more expressive power than relational algebra.
- Consists of a data definition, data manipulation and query language.
- Many alternative ways to write a query; optimizer should look for most efficient evaluation plan.
  - In practice, users need to be aware of how queries are optimized and evaluated for best results.

# Summary (Cont')

---

- NULL for unknown field values brings many complications
- SQL allows specification of rich integrity constraints (and triggers)
- Embedded SQL allows execution within a host language; cursor mechanism allows retrieval of one record at a time
- APIs such as ODBC and JDBC introduce a layer of abstraction between application and DBMS

# Learning Goals Revisited

---

- Given the schemas of a relation, create SQL queries using: SELECT, FROM, WHERE, EXISTS, NOT EXISTS, UNIQUE, NOT UNIQUE, ANY, ALL, DISTINCT, GROUP BY and HAVING.
- Show that there are alternative ways of coding SQL queries to yield the same result. Determine whether or not two SQL queries are equivalent.
- Given a SQL query and table schemas and instances, compute the query result.
- Translate a query between SQL and RA.
- Comment on the relative expressive power of SQL and RA.
- Explain the purpose of NULL values and justify their use. Also describe the difficulties added by having nulls.
- Create and modify table schemas and views in SQL.
- Explain the role and advantages of embedding SQL in application programs.
- Write SQL for a small-to-medium sized programming application that requires database access.
- Identify the pros and cons of using general table constraints (e.g., CONSTRAINT, CHECK) and triggers in databases.