SQL Part 1

The plan for today

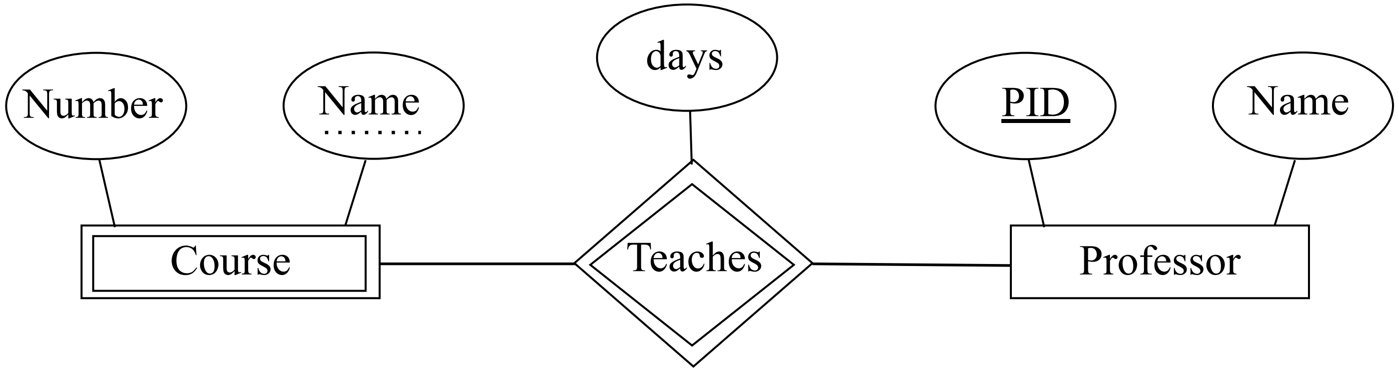
- Translating ER to Relational Tables
- SQL Basics
  - UNION, INTERSECT, EXCEPT
  - Nested Queries
  - ANY, ALL operators
  - Aggregate Operators
  - Some SQL Examples
- Advanced SQL next time

Translating ER diagrams into Relations

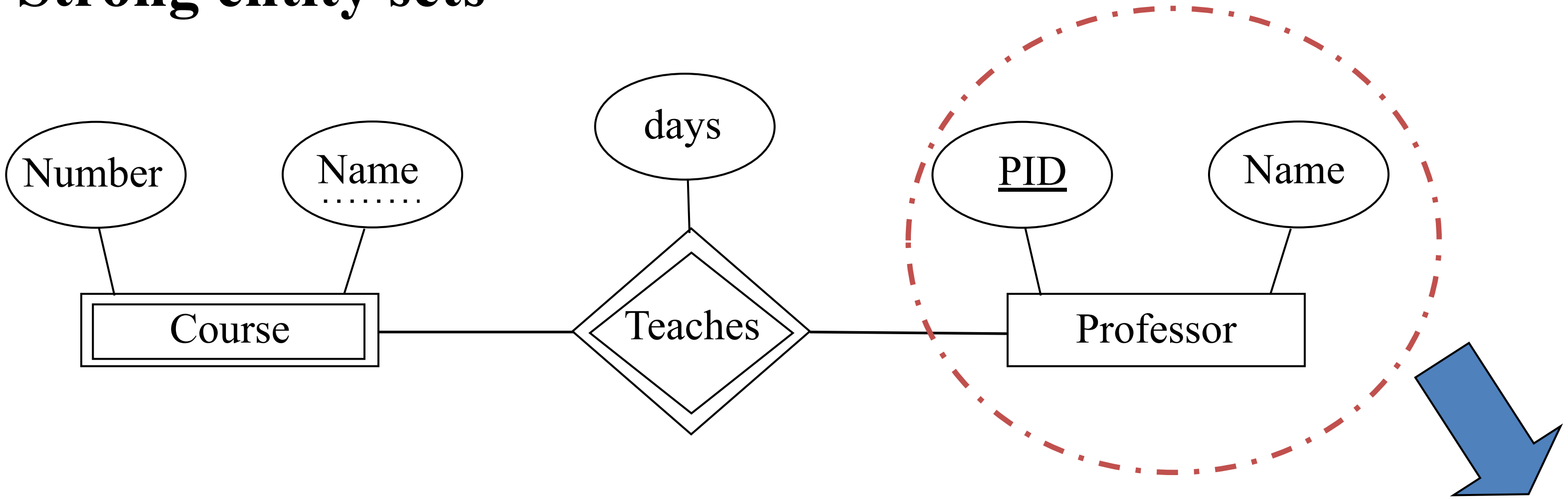
We need to figure out how to translate ER diagrams into relations.

There are only three cases to worry about.

- Strong entity sets
- Weak entity sets
- Relationship sets



Strong entity sets

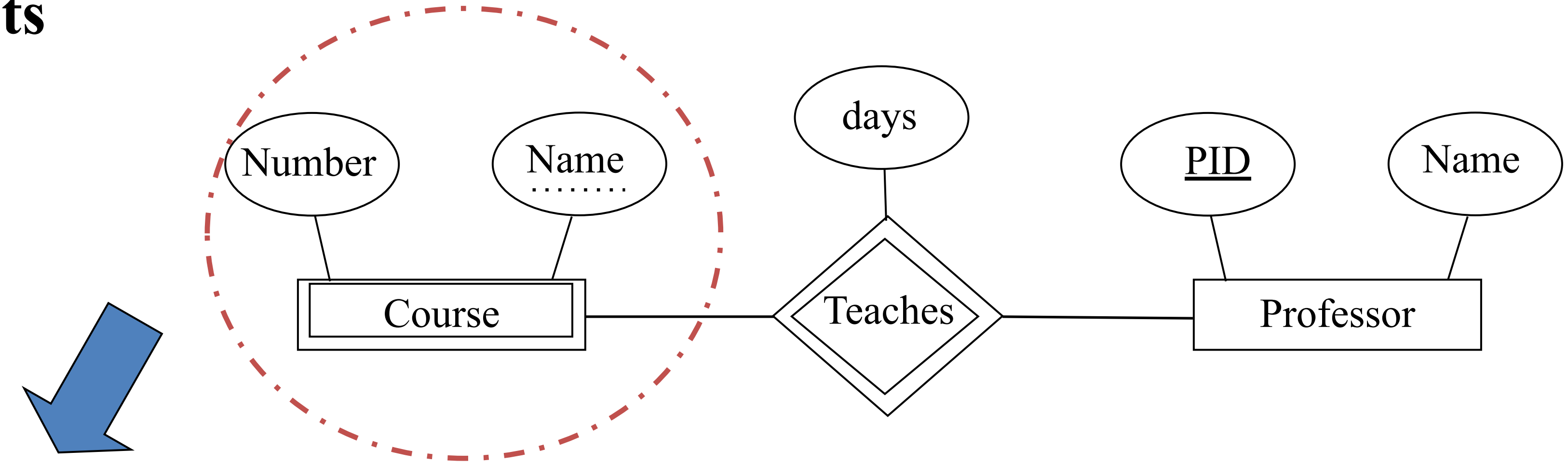


professor(*PID* : string, *name* : string)

PID	name
1234	John
3421	Daisy
2342	Barbara
4531	Audrey

This is trivial, the primary key of the ER diagram becomes the primary key of the relation. All other fields are copied in (in any order)

Weak entity sets



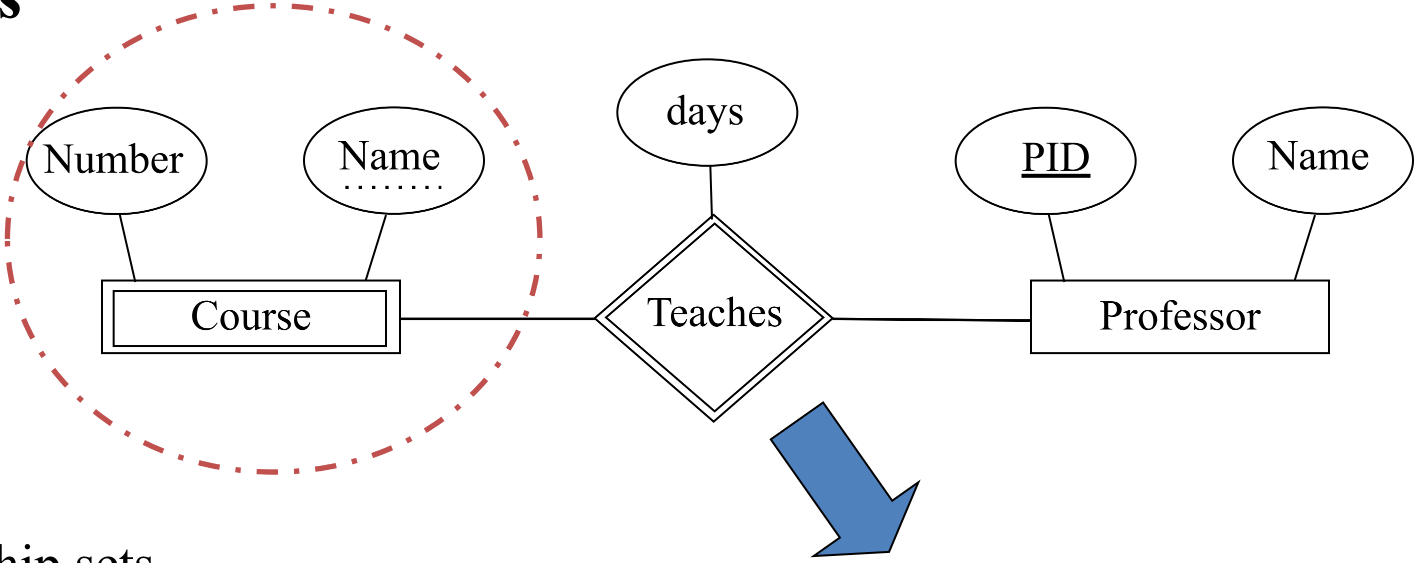
course(*PID* : string, *number* : string, *name* : string)

PID	number	name
1234	CS12	C++
3421	CS11	Java
2342	CS12	C++
4531	CS15	LISP

The primary key of the relation consists of the union of the primary key of the strong entity set and the discriminator of the weak entity set. The "imported" key from the strong entity set is called the **foreign key**.

All other fields are copied in (in any order)

Relationship entity sets



teaches(<u>*PID*</u> : string, *days* : string)

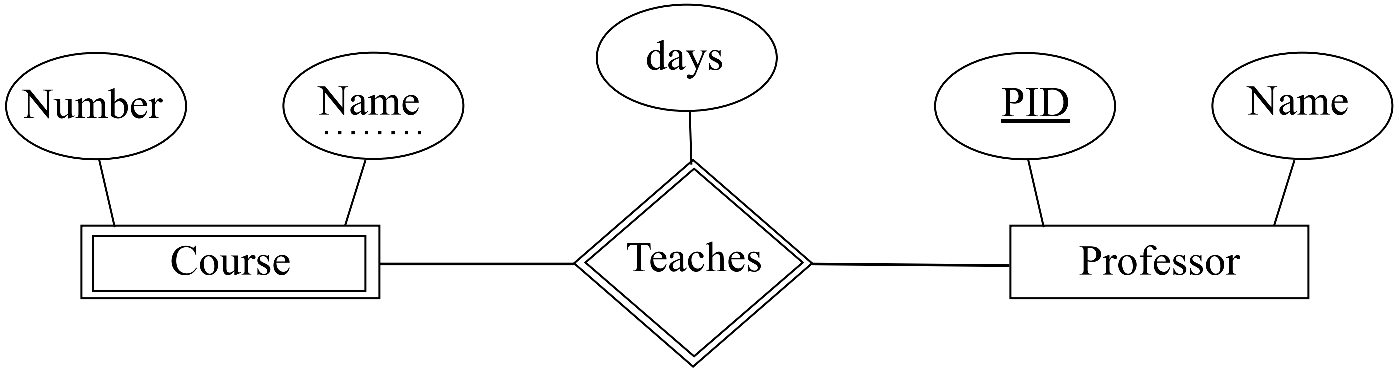
For one-to-one relationship sets, the relation's primary key can be that of either entity set.

For many-to-many relationship sets, the union of the primary keys becomes the relation's primary key

For other cases, the the relation's primary key is taken from the strong entity set.

<u>**PID**</u>	**days**
1234	mwf
3421	wed
2342	tue
4531	sat

So, this ER Model…



… maps to this **database schema**

professor(<u>*PID*</u> : string, *name* : string)
course(<u>*PID*</u> : string, *number* : string, <u>*name*</u> : string)
teaches(<u>*PID*</u> : string, *days* : string)

We have seen how to create a database schema, but how do we create an actual database on our computers?

We use SQL, a language that allows us to build, modify and query databases.

SQL Introduction

- SQL is a standard language for querying and manipulating data

- SQL is a **very high-level** programming language
  - This works because it is optimized well!

- Many standards out there:
  - ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ….
  - Vendors support various subsets

SQL stands for
Structured **Q**uery **L**anguage

Probably the world’s most successful **parallel** programming language (multicore?)

SQL Motivation

- Dark times 7 years ago.
  - Are databases dead?

- Now, as before: everyone sells SQL
  - Pig, Hive, Impala

SQL is a…

- Data Definition Language (DDL)
  - Define relational *schemata*
  - Create/alter/delete tables and their attributes

- Data Manipulation Language (DML)
  - Insert/delete/modify tuples in tables
  - Query one or more tables

Creating Relations in SQL

- Creates the Students relation.
- Observe that the type (domain) of each field is specified and enforced by the DBMS whenever tuples are added or modified.
- As another example, the Enrolled table holds information about courses that students take.

```
CREATE TABLE Students
(sid CHAR(20), name CHAR(20),
login CHAR(10), age INTEGER, gpa REAL)
```

```
CREATE TABLE Enrolled
    (sid CHAR(20),  cid CHAR(20),
     grade CHAR(2))
```

Adding and Deleting Tuples

- Can insert a single tuple using:

```
INSERT INTO  Students (sid, name, login, age, gpa)
VALUES  (53688, ‘Smith’, ‘smith@ee’, 18, 3.2)
```

- Can delete all tuples satisfying some condition (e.g., name = Smith):

```
DELETE
FROM Students
WHERE name = ‘Smith’
```

Destroying and Altering Relations

```
DROP TABLE  Students
```

- Destroys the relation Students. The schema information *and* the tuples are deleted.

```
ALTER TABLE  Students
      ADD COLUMN firstYear integer
```

- The schema of Students is altered by adding a new field; every tuple in the current instance is extended with a *null* value in the new field.

Integrity Constraints (ICs)

- IC: condition that must be true for *any* instance of the database; e.g., *domain constraints.*
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.

- A *legal* instance of a relation is one that satisfies all specified ICs.
  - DBMS should not allow illegal instances.

- If the DBMS checks ICs, stored data is more faithful to real-world meaning.
  - Avoids data entry errors, too!

Primary and Candidate Keys in SQL

- Possibly many *candidate keys* (specified using UNIQUE), one of which is chosen as the *primary key*.

- "For a given student and course, there is a single grade." vs. "Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade."

- Used carelessly, an IC can prevent the storage of database instances that arise in practice!

```
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid,cid) )
```

```
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid),
   UNIQUE (cid, grade) )
```

Foreign Keys, Referential Integrity

- Foreign key : Set of fields in one relation that is used to `refer’ to a tuple in another relation. (Must correspond to primary key of the second relation.) Like a `logical pointer’.

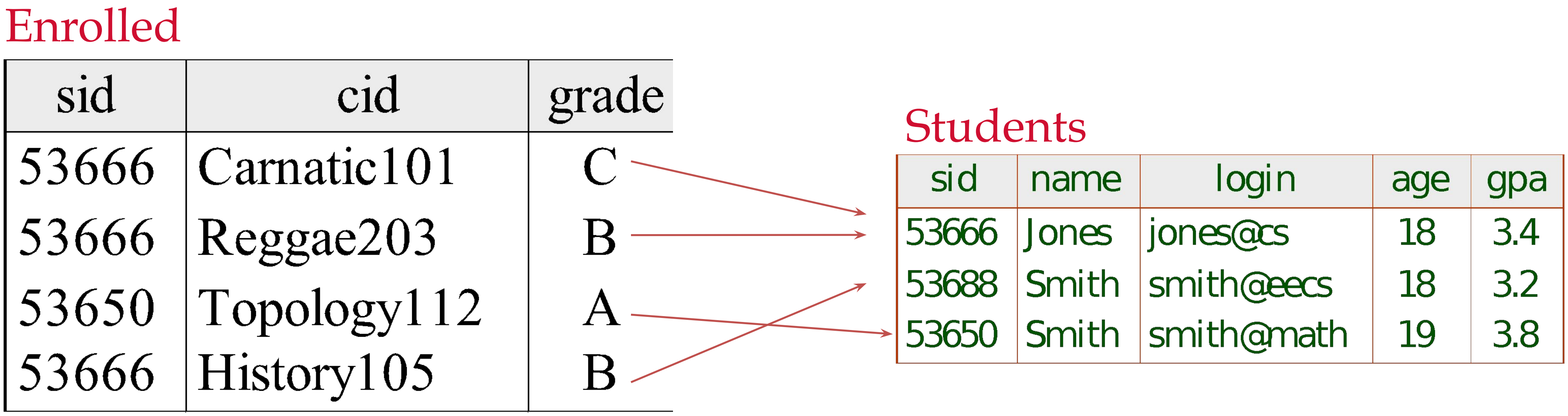
- e.g. sid is a foreign key referring to Students:
  - Enrolled(sid: string, cid: string, grade: string)

  - If all foreign key constraints are enforced, referential integrity is achieved, i.e., no dangling references.

Foreign Keys in SQL

- Only students listed in the Students relation should be allowed to enroll for courses.

```
CREATE TABLE Enrolled
    (sid CHAR(20),  cid CHAR(20),  grade CHAR(2),
    PRIMARY KEY  (sid,cid),
    FOREIGN KEY (sid) REFERENCES Students )
```



Enrolled

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B

Students

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

Enforcing Referential Integrity

- Consider Students and Enrolled; *sid* in Enrolled is a foreign key that references Students.
- What should be done if an Enrolled tuple with a non-existent student id is inserted? (*Reject it!*)
- What should be done if a Students tuple is deleted?
  - Also delete all Enrolled tuples that refer to it.
  - Disallow deletion of a Students tuple that is referred to.
  - Set sid in Enrolled tuples that refer to it to a *default sid*.
  - (In SQL, also: Set sid in Enrolled tuples that refer to it to a special value *null*, denoting *`unknown'* or *`inapplicable'*.)
- Similar if primary key of Students tuple is updated.

Basic form of SQL Queries

```
SELECT      target-list
FROM        relation-list
WHERE       qualification
```

- *target-list* A list of attributes of output relations in *relation-list*
- *relation-list* A list of relation names (possibly with a *range-variable* after each name)

  e.g. Sailors S, Reserves R
- *qualification* Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of <, >, ≤, ≥, =, ≠) combined using AND, OR and NOT.

What’s contained in an SQL Query?

```
SELECT     target-list
FROM       relation-list
WHERE      qualification
```

Every SQL Query must have:

- SELECT clause: specifies columns to be retained in result
- FROM clause: specifies a cross-product of tables

The WHERE clause (optional) specifies selection conditions on the tables mentioned in the FROM clause

Table Definitions

We will be using the following relations in our examples:

Sailors(sid:integer, sname:string, rating:integer, age:real)

Boats(bid:integer, bname:string, color:string)

Reserves(sid:integer, bid:integer, day:date)

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

A Simple SQL Query

Find the names and ages of all sailors

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

sname	*age*
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Horatio	35.0
Art	25.5
Bob	63.5

```
SELECT S.sname, S.age
FROM Sailors S
```

Duplicate Results

Preventing Duplicate Tuples in the Result

- Use the DISTINCT keyword in the SELECT clause:

```
SELECT DISTINCT S.sname, S.age
FROM Sailors S
```

Results of Original Query without Duplicates

sname	***age***
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Horatio	35.0
Art	25.5
Bob	63.5

Appears only once

Example SQL Query…1

Sailors ($\underline{sid}$, sname, rating, age)
Reserves ($\underline{sid}$, bid, day)
Boats ($\underline{bid}$, bname, color)

Find the names of sailors who have reserved boat 103

Relational Algebra:

$\pi_{sname}\ ((\sigma_{bid=103} \mathit{Reserves}) \bowtie \mathit{Sailors})$

SQL:

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

Result of Previous Query

sid	*bid*	*day*
22	103	10/08/04
31	103	11/06/04
74	103	09/08/04

⋈

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

Result:

sname
Dustin
Lubber
Horatio

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 R.bid=103
```

OR

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

However, it is a good style to always use range variables!

Example SQL Query…2

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

*Find the **sids** of sailors who have reserved a red boat*

```
SELECT R.sid
FROM Boats B, Reserves R
WHERE B.bid=R.bid AND B.color=‘red’
```

Example SQL Query…3

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

Find the ***names*** *of sailors who have reserved a red boat*

```
SELECT S.sname

FROM Sailors S, Boats B, Reserves R

WHERE S.sid=R.sid AND B.bid=R.bid AND B.color=‘red’
```

Example SQL Query…4

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

Find the ***colors*** *of boats reserved by ‘Lubber’*

```
SELECT B.color

FROM Sailors S, Reserves R, Boats B

WHERE S.sid=R.sid AND R.bid=B.bid AND  S.sname=‘Lubber’
```

Expressions and Strings

- AS and = are two ways to name fields in result.
- LIKE is used for string matching. ‘_’ stands for exactly one arbitrary character and ‘%’ stands for 0 or more arbitrary characters.

Expressions and Strings Example

Find triples (of ages of sailors and two fields defined by expressions, i.e. current age-1 and twice the current age) for sailors whose names begin and end with B and contain at least three characters.

```
SELECT  S.age, age1=S.age-1, 2*S.age AS age2
FROM  Sailors S
WHERE  S.sname LIKE ‘B_%B’
```

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

Result:

age	*age1*	*Age2*
63.5	62.5	127.0

UNION, INTERSECT, EXCEPT

- UNION: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).
- EXCEPT: Can be used to compute the set-difference operation on two *union-compatible* sets of tuples.
- INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples.

Illustration of UNION…1

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

Find the sids of sailors who have reserved a red ***or*** *a green boat*

Intuitively, we would write:

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
                AND (B.color='red' OR B.color='green')
```

Illustration of UNION…2

We can also do this using a UNION keyword:

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
          AND B.color=‘red’

UNION

SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
          AND B.color=‘green’
```

Find the sids of sailors who have reserved a red ***or*** *a green boat*

Illustration of INTERSECT…1

Sailors (<u>sid</u>, sname, rating, age)
Reserves (<u>sid</u>, bid, day)
Boats (<u>bid</u>, bname, color)

*Find names of sailors who've reserved a red **and** a green boat*

Intuitively, we would write:

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
                    AND (B.color='red' AND B.color='green')
```

Is this correct???

Illustration of INTERSECT…2

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

*Find names of sailors who've reserved a red **and** a green boat*

Intuitively, we would write the SQL query as:

```
SELECT  S.sname
FROM     Sailors S, Boats B1, Reserves R1, Boats B2,  Reserves R2
WHERE  S.sid=R1.sid AND
        R1.bid=B1.bid AND
        S.sid=R2.sid AND
        R2.bid=B2.bid  AND
        (B1.color='red' AND B2.color='green')
```

Illustration of INTERSECT…2

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

We can also do this using a INTERSECT keyword:

```
SELECT  S.sname
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color=‘red’

INTERSECT

SELECT  S.sname
FROM  Sailors S2, Boats B2, Reserves R2
WHERE  S2.sid=R2.sid AND R2.bid=B2.bid AND B2.color=‘green’
```

(Is this correct??)

Correct SQL Query for the Previous Example

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
              AND B.color=‘red’
INTERSECT
SELECT  S2.sid
FROM  Sailors S2, Boats B2, Reserves R2
WHERE  S2.sid=R2.sid AND R2.bid=B2.bid
              AND B2.color=‘green’
```

(This time we have actually extracted the ***sids*** of sailors, and not their names.)

Illustration of EXCEPT

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

Find the sids of all sailors who have reserved red boats ***but not*** *green boats:*

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color=‘red’

EXCEPT

SELECT  S2.sid
FROM  Sailors S2, Boats B2, Reserves R2
WHERE  S2.sid=R2.sid AND R2.bid=B2.bid AND B2.color=‘green’
```

Nested Queries

- A **nested** query is a query that has another query embedded within it; this embedded query is called the **subquery**.

- Subqueries generally occur within the WHERE clause (but can also appear within the FROM and HAVING clauses)

- Nested queries are a very powerful feature of SQL. They help us write short and efficient queries.

(Think of nested **for** loops in C++. Nested queries in SQL are similar)

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

Example of a Nested Query

Find names of sailors who have reserved boat 103

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.sid IN  ( SELECT  R.sid
                   FROM  Reserves R
                   WHERE  R.bid=103 )
```

Another Example of a Nested Query

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

*Find names of sailors who **have not** reserved boat 103*

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.sid NOT IN  ( SELECT  R.sid
                        FROM  Reserves R
                        WHERE  R.bid=103 )
```

Correlated Nested Queries…1

- Thus far, we have seen nested queries where the inner subquery is independent of the outer query.

- We can make the inner subquery **depend** on the outer query. This is called correlation.

Correlated Nested Queries…2

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

Find names of sailors who have reserved boat 103

```
SELECT  S.sname
FROM  Sailors S
WHERE  EXISTS  (SELECT  *
                FROM  Reserves R
                WHERE  R.bid=103 AND R.sid=S.sid)
```

Tests whether the set is nonempty

(For finding sailors who have **not** reserved boat 103, we would use NOT EXISTS)

UNIQUE operator

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

- When we apply UNIQUE to a subquery, it returns **true** if no row is duplicated in the answer to the subquery.
- What would the following SQL query return?

```
SELECT  S.sname
FROM  Sailors S
WHERE  UNIQUE  ( SELECT  R.bid
                 FROM  Reserves R
                       WHERE  R.bid=103 AND R.sid=S.sid)
```

(All sailors with at most one reservation for boat 103.)

ANY and ALL operators

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

Find sailors whose rating is better than some sailor named Horatio

```
SELECT S.sid
FROM Sailors S
WHERE S.rating > ANY (`  SELECT S2.rating
                    FROM Sailors S2
                    WHERE S2.sname=‘Horatio’)
```

(Can you find the probable bug in this SQL query??) Hint: what if there’re several sailors named Horatio?

Using ALL operator

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

*Find sailors whose rating is better than **every** sailor named Horatio*

```
SELECT S.sid
FROM Sailors S
WHERE S.rating > ALL( SELECT S2.rating
                      FROM Sailors S2
                   WHERE S2.sname=‘Horatio’)
```

Aggregate operators

- What is aggregation?
  - Computing arithmetic expressions, such as **Minimum** or **Maximum**

- The aggregate operators supported by SQL are:
  COUNT, SUM, AVG, MIN, MAX

Aggregate Operators

- **COUNT**(A): The number of values in the column A
- **SUM**(A): The sum of all values in column A
- **AVG**(A): The average of all values in column A
- **MAX**(A): The maximum value in column A
- **MIN**(A): The minimum value in column A

(We can use DISTINCT with COUNT, SUM and AVG to compute only over non-duplicated columns)

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

Using the COUNT operator

Count the number of sailors

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

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

Example of SUM operator

Find the sum of ages of all sailors with a rating of 10

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

Example of AVG operator

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

Find the average age of all sailors with rating 10

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

(Shouldn't we use DISTINCT in this case to take care of duplicated sailor ages??)

Example of MAX operator

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

Find the name and age of the oldest sailor

```
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 )
```

But this is illegal in SQL!!

BETWEEN and **AND** Example

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

- The **BETWEEN** and **AND** operator selects a range of data between two values.
- These values can be numbers, text, or dates.

Find the names of sailors whose age is between 25 and 35

```
SELECT sname
FROM Sailors
WHERE age BETWEEN 25 AND 35
```

SQL Examples…

Sailors (<u>sid</u>, sname, rating, age)
Reserves (<u>sid</u>, bid, day)
Boats (<u>bid</u>, bname, color)

```
SELECT *
FROM Sailors
WHERE sname NOT BETWEEN  'Hansen' AND 'Pettersen'
```

*Finds all sailors whose name is **not** (alphabetically) between Hansen and Pettersen*

SQL Examples…

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

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

```
SELECT *
FROM Sailors
WHERE sname LIKE ‘A%’
```

sid	*sname*	*rating*	*age*
32	Andy	8	25.5
85	Art	3	25.5

Finds all sailors whose name begins with ‘A’ and is at least 1-character long

SQL Examples…

Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)

```
SELECT SUM(age)
FROM Sailors
WHERE age>20
```

Finds the sum of ages of all sailors whose age is greater than 20

```
SELECT MIN(age)
FROM Sailors
WHERE age>20
```

Finds the minimum age from the ages of all sailors whose age is greater than 20

Select Statements Revisited

- select clause ----------- attribute selection part
- from clause ----------- relation selection part
- where clause ----------- join, selection conditions part
- group by clause ----------- partition part
- having clause ----------- partition filtering part
- order by clause ----------- ordering rows part

+

Aggregates, set operations, subqueries

Subquery predicates revisited ---- EXISTS

- The *EXISTS* predicate is TRUE if and only if the Subquery returns a non-empty set.

- The *NOT EXISTS* predicate is TRUE if and only if the Subquery returns an empty set.

- The *NOT EXISTS* can be used to implement the MINUS operator from relational algebra.

Banking Example

- branch (branch-name, branch-city, assets)
- customer (customer-name, customer-street, customer-only)
- account (account-number, branch-name, balance)
- loan (loan-number, branch-name, amount)
- depositor (customer-name, account-number)
- borrower (customer-name, loan-number)

Groups of Rows in SQL

- SQL allows Select statements to provide a kind of natural ‘report’ function, grouping the rows on the basis of commonality of values and performing set functions on the rows grouped:
- *SELECT branch_name, SUM(balance) FROM account GROUP BY branch_name*.
- The GROUP BY clause of the Select statement will result in the set of rows being generated as if the following loop-controlled query were being performed:

```
FOR EACH DISTINCT VALUE v OF branch_name IN account
     SELECT branch_name, SUM(balance) FROM account
    WHERE branch_name=v
END FOR
```

Groups of Rows in SQL

- A set of functions occurring in the SELECT list aggregates for the set of rows in each group and thus creates a single value for each group.
- It is important that all of the attributes named in the select list have a single atomic value, for each group of common GROUP BY values:

```
SELECT account-number, branch-name, SUM(balance)
GROUP BY account-number  ----------- INVALID
```

Groups of Rows in SQL –Example 1

“Find the total amount of money owed by each depositor”

```
SELECT c.customer-name, SUM(balance)
FROM account S, customer C, depositor D
WHERE S.account-number = D.account-number and
          C.customer-name = D.customer-name
GROUP BY customer-name
```

Filter grouping

- To eliminate rows from the result of a select statement where a *GROUP BY* clause appears we use the *HAVING* clause, which is evaluated after the *GROUP BY*.
- For example, the query:

  SELECT account-branch, SUM(balance) FROM account GROUP BY account-branch HAVING SUM(balance)>1000.

  will print the account branches and total balances for every branch where the total account balance exceeds 1000.
- The *HAVING* clause can only apply tests to values that are single-valued for groups in the SELECT statement.
- The *HAVING* clause can have a nested subquery, just like the *WHERE* clause

Filter Grouping –Example 1

“Find the total amount of money owed by each depositor, for each depositor that own at least 2 accounts”

```
SELECT C.customer-name, SUM(balance)
FROM account S, customer C, depositor D
WHERE S.account-number = D.account-number and
           C.customer-name = D.customer-name
GROUP BY customer-name
HAVING COUNT(*) > 1
```

Order Results

- We use the *ORDER BY* clause when we want the output to be presented in a particular order.

- We provide the list of attributes to order on.

Order –Example 1

“Find the total amount of money owed by each depositor, for each depositor that own at least 2 accounts, present the results in descending order of total balance”

```
SELECT C.customer-name, SUM(balance) AS sbalance
From account S, customer C, depositor D
WHERE S.account-number = D.account-number and
          C.customer-name = D.customer-name
GROUP BY customer-name
HAVING COUNT(*) > 1
ORDER BY Desc sbalance
```

Null Values

- We use *null* when the column value is either *unknown* or *inapplicable*.
- A comparison with at least one null value always returns *unknown*.
- SQL also provides a special comparison operator *IS NULL* to test whether a column value is *null*.
- To incorporate nulls in the definition of duplicates we define that two rows are duplicates if corresponding rows are equal or both contain *null*.

Joins – Covered next time

- Let R and S be two tables. The outer join preserves the rows of R and S that have no matching rows according to the join condition and outputs them with nulls at the non-applicable columns.

- There exist three different variants: *left outer join*, *right outer join* and *full outer join*.

Conceptual order in query evaluation

- First the relational products of the tables in the *FROM* clause are evaluated.
- From this, rows not satisfying the *WHERE* clause are eliminated.
- The remaining rows are grouped in accordance with the *GROUP BY* clause.
- Groups not satisfying the *HAVING* clause are then eliminated.
- The expressions in the *SELECT* list are evaluated.
- If the keyword *DISTINCT* is present, duplicate rows are now eliminated.
- Evaluate *UNION, INTERSECT* and *EXCEPT* for Subqueries up to this point.
- Finally, the set of all selected rows is sorted if the *ORDER BY* is present.