Week 4 - SQL: The Query Language

"Life is just a bowl of queries."

-Anon

Lecturer: Dr. Tony Stockman

Room CS405

Lecture Outline

- Part 1: Data definition language
- Part 2: Data manipulation language
  - Queries
  - Insert, delete and update

Relational Query Languages

- A major strength of the relational model: supports simple, powerful querying of data
- Two sublanguages:
- DDL – Data Definition Language
  - Define and modify schema (at all 3 levels)
- DML – Data Manipulation Language
  - Queries can be written intuitively
- The DBMS is responsible for efficient evaluation.
  - The key: precise semantics for relational queries
  - Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change
  - Internal cost model drives use of indexes and choice of access paths and physical operators

The SQL Query Language

- The most widely used relational query language
- Originally IBM, then ANSI in 1986
- Current standard is SQL-2008
  - 2003 was last major update: XML, window functions, sequences, auto-generated IDs
  - Not fully supported yet
- SQL-1999 Introduced "Object-Relational" concepts
  - Also not fully supported yet
- SQL92 is a basic subset
  - Most systems support at least this
- PostgreSQL has some "unique" aspects (as do most systems)
- SQL is not synonymous with Microsoft's "SQL Server"

Part 1: SQL Data Definition (DDL)

- Objects
  - Table
- Commands
  - CREATE
  - ALTER
  - DROP

Tables in SQL

- Created by CREATE TABLE statement

```
CREATE TABLE EMPLOYEE;
```

- Known as base tables
- Attributes ordered by creation order
- Rows not ordered

Description of the Relational Model

- All of the information stored in a Relational Database is held in relations

No other data structures!

- A relation may be thought of as a table

Student

name	id	exam1	exam2
Mounia	891023	12	58
Jane	891024	66	90
Thomas	891025	50	65

- A relation has:
  - a **name**
  - an unchanging set of **columns;** named and typed
  - a time varying set of **rows**

DDL – Create Table

CREATE TABLE *table_name*

({ *column_name data_type*

[DEFAULT *default_expr*] [

column_constraint [, ...]] |

table_constraint } [, ...])

- Data Types (PostgreSQL) include:
    - character(n) – fixed-length character string
    - character varying(n) – variable-length character string
    - smallint, integer, bigint, numeric, real, double precision
    - date, time, timestamp, …
    - serial - unique ID for indexing and cross reference

Data Types

- Numeric
  - Integer: INT
  - Real: FLOAT
- Character-string
  - Fixed length: CHAR(n)
  - Varying length: VARCHAR(n)
- DATE
  - Has main components YEAR, MONTH, DAY
  - Also stores century, hour, minute, second
  - Has format DD-MON-YYYY

    e.g. 05-FEB-2001

Constraints

- Recall that the schema defines the legal instances of the relations

- Data types are a way to limit the kind of data that can be stored in a table, but they are often insufficient
  - e.g. prices must be positive values
  - uniqueness, etc.

- Can specify constraints on individual columns or on tables

Column (attribute) Constraints

[CONSTRAINT *constraint_name*]

{ NOT NULL | NULL | UNIQUE | PRIMARY KEY |

CHECK (*expression*) |

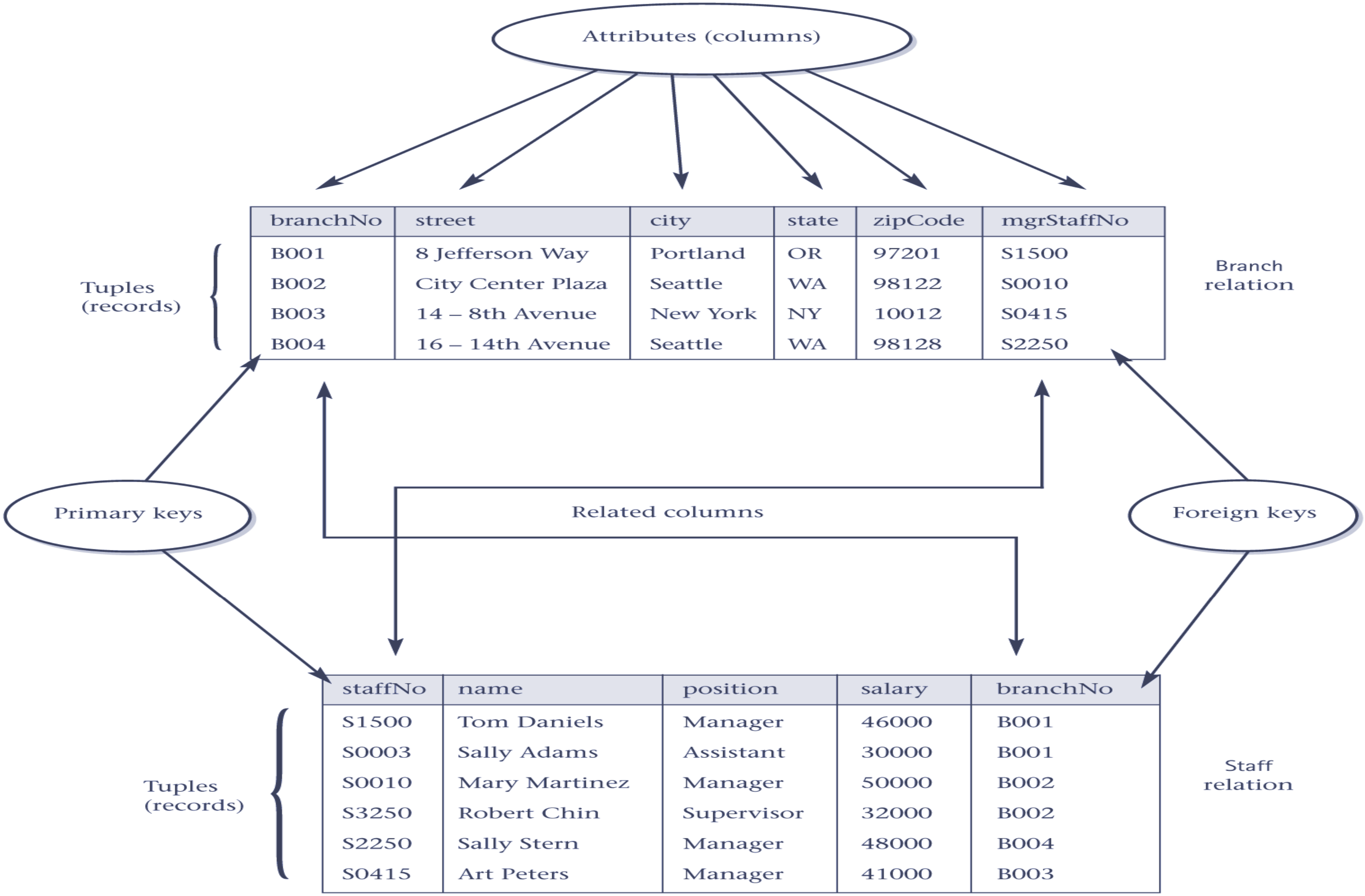
REFERENCES *reftable* [(*refcolumn*)]

[ON DELETE *action*] [ON UPDATE *action*] }

primary key = *unique + not null; also used as default target for references (can have at most 1)*

references is for foreign keys; *action* is one of:

NO ACTION, CASCADE, SET NULL, SET DEFAULT



Attributes (columns)

Tuples (records)

branchNo	street	city	state	zipCode	mgrStaffNo
B001	8 Jefferson Way	Portland	OR	97201	S1500
B002	City Center Plaza	Seattle	WA	98122	S0010
B003	14 – 8th Avenue	New York	NY	10012	S0415
B004	16 – 14th Avenue	Seattle	WA	98128	S2250

Branch relation

Primary keys

Related columns

Foreign keys

Tuples (records)

staffNo	name	position	salary	branchNo
S1500	Tom Daniels	Manager	46000	B001
S0003	Sally Adams	Assistant	30000	B001
S0010	Mary Martinez	Manager	50000	B002
S3250	Robert Chin	Supervisor	32000	B002
S2250	Sally Stern	Manager	48000	B004
S0415	Art Peters	Manager	41000	B003

Staff relation

Column Constraints: Default Values

- A type of column constraint
- Specified by DEFAULT <value>
- Used if no explicit value assigned to attribute
- NULL unless otherwise stated

Example of Column Constraints

```
CREATE TABLE sp
 (sno VARCHAR(5) NOT NULL REFERENCES s,
 pno VARCHAR(5) NOT NULL REFERENCES p,
 qty INT CHECK (qty >= 0),
 PRIMARY KEY (sno, pno))
```

Table Constraints

- PRIMARY KEY
- UNIQUE (secondary key)
- FOREIGN KEY (referential integrity)
  - Referential integrity constraints can be violated by
    - Insertion or deletion of tuples
    - Foreign key value modified
  - Referential triggered action
    - Can be added to foreign key constraint to cause automatic update: ON UPDATE and ON DELETE
    - Options are SET NULL, CASCADE and SET DEFAULT

Table Constraints

CREATE TABLE *table_name* ({

column_name data_type [DEFAULT *default_expr*] [*column_constraint* [, ...]] | *table_constraint* } [, ...])

- Table Constraints:

[CONSTRAINT *constraint_name*]

{ UNIQUE (*column_name* [, ...]) |

PRIMARY KEY (*column_name* [, ...]) |

CHECK (*expression*) |

FOREIGN KEY (*column_name* [, ...]) REFERENCES *reftable* [(*refcolumn* [, ...])] [ON DELETE *action*] [ON UPDATE *action*] }

Create Table (Examples)

```
CREATE TABLE films (
  code          CHAR(5) PRIMARY KEY,
  title         VARCHAR(40),
  did           DECIMAL(3),
  date_prod     DATE,
  kind          VARCHAR(10),
CONSTRAINT production UNIQUE(date_prod)
FOREIGN KEY did REFERENCES distributors
 ON DELETE NO ACTION
);
CREATE TABLE distributors (
  did    DECIMAL(3) PRIMARY KEY,
  name   VARCHAR(40)
  CONSTRAINT con1 CHECK (did > 100 AND name <> ‘ ’ )
);
```

Other DDL Statements

- ALTER TABLE
  - use to add/remove columns, constraints, rename things …
- DROP TABLE
  - Compare to “Delete * From Table”
- CREATE/DROP VIEW
- CREATE/DROP INDEX
- GRANT/REVOKE PRIVILEGES
  - SQL has an authorization model for saying who can read/modify/delete etc. data and who can grant and revoke privileges!

DROP TABLE

- Option
  - CASCADE CONSTRAINT
  - e.g. **DROP TABLE** PropertyForRent **CASCADE;**
    - The DROP operation drops all dependent on these objects (and objects dependent on these objects)
  - RESTRICT CONSTRAINT
  - e.g. **DROP TABLE** PropertyForRent **RESTRICT;**

ALTER TABLE [1]

- Command which allows
  - Adding column

```
ALTER TABLE EMPLOYEE ADD JOB VARCHAR(12);
```

  - Dropping column

```
ALTER TABLE EMPLOYEE DROP ADDRESS CASCADE;
```

ALTER TABLE [2]

- Changing column definition (add/drop default)

```
ALTER TABLE DEPARTMENT ALTER MGRSSN
DROP DEFAULT;

ALTER TABLE DEPARTMENT ALTER MGRSSN
SET DEFAULT ‘11111111’;
```

ALTER TABLE [3]

- Adding / dropping table constraints

```
ALTER TABLE EMPLOYEE DROP CONSTRAINT EMPSUPERFK CASCADE;

ALTER TABLE EMPLOYEE ADD CONSTRAINT EMPSUPERFK;

FOREIGN KEY(SUPERSSN) REFERENCES EMPLOYEE(SSN)
ON DELETE SET NULL;
```

Part 2: SQL Data Manipulation (DML)

- Single-table queries are straightforward
  - Example Query 0

To find all 18 year old students, we can write:

```
SELECT *
  FROM Students S
  WHERE S.age=18
```

To find just names and logins, replace the first line:

```
SELECT S.name, S.login
```

Querying Multiple Relations

- Can specify a join over two tables as follows:

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

sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2

result =

S.name	E.cid
Jones	History105

Note: obviously no referential integrity constraints have been used here

Basic SQL Query

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

relation-list : A list of relation names

- possibly with a *range-variable* after each name

target-list : A list of attributes of tables in *relation-list*

qualification : Comparisons combined using AND, OR and NOT

- Comparisons are Attr *op* const or Attr1 *op* Attr2, where *op* is one of $= \neq < > \leq \geq$

DISTINCT: optional keyword indicating that the answer should not contain duplicates

- In SQL SELECT, the default is that duplicates are *not* eliminated! (Result is called a "multiset")

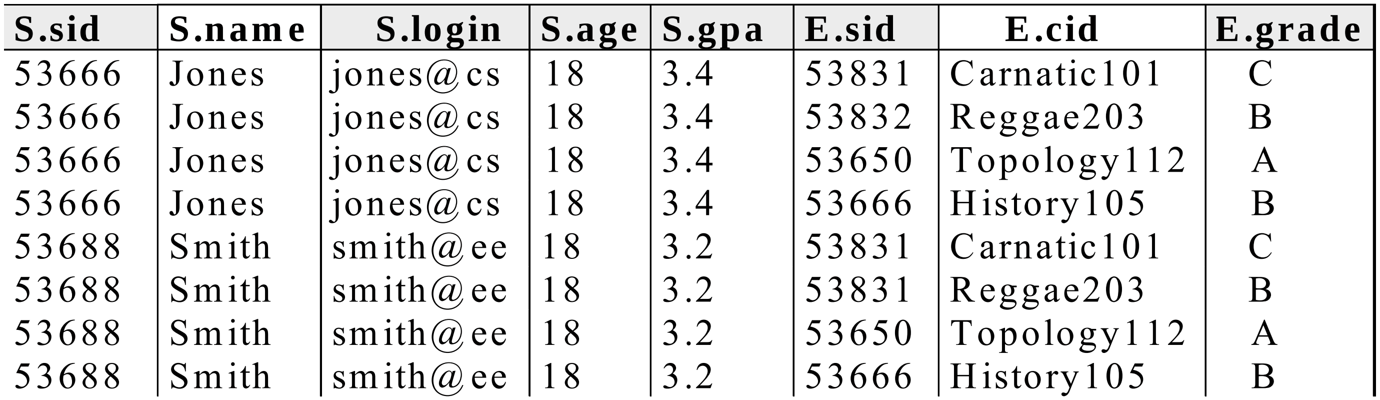
Query Semantics

- Semantics of an SQL query are defined in terms of the following conceptual evaluation strategy:
  1. do FROM clause: Compute *cross-product* of tables (e.g. Students and Enrolled)
  2. do WHERE clause: Check conditions, discard tuples that fail, i.e. “selection”
  3. do SELECT clause: Delete unwanted fields, i.e. “projection”
  4. if DISTINCT specified, eliminate duplicate rows

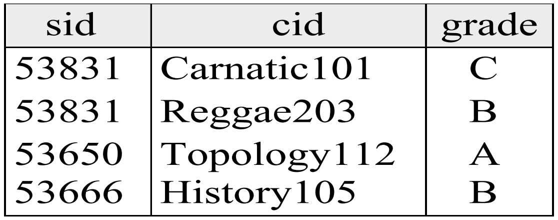
Probably the least efficient way to compute a query!

- An optimizer will find more efficient strategies to get the *same answer*

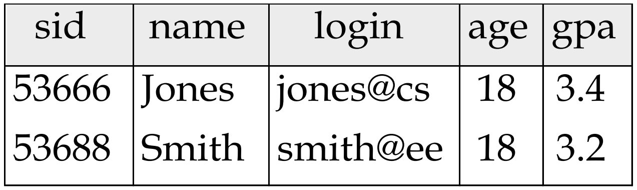
Step 1 – Cross Product



S.sid	S.name	S.login	S.age	S.gpa	E.sid	E.cid	E.grade
53666	Jones	jones@cs	18	3.4	53831	Carnatic101	C
53666	Jones	jones@cs	18	3.4	53832	Reggae203	B
53666	Jones	jones@cs	18	3.4	53650	Topology112	A
53666	Jones	jones@cs	18	3.4	53666	History105	B
53688	Smith	smith@ee	18	3.2	53831	Carnatic101	C
53688	Smith	smith@ee	18	3.2	53831	Reggae203	B
53688	Smith	smith@ee	18	3.2	53650	Topology112	A
53688	Smith	smith@ee	18	3.2	53666	History105	B

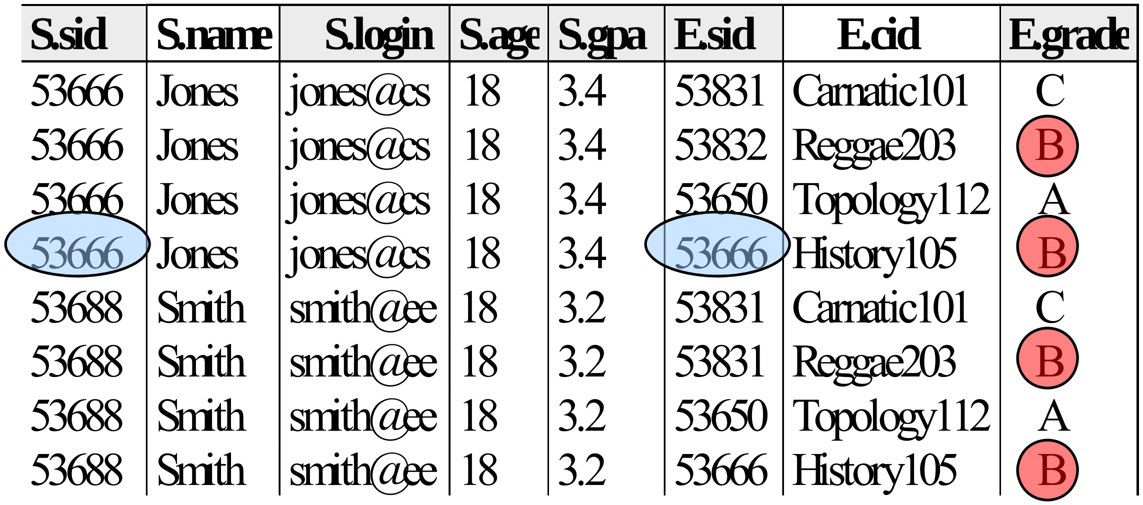


sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B



sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2

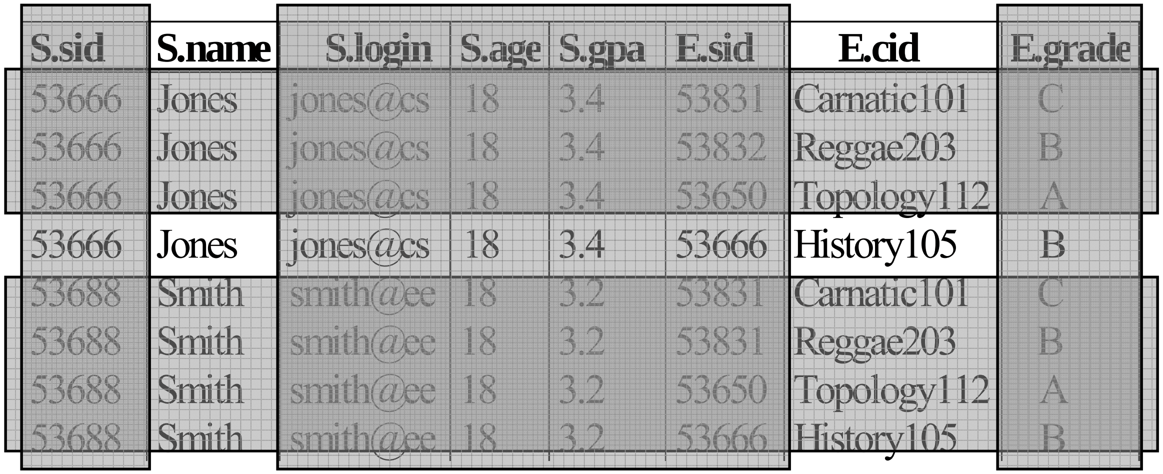
Step 2 - Discard Tuples that Fail Predicate



S.sid	S.name	S.login	S.age	S.gpa	E.sid	E.cid	E.grade
53666	Jones	jones@cs	18	3.4	53831	Carnatic101	C
53666	Jones	jones@cs	18	3.4	53832	Reggae203	B
53666	Jones	jones@cs	18	3.4	53650	Topology112	A
53666	Jones	jones@cs	18	3.4	53666	History105	B
53688	Smith	smith@ee	18	3.2	53831	Carnatic101	C
53688	Smith	smith@ee	18	3.2	53831	Reggae203	B
53688	Smith	smith@ee	18	3.2	53650	Topology112	A
53688	Smith	smith@ee	18	3.2	53666	History105	B

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

Step 3 - Discard Unwanted Columns



S.sid	S.name	S.login	S.age	S.gpa	E.sid	E.cid	E.grade
53666	Jones	jones@cs	18	3.4	53831	Carnatic101	C
53666	Jones	jones@cs	18	3.4	53832	Reggae203	B
53666	Jones	jones@cs	18	3.4	53650	Topology112	A
53666	Jones	jones@cs	18	3.4	53666	History105	B
53688	Smith	smith@ee	18	3.2	53831	Carnatic101	C
53688	Smith	smith@ee	18	3.2	53831	Reggae203	B
53688	Smith	smith@ee	18	3.2	53650	Topology112	A
53688	Smith	smith@ee	18	3.2	53666	History105	B

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

Now the Details…

We will use these relations in our examples

Reserves

sid	bid	day
22	101	10/10/96
95	103	11/12/96

Sailors

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

Boats

bid	bname	color	
101	Interlake	blue	
102	Interlake	red	
103	Clipper	green	
104	Marine	red	

Example Schemas (in SQL DDL)

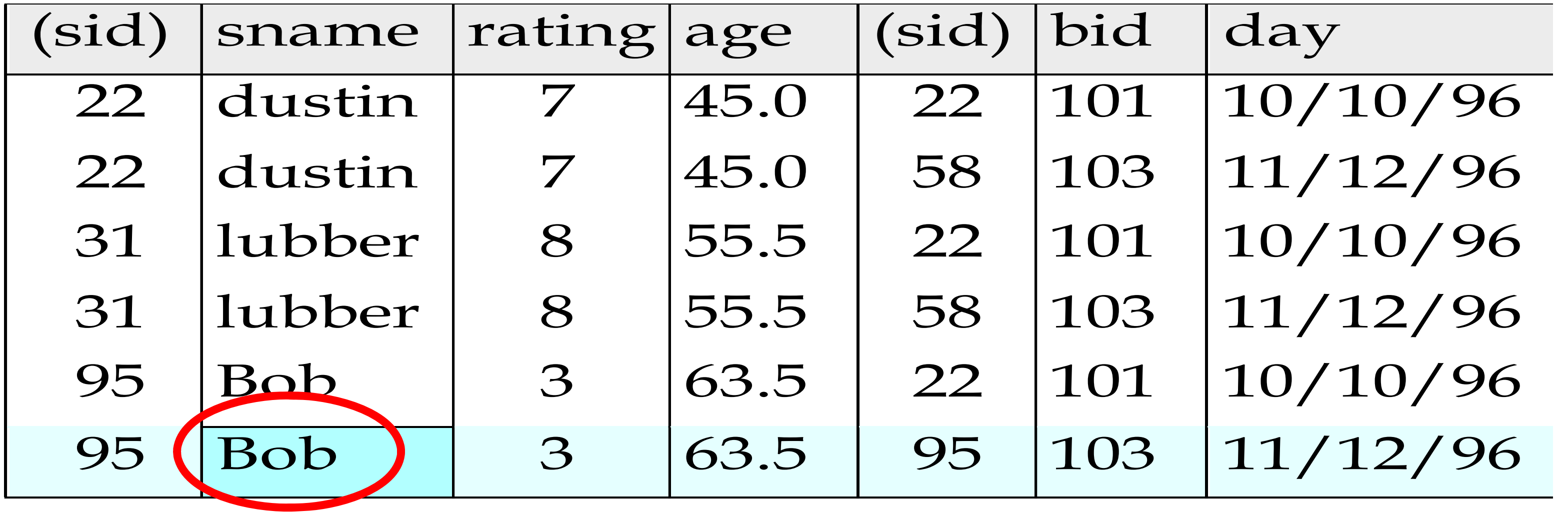
```
CREATE TABLE Sailors (sid INTEGER, sname CHAR(20),rating INTEGER,
 age REAL, PRIMARY KEY sid)


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


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

Another Join Query

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



(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
95	Bob	3	63.5	22	101	10/10/96
95	Bob	3	63.5	95	103	11/12/96

Some Notes on Aliasing

- Can associate an alias with the tables in the FROM clause
  - saves writing, makes queries easier to understand
- Needed when ambiguity could arise
  - for example, if same table used multiple times in same FROM (called a "self-join")

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

Can be rewritten using range variables as:

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

More Notes

- Here’s an example where aliases are required (self-join example):

```
SELECT  x.sname, x.age, y.sname, y.age
FROM  Sailors x, Sailors y
WHERE  x.age > y.age
```

Note that target list can be replaced by “*” if you don’t want to do a projection:

```
SELECT *
FROM Sailors x
WHERE x.age > 20
```

Find sailors who’ve reserved at least one boat

```
SELECT  S.sid
FROM  Sailors S, Reserves R
WHERE  S.sid=R.sid
```

Would adding DISTINCT to this query make a difference?

What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause?

- Would adding DISTINCT to this variant of the query make a difference?

Expressions

- Can use arithmetic expressions in SELECT clause (plus other operations we will discuss later)

Use AS to provide column names (Example - q1b)

```
SELECT S.age, S.age-5 AS age1, 2*S.age AS age2
FROM  Sailors S
WHERE  S.sname = ‘dustin’
```

Can also have expressions in WHERE clause:

```
SELECT  S1.sname AS name1, S2.sname AS name2
FROM  Sailors S1, Sailors S2
WHERE  2*S1.rating = S2.rating - 1
```

No WHERE Clause

- No condition on tuple selection
- Example - query 9

- More than one relation in FROM clause means cross product
- Example - query 10
- Similar to relational algebra cross product - PROJECT combination

Use of Asterisk

- Used to retrieve all attribute values in SELECT clause
- Examples - queries 1C, 1D, 10A

String Operations

- SQL also supports some string operations
- Example - queries 12, 12A
- “LIKE” is used for string matching

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

‘_’ stands for any one character and ‘%’ stands for 0 or more arbitrary characters.

Find ‘sid’s of sailors who have reserved a red or a green boat (Example - queries 11, 11A)

- UNION can be used to compute the union of any two union-compatible sets of tuples (which are themselves the result of SQL queries)

```
SELECT DISTINCT R.sid
FROM Boats B,Reserves R
WHERE R.bid=B.bid AND
(B.color=‘red’OR B.color=‘green’)
```

Vs.

(note: UNION eliminates duplicates by default. Override w/ UNION ALL)

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

Nested Queries

- Powerful feature of SQL: WHERE clause can itself contain an SQL query! (Example - query 16)
  - Actually, so can FROM and HAVING clauses

Names of sailors who've reserved boat #103:

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

To find sailors who've ***not*** reserved #103, use NOT IN

To understand semantics of nested queries:

- think of a *nested loops* evaluation: *For each Sailors tuple, check the qualification by computing the subquery*

Nested Queries with Correlation

Find names of sailors who've reserved boat #103:

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

- EXISTS is another set comparison operator, like IN
- Can also specify NOT EXISTS
- Subquery must be recomputed for each Sailors tuple
  - Think of subquery as a function call that runs a query!

Other Comparison Operators

- >, >=, <, <=, < >
- Can be used with ANY, SOME, ALL

```
SELECT    LNAME, FNAME
FROM      EMPLOYEE
WHERE     SALARY > ALL        (SELECT SALARY
                               FROM   EMPLOYEE
                               WHERE   DNO = 5);
```

```
SELECT LNAME, SALARY

FROM EMPLOYESS

WHERE SALARY > ALL (1600, 2999);
```

Arithmetic and Other Operators

- Standard arithmetic operators can be applied
- Example - query 13
- String concatenation ||
- Numeric value range BETWEEN
- Example - query 14
- Ordering by value of one or more attributes
- Example - query 15

Aggregate Operators

- Significant extension of relational algebra
- Example queries: 19, 20, 21, 22, 23, 5

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

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

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

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

single column

Find name and age of the oldest sailor(s)

- The first query is incorrect!
  - Returns the sname of each sailor along with the (same) age of the oldest sailor

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

- The second query is correct!

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

GROUP BY and HAVING

- So far, we have 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 $i = 1, 2, \ldots, 10$:

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

Queries with GROUP BY

- To generate values for a column based on groups of rows, use aggregate functions in SELECT statements with the GROUP BY clause

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

The *target-list* contains:

I. list of column names &

II. terms with aggregate operations e.g. MIN (S.age)

column name list (I) can contain only attributes from the *grouping-list*

GROUP BY Examples (Example - queries 24&25)

For each rating, find the average age of the sailors

```
SELECT S.rating, AVG (S.age)
FROM Sailors S
GROUP BY S.rating
```

For each rating find the age of the youngest sailor with age ≥ 18

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
```

Conceptual Evaluation

- The cross-product of relation-list is computed, tuples that fail qualification are discarded, `unnecessary' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in grouping-list

- One answer tuple is generated per qualifying group

```
SELECT  S.rating,  MIN (S.age)
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
```

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
71	zorba	10	16.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

1. Form cross product

rating	age
1	33.0
7	45.0
7	35.0
8	55.5
10	35.0

2. Delete unneeded columns, rows; form groups

3. Perform Aggregation

rating	age
1	33.0
7	35.0
8	55.0
10	35.0

Queries with GROUP BY and HAVING

```
SELET    [DISTINCT] target-list
FROM      relation-list
WHERE     qualification
GROUP BY  grouping-list
HAVING    group-qualification
```

- Use the HAVING clause with the GROUP BY clause to restrict which group-rows are returned in the result set

(Example - queries 26, 28)

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

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

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
71	zorba	10	16.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

rating	age
1	33.0
7	45.0
7	35.0
8	55.5
10	35.0

rating	m-age	count
1	33.0	1
7	35.0	2
8	55.0	1
10	35.0	1

rating	
7	35.0

Answer

Summary of SQL Queries

```
SELECT <attribute and function list>
FROM <table list>
[WHERE <condition>]
[GROUP BY <grouping attribute(s)>]
[HAVING <group condition>]
[ORDER BY <attribute list>]
```

INSERT

```
INSERT [INTO] table_name [(column_list)]
VALUES (value_list)

INSERT [INTO] table_name [(column_list)]
<select statement>
```

INSERT INTO Boats VALUES (105, ‘Clipper’, ‘purple’)

INSERT INTO Boats (bid, color) VALUES (99, ‘yellow’)

- You can also do a “bulk insert” of values from one table into another (must be type compatible):

INSERT INTO TEMP(bid)

SELECT r.bid FROM Reserves R WHERE r.sid = 22

(Example - update 1)

DELETE & UPDATE

```
DELETE   [FROM]   table_name
[WHERE    qualification]
```

DELETE FROM Boats WHERE color = ‘red’

DELETE FROM Boats b
WHERE b. bid =
   (SELECT r.bid FROM Reserves R WHERE r.sid = 22)

- Can also modify tuples using UPDATE statement:

UPDATE Boats
SET Color = “green”
WHERE bid = 103;

(Example - updates 6, 5)

Views

```
CREATE VIEW view_name
AS select_statement
```

- Makes development simpler
- Often used for security
- Not instantiated - makes updates tricky

```
CREATE VIEW Reds
AS SELECT  B.bid,  COUNT (*) AS scount
    FROM Boats B, Reserves R
    WHERE  R.bid=B.bid AND   B.color= ‘red’
     GROUP BY  B.bid
```

```
CREATE VIEW Reds AS
    SELECT  B.bid,  COUNT (*) AS scount
    FROM Boats B, Reserves R
    WHERE  R.bid=B.bid AND B.color= ‘red’
     GROUP BY  B.bid
```

b.bid	scount
102	1

Reds

Querying and Deleting Views

- SQL queries can be specified on views

(Example - query view 1)

- Deleted using DROP VIEW

(Example - view 1A)

Assertions [1]

- More general constraints can be specified via declarative assertions
- Created using CREATE ASSERTION
- Deleted using DROP ASSERTION

Assertions [2]

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS (SELECT * FROM EMPLOYEE E,
                             EMPLOYEE M, DEPARTMENT D
                  WHERE E.SALARY > M.SALARY AND
                            E.DNO = D.DNUMBER AND
                            D.MGRSSN = M.SSN) );
```

- CHECK clause can be used with CREATE DOMAIN statement:

```
CREATE DOMAIN D_NUM AS INTEGER
CHECK (D_NUM > 0 AND D_NUM < 21);
```

```
SELECT r.sid, b.bid, b.name
FROM Reserves r FULL OUTER JOIN Boats b
ON r.bid = b.bid
```

sid	bid	day
22	101	10/10/96
95	103	11/12/96

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

r.sid	b.bid	b.name
22	101	Interlake
	102	Interlake
95	103	Clipper
	104	Marine

Note: in this case it is the same as the ROJ because bid is a foreign key in reserves, so all reservations must have a corresponding tuple in boats

Division in SQL

Find names of sailors who’ve reserved all boats

Example in book, not using EXCEPT:

```
SELECT  S.sname         Sailors S such that …
FROM  Sailors S
WHERE  NOT EXISTS  (SELECT  B.bid
there is no boat B  FROM  Boats B
                    WHERE  NOT EXISTS  (SELECT  R.bid
that doesn’t have …                     FROM  Reserves R
                                        WHERE  R.bid=B.bid
a Reserves tuple showing S reserved B       AND R.sid=S.sid))
```

Find the number of reservations for each red boat

- Grouping over a join of two relations

```
SELECT B.bid, COUNT(*) AS scount
FROM Boats B, Reserves R
WHERE  R.bid=B.bid
      AND B.color= ‘red’
GROUP BY  B.bid
```

```
SELECT  B.bid,  COUNT (*) AS scount
FROM Boats B, Reserves R
WHERE  R.bid=B.bid AND B.color= ‘red’
GROUP BY  B.bid
```

b.bid	b.color	r.bid
101	blue	101
102	red	101
103	green	101
104	red	101
101	blue	102
102	red	102
103	green	102
104	red	102

1

b.bid	b.color	r.bid
102	red	102

2

b.bid	scount
102	1

answer

Null Values – 3 Valued Logic

(null > 0)	is null
(null + 1)	is null
(null = 0)	is null
null AND true	is null

AND	T	F	Null
T	T	F	Null
F	F	F	F
NULL	Null	F	Null

OR	T	F	Null
T	T	T	T
F	T	F	Null
NULL	T	Null	Null

Joins

```
SELECT (column_list)
FROM  table_name
 [INNER |{LEFT|RIGHT|FULL} OUTER] JOIN table_name
   ON qualification_list
WHERE …
```

- Explicit join semantics needed unless it is an INNER join (INNER is default)

Inner Join

- Only the rows that match the search conditions are returned

```
SELECT s.sid, s.name, r.bid
FROM Sailors s INNER JOIN Reserves r
ON s.sid = r.sid
```

Returns only those sailors who have reserved boats

SQL-92 also allows:

```
SELECT s.sid, s.name, r.bid
FROM Sailors s NATURAL JOIN Reserves r
```

- “NATURAL” in SQL means equi-join for each pair of attributes with the same name (may need to rename with “AS”)

```
SELECT s.sid, s.name, r.bid
FROM Sailors s INNER JOIN Reserves r
ON s.sid = r.sid
```

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

sid	bid	day
22	101	10/10/96
95	103	11/12/96

s.sid	s.name	r.bid
22	Dustin	101
95	Bob	103

Left Outer Join

- Left Outer Join returns all matched rows, plus all unmatched rows from the table on the left of the join clause (use nulls in fields of non-matching tuples)

```
SELECT s.sid, s.name, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid
```

Returns all sailors & information on whether they have reserved boats

```
SELECT s.sid, s.name, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid
```

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

sid	bid	day
22	101	10/10/96
95	103	11/12/96

s.sid	s.name	r.bid
22	Dustin	101
95	Bob	103
31	Lubber	

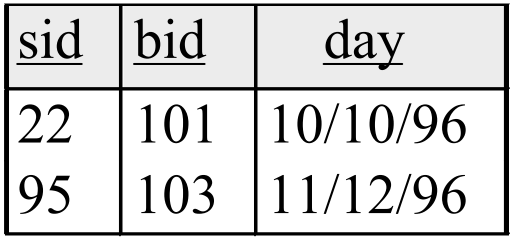
Right Outer Join

- Right Outer Join returns all matched rows, plus all unmatched rows from the table on the right of the join clause

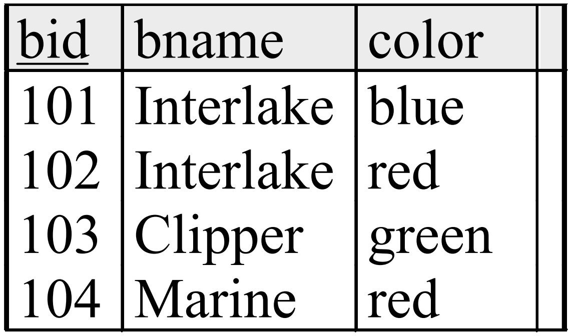
```
SELECT r.sid, b.bid, b.name
FROM Reserves r RIGHT OUTER JOIN Boats b
ON r.bid = b.bid
```

Returns all boats & information on which ones are reserved

```
SELECT r.sid, b.bid, b.name
FROM Reserves r RIGHT OUTER JOIN Boats b
ON r.bid = b.bid
```



sid	bid	day
22	101	10/10/96
95	103	11/12/96



bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

r.sid	b.bid	b.name
22	101	Interlake
	102	Interlake
95	103	Clipper
	104	Marine

Full Outer Join

- Full Outer Join returns all (matched or unmatched) rows from the tables on both sides of the join clause

```
SELECT r.sid, b.bid, b.name
FROM Reserves r FULL OUTER JOIN Boats b
ON r.bid = b.bid
```

Returns all boats & all information on reservations

Find names of sailors who’ve reserved all boats

Example in book, not using EXCEPT:

```
SELECT  S.sname
FROM  Sailors S        Sailors S such that …
WHERE  NOT EXISTS  (SELECT  B.bid
                    FROM  Boats B        there is no boat B without …
                    WHERE  NOT EXISTS  (SELECT  R.bid
  a Reserves tuple showing S reserved B  FROM  Reserves R
                                         WHERE  R.bid=B.bid
                                            AND R.sid=S.sid))
```

Find names of sailors who’ve reserved all boats

Can you do this using Group By and Having?

```
SELECT  S.sname
FROM  Sailors S, reserves R
WHERE  S.sid = R.sid
  GROUP BY S.sname, S.sid
  HAVING
           COUNT(DISTINCT R.bid) =
                    (Select COUNT (*) FROM Boats)
```

Note: must have both sid and name in the GROUP BY clause. Why?

Sailors who have reserved all boats

```
SELECT  S.name
FROM  Sailors S, reserves R
WHERE  S.sid = R.sid
GROUP BY S.name, S.sid
HAVING COUNT(DISTINCT R.bid) =
             (Select COUNT (*) FROM Boats)
```

sname	sid	bid
Frodo	1	102
Bilbo	2	101
Bilbo	2	102
Frodo	1	102
Bilbo	2	103

sname	sid	bid
Frodo	1	102,102
Bilbo	2	101, 102, 103

sname	sid	count
Frodo	1	1
Bilbo	2	3

count
3

Sailors

sid	sname	rating	age
1	Frodo	7	22
2	Bilbo	2	39
3	Sam	8	27

Boats

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

Reserves

sid	bid	day
1	102	9/12
2	102	9/12
2	101	9/14
1	102	9/10
2	103	9/13