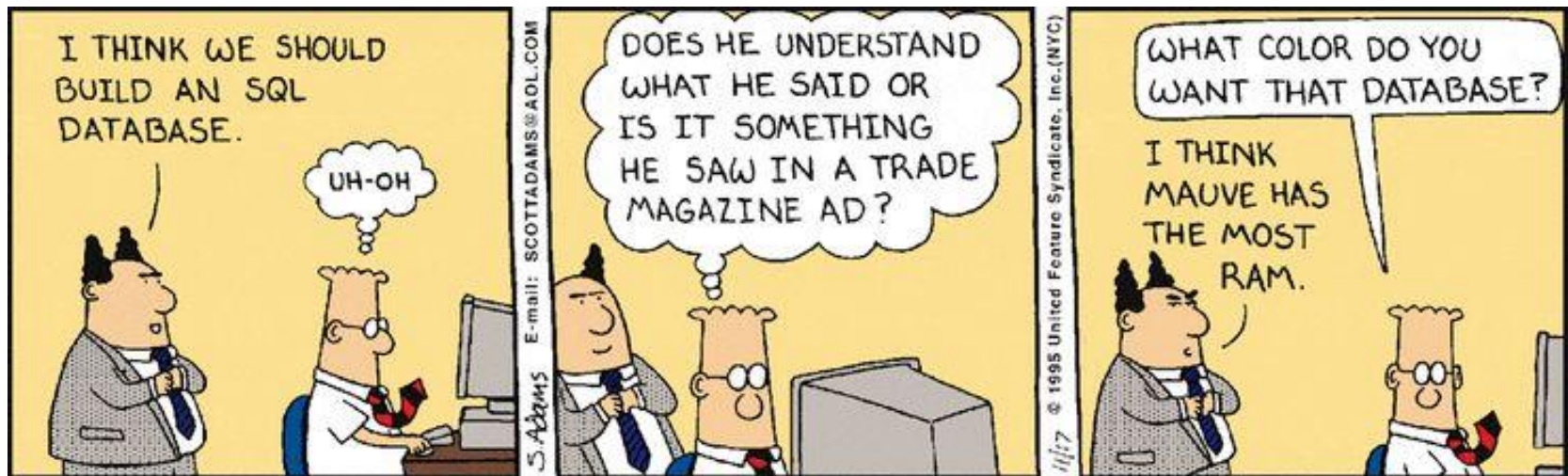


SQL



SQL tutorial:

<http://www.w3schools.com/sql/default.asp>

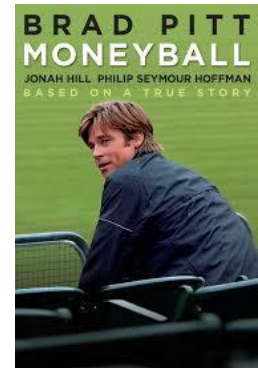
Outline this week

Today

- Why databases?
- What is a database anyway?
 - History of databases
- Important DMBS features
- Relational data model
 - Why its great
 - What it looks like (SQL)

Friday

- On Friday we will work on a DB example using Money Ball data



- Next week we will cover crawling and other data storage topics.

Why Databases?

- In the early days, database applications were built on top of *file systems*.
- Drawbacks of using file systems to store data:
 - **Data redundancy and inconsistency**
 - Multiple file formations, duplication of information in different files
 - **Difficulty in accessing data**
 - Need to write a new program to carry out each new task
 - **Data Isolation** – multiple files and formats
 - **Integrity problems**
 - Enforcing integrity constraints
 - Adding / changing existing constraints

Why Databases?

- Drawbacks of using file systems (cont.)
 - **Atomicity of updates**
 - Failures may leave database in an inconsistent state with partial updates carried out
 - **Concurrent access by multiple users**
 - Concurrent access needed for performance
 - Uncontrolled concurrent accesses can lead to inconsistencies
 - i.e. Two people reading a balance and updating it at the same time
 - **Security problems**

Students

Name, Year, Major, Enrollment

Mary Jane, 2017 , CS , [CS135 – Web Dev, CS105 - DS, CS181-SW ENG]

John Smith , 2018 , Math , [Math105 – Linear Alg, CS135- Web Dev]

Erin Key , 2017 , Econ , [CS181 – SW ENG, CS105- DS]

...

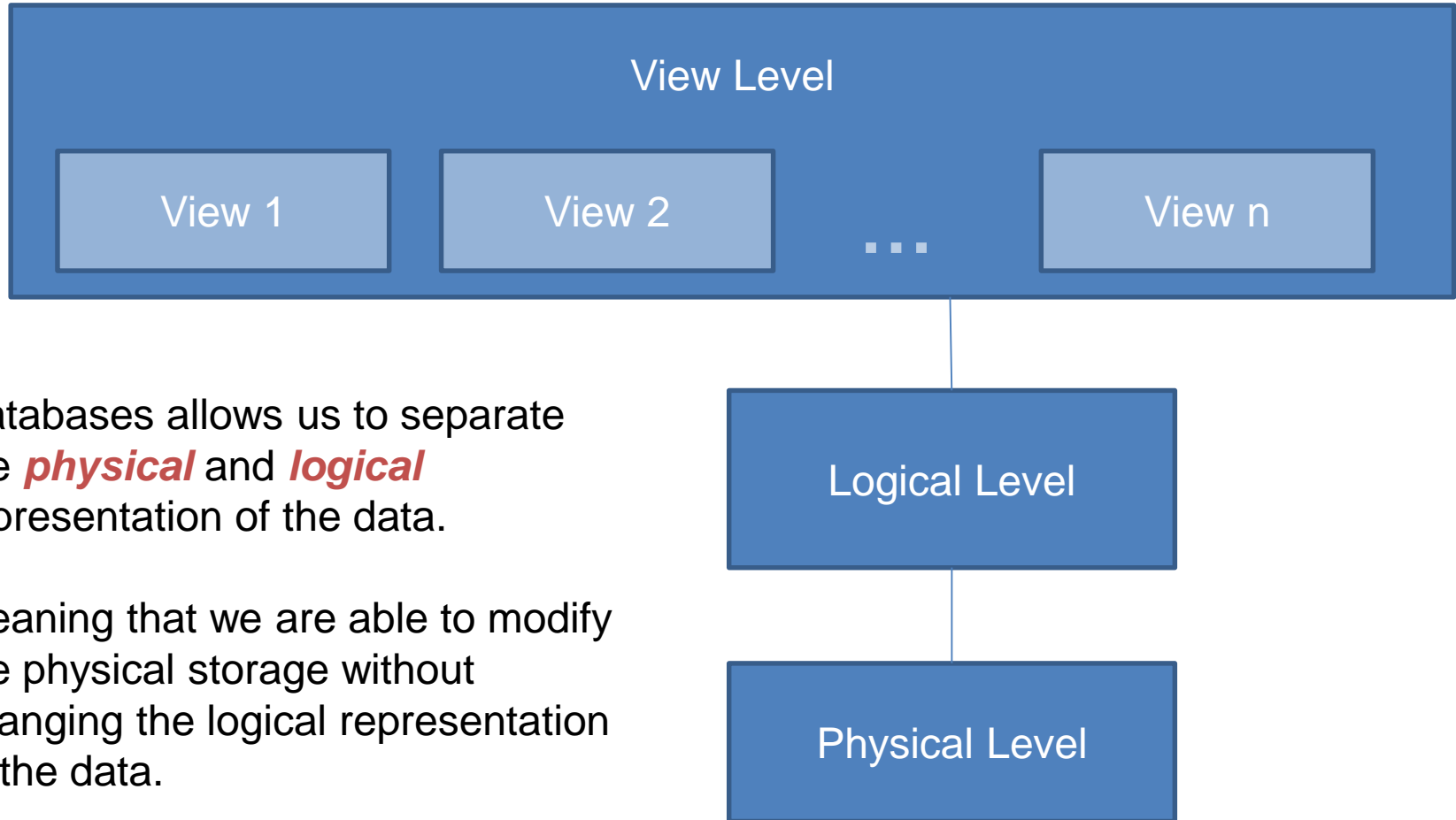
- What we add/remove courses, or update course information such as name? ... we would need to update in all relevant places!!
- What if we want to answer the query “How many students are registered for CS 105? ... The programmer has to write specific code to form the query!
- What if we want to enforce constraint that each student is registered for at least 1 course? ... Its hard to enforce as students add/drop, and courses are canceled and added continuously.

Database systems offer solutions to all the above problems

Timeline of Databases

- **1960s** – hierarchical databases which provided support for concurrency, recover, and fast access.
- **1970-1972** - Edgar Codd who was working at IBM proposed the 'relational database model'. Provided support for more reliability, less redundancy, more flexibility, etc.
- **1970s** – two major RDBMS prototypes were proposed: Ingres and System R
- **Mid 1970s** – A DB model called Entity –Relationship(ER) was proposed
- **1980s** – Structured Query Language (SQL) became standard querying language.
- **Late 1980s - 1990s** – Parallel and distributed databases
- **2000s & Now** – NoSQL databases

Architecture of Databases



- Databases allows us to separate the **physical** and **logical** representation of the data.
- Meaning that we are able to modify the physical storage without changing the logical representation of the data.

Relational DBMS to the rescue

- **Relational data model:** data is stored in relations

- Example: Banking Info

Account	Branch	Name	Balance
4500	Pomona	Mary	18,300
6831	Claremont	John	15,000
9834	Pasadena	Erin	11,000

- A declarative query language
 - Specify what answer a query should return, but not how the query is executed
 - Ex. SQL – structured query language
 - Query : What is Mary's balance?

```
SELECT    balance
FROM      Banking
WHERE     name = "Mary" ;
```


Relational Model: Levels of Abstraction

- Conceptual/Logical schema Describes the data model
 - **Students**(*sid*: string, *name*: string, *login*: string, *gpa*: real)
 - **Courses**(*cid*: string, *cname*: string, *credits*: integer)
 - **Enrollment**(*sid*: string, *cid*: string, *grade*: string)
- Physical schema Storage details
 - Store the relations as unsorted files
 - Create **indexes** on **Students.sid** and **Courses.sid**
- External schema (“views”) Allow Customized data access
 - View each course's enrollment

```
CREATE VIEW CourseInfo AS
SELECT cid, COUNT (*) as enrollment
FROM Enrolled
GROUP BY cid;
```
 - **CoureInfo** (*cid*: string, *enrollment*: integer)

Relational DBMS to the rescue

- **Relational data model:** data is stored in relations

- Example: Banking Info

Account	Branch	Name	Balance
4500	Pomona	Mary	18,300
6831	Claremont	John	15,000
9834	Pasadena	Erin	11,000

- A declarative query language
 - Specify what answer a query should return, but not how the query is executed
 - Ex. SQL – structured query language
 - Query : What is Mary's balance?

```
SELECT    balance
FROM      Banking
WHERE     name = "Mary" ;
```

Relational Model: Levels of Abstraction

- Conceptual/Logical schema Describes the data model
 - **Students**(*sid*: string, *name*: string, *login*: string, *gpa*: real)
 - **Courses**(*cid*: string, *cname*: string, *credits*: integer)
 - **Enrollment**(*sid*: string, *cid*: string, *grade*: string)
- Physical schema Storage details
 - Store the relations as unsorted files
 - Create **indexes** on **Students.sid** and **Courses.sid**
- External schema (“views”) Allow Customized data access
 - View each course's enrollment

```
CREATE VIEW CourseInfo AS
SELECT cid, COUNT (*) as enrollment
FROM Enrolled
GROUP BY cid;
```
 - **CoureInfo** (*cid*: string, *enrollment*: integer)

ACID Properties

- **Atomicity** – system should ensure that updates of a partially executed transaction are not reflected in the database.
- **Consistency** – system should ensure that any changes to values in an instance are consistent with changes to other values in the same instance.
- **Durability** – system should ensure updates of committed transactions is critical.
- **Isolation** – system should ensure that transactions that occur in parallel will have same effect if they were run sequentially.

ACID (Cont.) - System Failures

- Banking example ... balance transfer
 - Decrement account X by \$100
 - Increment account Y by \$100
- What if power goes out after first instruction?
 - If first instruction is executed but not the second, then operations are not atomic.
 - DBMS must keep a log of updates, and upon system failure the DBMS will replay the log checking the status of the records to recover database to a consistent state.

ACID (Cont.) - Parallel Transactions

- Transaction 1 – Deposit to account X
- Transaction 2 – Add interest to account X

Transaction 1

Lookup balance of
account X

Deposit 2 times the
balance of account x

Transaction 2

Lookup balance of
account X

Add 3% to balance
of account X

This is an example
of lost update...

There are many
other scenarios
that cause issues
when we don't
consider the order
of transactions
running in parallel.

Structured Query Language (SQL)

- SQL was proposed in 1970s by D. Chamberlin and R. Boyce
- Data definition language (DDL)
 - Define the schema (create, change, delete relations)
 - Specify constraints, user permissions
 - Ex. `CREATE TABLE Students (sid string, name string,);`
- Data modification language (DML)
 - Find data that matches criteria
 - Add, remove, update data
 - The DBMS is responsible for efficient evaluation
 - Ex. `SELECT * FROM Students where name = "Mary";`

SQL: Creating Relations

- Create **Students** relation:

```
CREATE TABLE Students (  
    sid CHAR(20),  
    name CHAR(20),  
    login CHAR(20),  
    SSN CHAR(12),  
    gpa FLOAT  
);
```

- Create **Enrolled** relation:

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

Insert a single tuple

```
INSERT INTO Students (sid, name, login, SSN, gpa) VALUES (21,  
    "Mary", "marys", "000-00-0000", 3.4);
```

Delete tuples that satisfy condition

```
DELETE FROM Students S  
WHERE S.name = "Mary";
```


Integrity Constraints

- Students

SID	Name	Login	SSN	GPA
45	Mary	maryS	000-000-000	3.4
67	John	johnT	000-000-000	3.5
78	Erin	erinK	000-000-000	3.7



Primary Key

- Enrollment

SID	CID	Grade
45	Mary	100
67	John	99
78	Erin	89



Foreign Key

- The Primary Key is a field that uniquely identifies a tuple (a super key is a set of fields)
- A Foreign Key is a key in one relation refers to a primary key of another relation.

SQL: Creating Relations

- Create **Students** relation:

```
CREATE TABLE Students (  
    sid CHAR(20),  
    name CHAR(20),  
    login CHAR(20),  
    SSN CHAR(12),  
    gpa FLOAT  
    PRIMARY KEY(sid),  
    UNIQUE (SSN)  
);
```

- Create **Enrolled** relation:

```
CREATE TABLE Enrolled (  
    sid CHAR(20),  
    cid CHAR(20),  
    grade FLOAT,  
    PRIMARY KEY (sid,cid),  
    FOREIGN KEY (sid)  
);
```

Basic SQL Query

Selecting fields of interest

→ **SELECT**

Set of relations

→ **FROM**

Discard tuples that fail condition

→ **WHERE**

Order tuples in result

→ **ORDER BY**

Limit number of tuples in the result

→ **LIMIT**

Remove duplicate tuples

[**DISTINCT**] target-list

relation-list

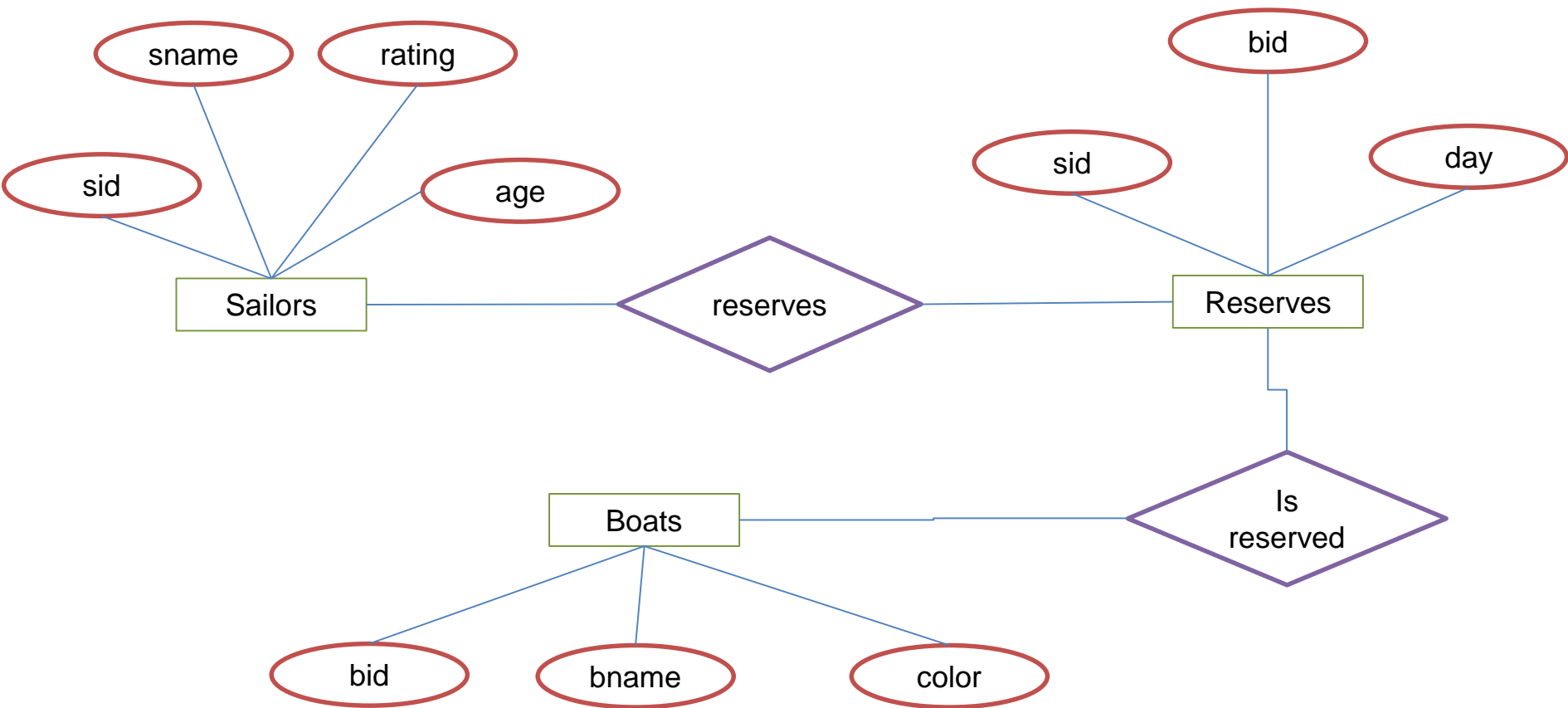
qualification

fields(s) [**ASC|DESC**]

num_rows

What actually happens when you write a SQL query?? Well, the query is optimized before execution... but we still should try to write efficient queries.

Example



Visualizing Query Evaluation

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

Sailors

sid	name	rating	age

Reserves

sid	bid	day

Join condition: are these the same sid?

Is this bid 103 ??

Example Relation Instances

Sailors

Sid v	name	rating	age
22	Dustin	7	45
31	Lubber	8	55.5
95	Bob	3	63.5

Boats

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

Reserves

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

Range Variables

- We can associate “range variables” with the relations in the FROM clause
 - Saves writing, makes queries easier to understand
 - Like an alias
- Needed when ambiguity could arise
 - For example, if the same relation used multiple times in the same FROM clause (called self-join)

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

QUERY: Find all the Sailors who have reserved at least 1 boat

Range Variables

- Example where range variables are required (self-join example)

```
SELECT      S1.sname, S1.age, S2.name, S2.age
FROM        Sailors S1, Sailors S2
WHERE       S1.age = S2.age AND
           S1.rating = S2.rating;
```

Since we are doing a self-join, we need to use the “Range Variables”

NULL Values

- Field values in a tuple are sometimes missing
 - **Unknown** (e.g. a rating or grade has not been assigned)
 - **Inapplicable** (e.g. no spouse's name)
 - SQL provides a special value null for such situations.
- The presence of null complicates things
 - Is “rating > 8” true or false when rating is null?
 - It is not possible to test for NULL values with comparison operators, such as =, <, or <>.
 - Proper way: check if a value is not null using **IS NULL**

Lastname	FirstName	Address
Smith	Dustin	
Hansen	Lubber	
Patterson	Bob	

```
SELECT LastName, FirstName, Address
FROM    Persons
WHERE    Address IS NULL
```

Null Values – 3 Valued Logic

- We need a 3-valued logic:
 - Values : True, False, and **Unknown**

AND	T	F	NULL
T	T	F	Unknown
F	F	F	F
NULL	Unknown	F	Unknown

OR	T	F	NULL
T	T	T	T
F	T	F	Unknown
NULL	T	Unknown	Unknown

Expressions

- Can use arithmetic expressions in SELECT clause (plus other calculations we'll discuss later)
- Use AS to provide column names

```
SELECT    S.sname, S.rating %2 AS evenOddRating
FROM      Sailors S
WHERE     S.age >= 18;
```

- 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;
```

- QUERY: Find sids of sailors who have reserved a red or green boat

```
SELECT    DISTINCT R.sid  
  
FROM      Boats B, Reserves R  
  
WHERE     R.bid = B.bid AND  
          B.color='red' AND B.color='green';
```

Is this correct???

- **QUERY**: Find sids of sailors who have reserved a red or green boat

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

- **UNION**: allows to compute the union of two union-compatible sets of tuples

```
SELECT    DISTINCT R.sid
FROM      Boats B, Reserves R
WHERE     R.bid = B.bid AND B.color='red'
UNION
SELECT    DISTINCT R.sid
FROM      Boats B, Reserves R
WHERE     R.bid = B.bid AND B.color='green';
```

- **QUERY:** Find sids of sailors who have reserved a red and a green boat.

INTERSECT:

- Can be used to compute the intersection of any two union-compatible sets of tuples

INTERSECT will find the overlapping tuples between the first and second queries.

SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid = B.bid **AND** B.color='red'

INTERSECT

SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid = B.bid **AND** B.color='green';

Nested Queries

- Can use SQL queries to aid the evaluation of another SQL query
- WHERE clause can itself contain an SQL query~
 - Actually, so can FROM and HAVING clauses.

```
SELECT    S.sid
FROM      Sailors S
WHERE      S.rating > (SELECT AVG(rating)
                        FROM Sailors);
```

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

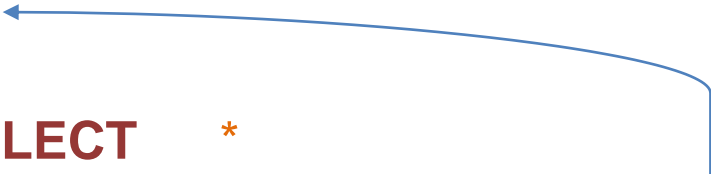
- Subqueries can also be relations with many tuples
- QUERY: Find Sailors who have not reserved boat #103

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


Nested Queries with Correlation

- **QUERY:** Find names of sailors who've reserved boat #103
- EXISTS is another set comparison operator, like IN.

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

A blue curved arrow originates from the `S.sid` in the `WHERE` clause of the inner query and points to the `S` in the `Sailors S` table reference in the outer query, indicating that the inner query is correlated with the outer query's table.

More on Set-Comparison Operators

- We've already seen IN, EXISTS and UNIQUE.
- Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: op ANY, op ALL, op IN
- QUERY: Find sailors whose rating is greater than that of some sailor called Horatio

```
SELECT *  
FROM Sailors S  
WHERE S.rating > ANY (  
    SELECT S2.rating  
    FROM Sailors S2  
    WHERE S2.sname='Horatio')
```

Rewriting INTERSECT Queries Using IN

- **QUERY:** Find sid's of sailors who've reserved both a red and a green boat

But why???

INTERSECT is not supported by all databases

```
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 S.sid IN (
            SELECT  S2.sid
            FROM    Sailors S2, Boats B2, Reserves R2
            WHERE   S2.sid=R2.sid AND R2.bid=B2.bid
                    AND B2.color='green')
```

- Similarly, EXCEPT queries can be re-written using NOT IN.

Aggregate Operators

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

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

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

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

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

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

- QUERY: Find name and age of the oldest sailor.

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

- Does the above query work???

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

GROUP BY and HAVING

SELECT [**DISTINCT**] target-list

FROM relation-list

WHERE qualification

GROUP BY grouping-list

HAVING group-qualification

- The target-list contains
 - (i) attribute names
 - (ii) terms with aggregate operations (e.g., MIN (S.age)).
- The attribute list (i) must be a subset of grouping-list.
- Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group. (A group is a set of tuples that have the same value for all attributes in grouping-list.)

Evaluations of GROUP BY

- 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.
- The group-qualification is then applied to eliminate some groups.
- Expressions in group-qualification must have a single value per group! In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping-list. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

- **QUERY:** 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
  
```

Sailors

sid	name	rating	age
22	Dustin	7	45
31	Lubber	8	55.5
64	Rusty	3	18
45	Chris	5	20
37	Brian	4	17
95	Bob	3	63.5



rating	age
7	45
8	55.5
3	18
5	20
4	17
3	63.5



rating	age
7	45
8	55.5
3	18
5	20
3	63.5

Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes 'unnecessary'.

2nd column of result is unnamed. (Use AS to name it.)



rating	age
3	18
3	63.5



rating	
3	18