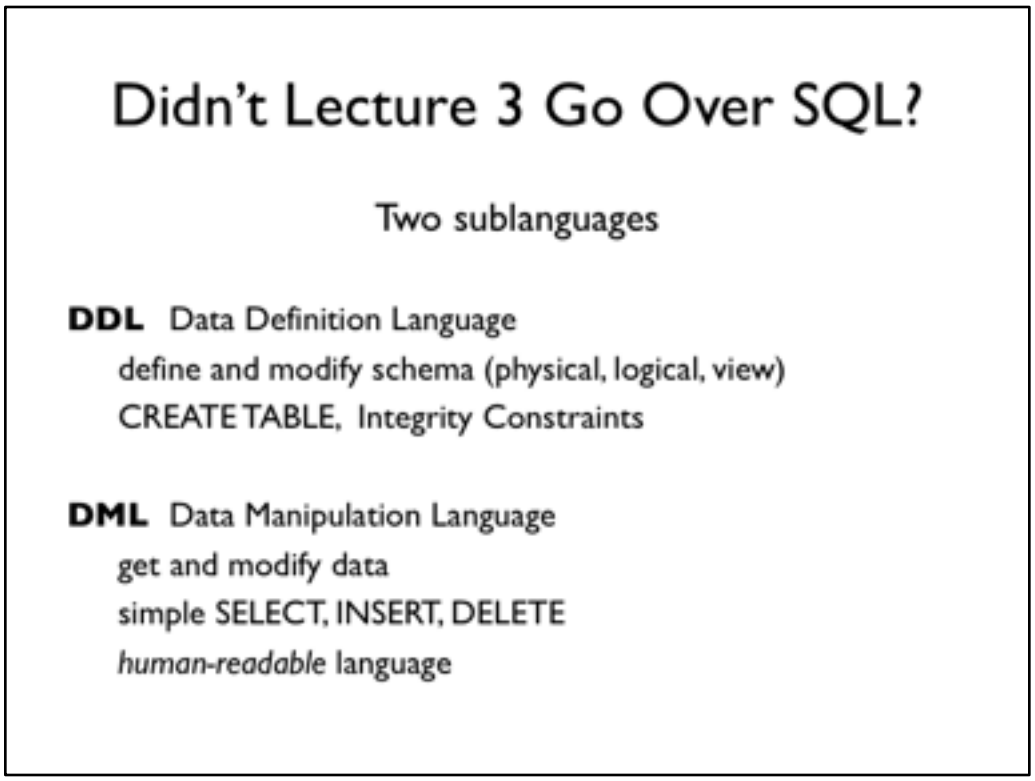
L5
SQL SQL SQL SQL SQL SQL SQL

Eugene Wu

Fall 2015



Yes, but only at a shallow amount of detail, we discussed ….
What did we discuss in lecture 3?

Unlike relaitonal algebra, where your keyboard doesn't even have the charactares for the operators

Gritty Details

DDL

NULL, Views

DML

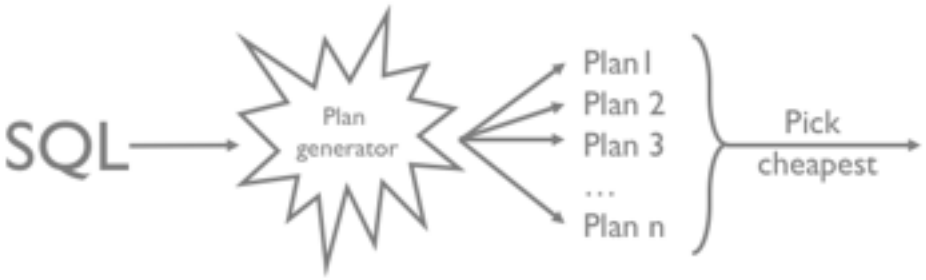
Basics, SQL Clauses, Expressions, Joins, Nested Queries, Aggregation, With, Triggers

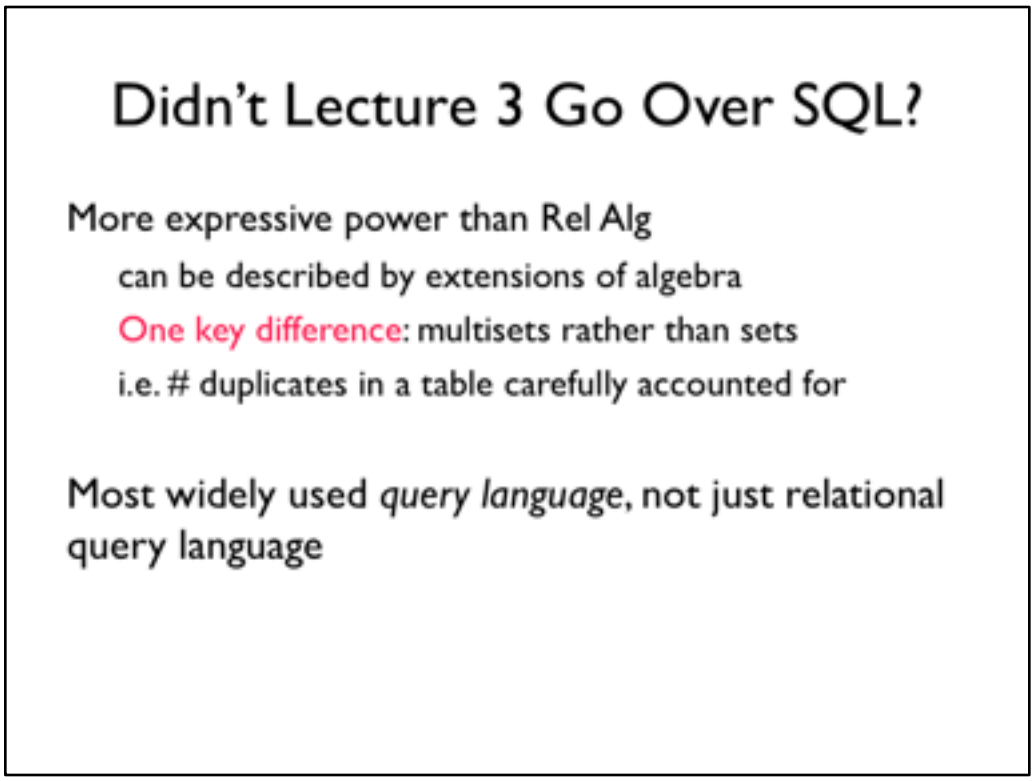
Today we wil go into gritty details about SQL

Didn't Lecture 3 Go Over SQL?

DBMS makes it run efficiently

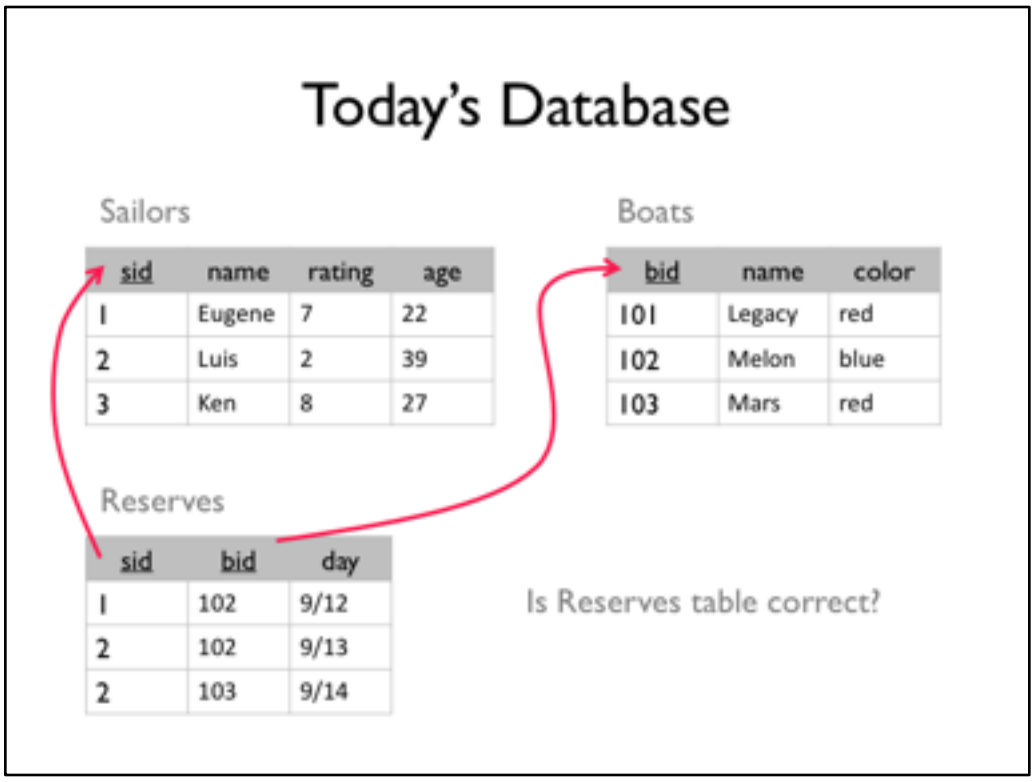
- Key: precise query semantics
- Reorder/modify queries while answers stay same
- DBMS estimates costs for different evaluation plans





Ironically, it was designed to be a very small language but over time the standard have been greatly extended, it's become extremely powerful, complicated.

Subtleties of nulls, multisets (bags), that make things tricky



Reserves has sid and bid as part of the primary key
Sailorl has sid as primary key
Boats uses bid as primary key

Today's Database

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Boats

bid	name	color
101	Legacy	red
102	Melon	blue
103	Mars	red

Reserves

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

Is Reserves table correct?
Day should be part of key

Follow along at home!

http://w4111db1.cloudapp.net:8000/

or
psql -U demo -h w4111db1.cloudapp.net demo
password: demo

<30 year old sailors

```
SELECT *
FROM Sailors
WHERE age < 30
```

sid	name	rating	age
1	Eugene	7	22
3	Ken	8	27

```
SELECT name, age
FROM Sailors
WHERE age < 30
```

name	age
Eugene	22
Ken	27

Cool, so this can be expressed in relaitonal algebra – how?

<30 year old sailors

```
SELECT *
FROM Sailors
WHERE age < 30
```

$\sigma_{age<30}$ (Sailors)

```
SELECT name, age
FROM Sailors
WHERE age < 30
```

$\pi_{name,\ age}$ ($\sigma_{age<30}$ (Sailors))

Multiple Relations

```
SELECT S.name
FROM   Sailors AS S, Reserves AS R
WHERE  S.sid = R.sid AND R.bid = 102
```

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

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

AS provides sailors with a variable name so we can reference it using a different name. We will see how this is useful for avoiding ambiguity

Which rows are matched in this join?

What kind of relational algebra join is this?

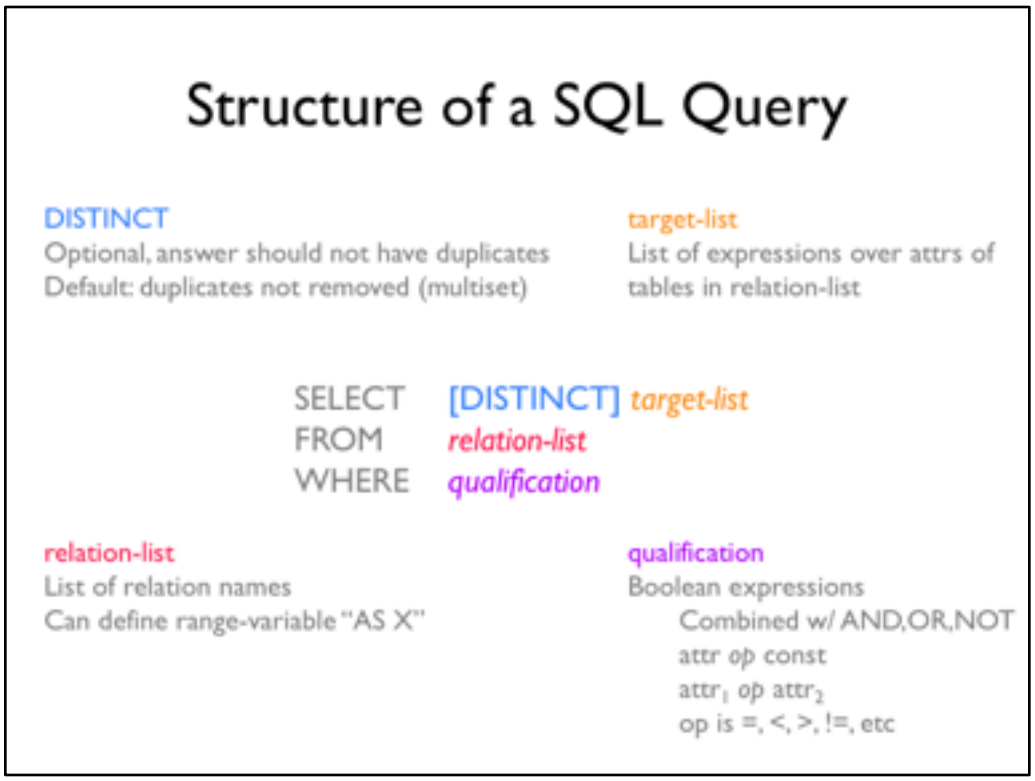
$\pi_{name}(\sigma_{bd=2}(\text{Sailors} \bowtie_{sid} \text{Reserves}))$

Multiple Relations

```
SELECT S.name
FROM   Sailors AS S, Reserves AS R
WHERE  S.sid = R.sid AND R.bid = 102
```

$$\pi_{name}(\sigma_{bid=2}(Sailors \bowtie_{sid} Reserves))$$

Which rows are matched in this join?
What kind of relational algebra join is this?



How that we have a feel of some queries, let's talk about the structure of SQL queries

There can be a bit of a confusion between the select operator in relational algebra, which picks out rows, and the SELECT clause is SQL, which picks out attributes in the result set. The best way is to remember than relational algebra and SQL are distinct languages!

Recall that SQL, unlite relational algebra has way more features. One of them is that instead of sets, it models relations as multisets or bags – basically it allows duplicates. To remove duplicates, we need to explicitly use the DISTINCT keyword

Expression may be a simple attribute reference, or any value expression – a function call, operator (e.g., and or, etc),

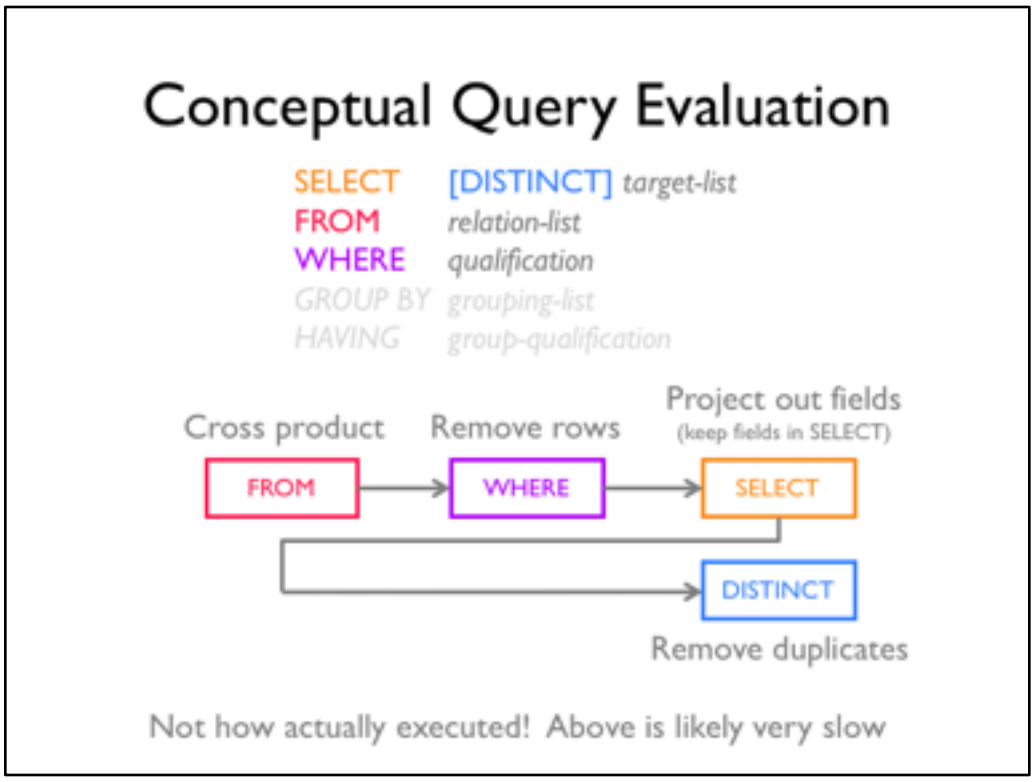
Semantics

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

FROM	compute cross product of relations
WHERE	remove tuples that fail qualifications
SELECT	remove fields not in target-list
DISTINCT	remove duplicate rows

Take the cross product of all possible results, then start dwindling it down

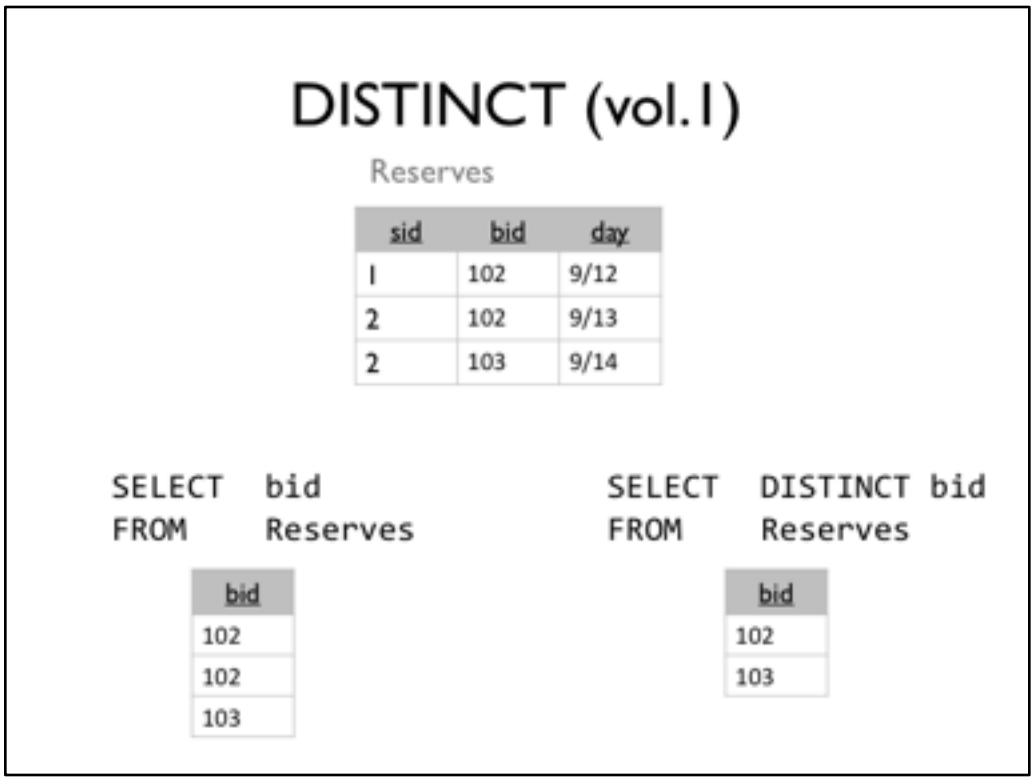
Note that WHERE is the same as select
SELECT is the same as project
(naming is really confusing)



To give you an alternative way of thinking about it, we can view this as a sequence of steps
You'll notice that there are two more greyed out clauses GROUP BY and HAVING, and we'll defer their description to later in this lecture

Helps to write it in this order (first figure out what data in the from, then remove things, then figure out what fields you want)

Prettyy lame way, DBMSes usually use join algorithms to speed things up



WHOA what gives!?

Because SQL != relational algebra!
So keep this multi-set business in mind, it'll show up in a lot of places

Sailors that reserved 1+ boats

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

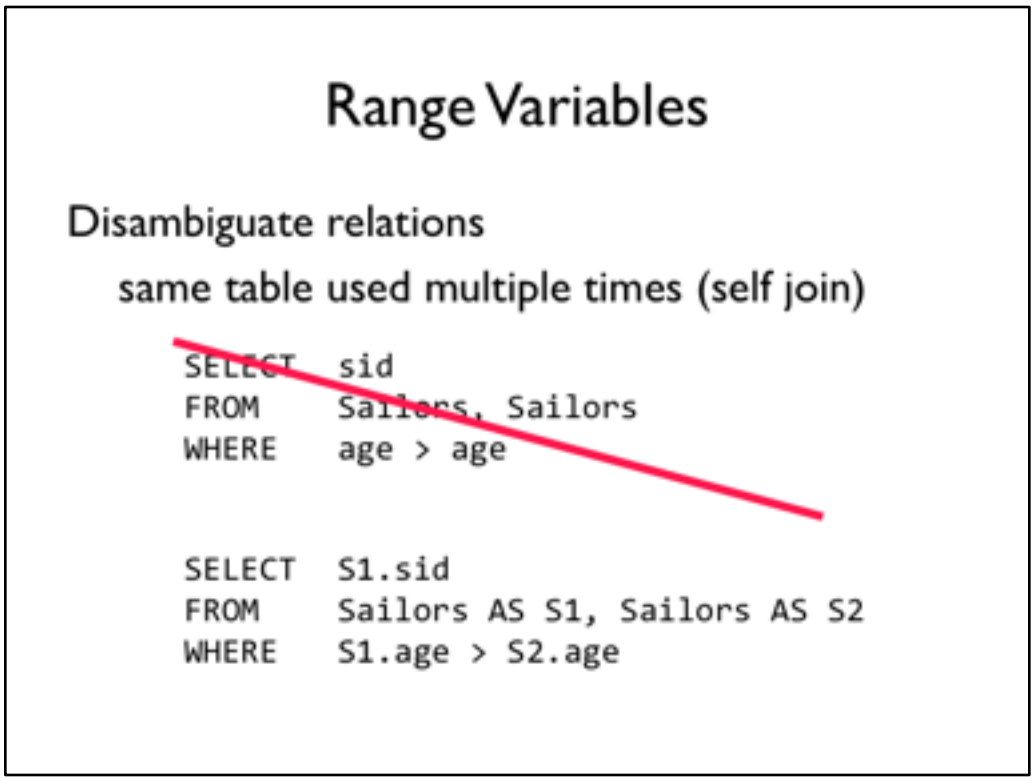
Would DISTINCT change anything in this query?

What if SELECT clause was SELECT S.name?

Like the conceptual query evaluation, we would star with the FROM, where want sailors and reservations
Which combinatons do we want? The ones where the sailor is the one that made the reservation
And finally, we just want the sid

Yes of course distinct will change things, because the output for SQl is a multiset.
Same thing. Of course!

Sid is a primary key so maybe it's no duplicates!
No, because of the join – the table with the foreign key dictates the number of duplicates and reservations has key over sid, bid, day!



A range variable is used to provide a name for a relation in the FROM clause

By default, the relation has a variable name the same as the relation, but if we use the same relation multiple times, then we would need to disambiguate them!
What does this query say?
Give me all sailfors whose age is greater than some other sailor. OR
give me all sailors whose age is less than some other sailor!
Which one!?

Range Variables

Disambiguate relations

same table used multiple times (self join)

```
SELECT  sid
FROM    Sailors, Sailors
WHERE   age > age
```

```
SELECT  S1.name, S1.age, S2.name, S2.age
FROM    Sailors AS S1, Sailors AS S2
WHERE   S1.age > S2.age
```

OK, so what does the query express? Let's project out some more data and see what happens in the database

Expressions (Math)

```
SELECT  S.age, S.age – 5 AS age2, 2*S.age AS age3
FROM    Sailors AS S
WHERE   S.name = ‘eugene’


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

AS in the SELECT clause give the resulting column a name, so that it could be referenced in a query on top of this result.

And ofcourse, arithmetic expressions are expressions, so they can be in the qualification list as well

Expressions (Strings)

```
SELECT  S.name
FROM    Sailors AS S
WHERE   S.name LIKE 'e_%'
```

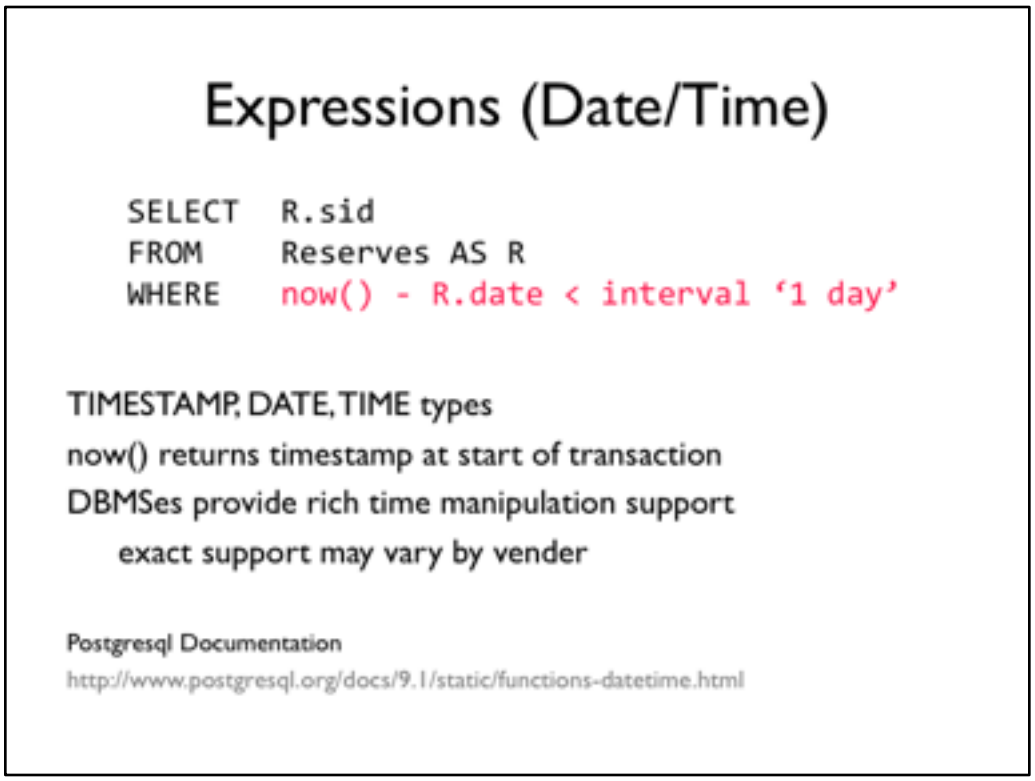
'_' any one character (. in regex)

'%' 0 or more characters of any kind (.* in regex)

Most DBMSes have rich string manipulation support e.g., regex

PostgreSQL documentation

http://www.postgresql.org/docs/9.1/static/functions-string.html



For example, we saw that reservations has a day attribute with type date.
Extract(day from day)
Extract(dow from day)

Expressions

Constant	1
Col reference	Sailors.name
Arithmetic	Sailors.sid * 10
Unary operators	NOT, EXISTS
Binary operators	AND, OR, IN
Function calls	abs(), sqrt(), ...
Casting	1.7::int, '10-12-2015'::date

sid of Sailors that reserved red or blue boat

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

OR

```
SELECT   R.sid
FROM     Boats B, Reserves R
WHERE    B.bid = R.bid AND B.color = 'red'
UNION ALL
SELECT   R.sid
FROM     Boats B, Reserves R
WHERE    B.bid = R.bid AND B.color = 'blue'
```

Recall union compatibility

Why UNION ALL? Because R.sid doesn't remove duplicates, and UNION by default does, so UNION ALL allows duplicates to exist

sid of Sailors that reserved red or blue boat

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

OR

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

Otherwise if we want only unique sids, we would use distinct or UNION. – be careful, this has bitten me many times

sid of Sailors that reserved red and blue boat

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

```
SELECT   R.sid
FROM     Boats B, Reserves R
WHERE    B.bid = R.bid AND B.color = 'red'
INTERSECT ALL
SELECT   R.sid
FROM     Boats B, Reserves R
WHERE    B.bid = R.bid AND B.color = 'blue'
```

sid of Sailors that reserved red and blue boat

Can use self-join instead

```
SELECT   R.sid
FROM     Boats B1, Reserves R1
WHERE
         B1.bid = R1.bid AND

         B1.color = 'red'
```

There's yet another way to implement it using a self join
Find sailors that render red boats,
Sailors that rented blue boats
And sailors whose ids are the same between the two groups of sailors

sid of Sailors that reserved red and blue boat

Can use self-join instead

```
SELECT   R.sid
FROM     Boats B1, Reserves R1, Boats B2,Reserves R2
WHERE
         B1.bid = R1.bid AND

         B1.color = 'red'
```

This says want to compute a cross product between reserved red boats with all boats and all reservations

sid of Sailors that reserved red and blue boat

Can use self-join instead

```
SELECT   R.sid
FROM     Boats B1, Reserves R1, Boats B2,Reserves R2
WHERE
         B1.bid = R1.bid AND
         B2.bid = R2.bid AND
         B1.color = 'red' AND B2.color = 'blue'
```

This says want all combinations of red reservations and all blue reservations

sid of Sailors that reserved red and blue boat

Can use self-join instead

```
SELECT   R.sid
FROM     Boats B1, Reserves R1, Boats B2,Reserves R2
WHERE    R1.sid = R2.sid AND
         B1.bid = R1.bid AND
         B2.bid = R2.bid AND
         B1.color = 'red' AND B2.color = 'blue'
```

THIS does the join between the b1r1s and the b2r2s

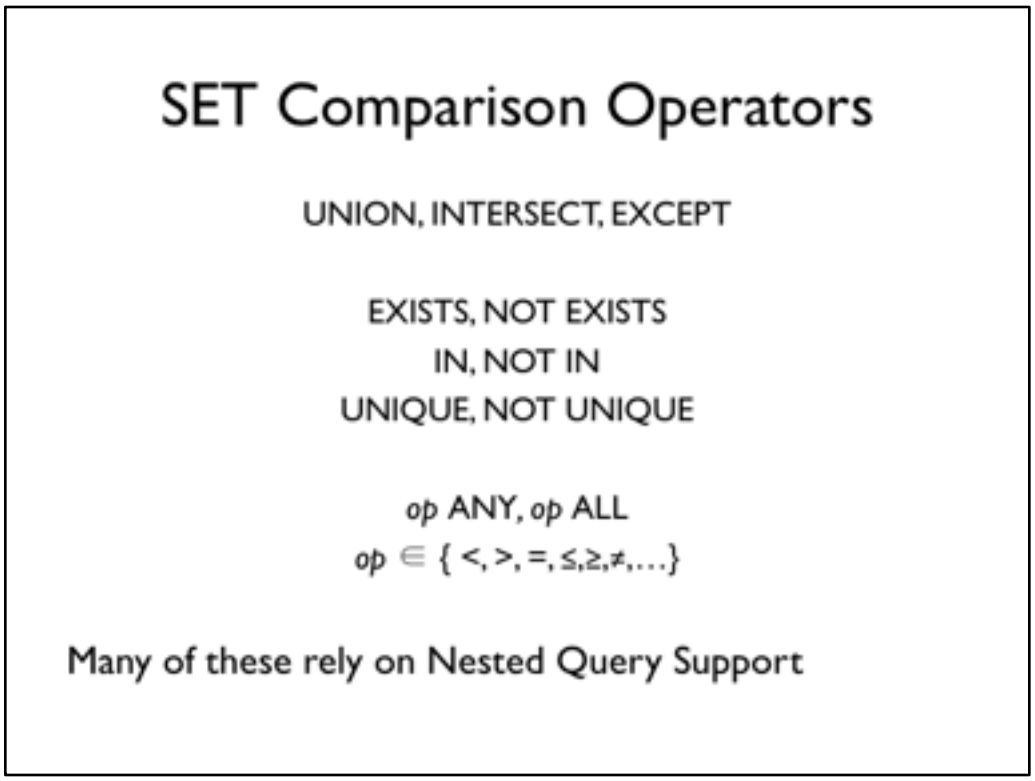
Notice how similar this is to the intersection! We computed two different sets of reservations, then combined them together. First with intersection (which only works because they are union compatible), and one using join

sids of sailors that haven't reserved a boat

```
SELECT   S.sid
FROM     Sailors S

EXCEPT

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

Can we write EXCEPT using more basic functionality?

EXCEPT is like UNION, it ignores duplicates and does set operator
EXCEPT ALL actually takes duplicates into account (multi-set cardinality) will them matter. Try it out!



What are nested queries? Queries within queries. Who says the WHERE clause has to only be ANDs, Ors, or simple arithemitec and logic statements?
You can also run sub-queries.

The key is that these nested queries return sets of records, so you can justcompare sailor.sid = nested query.

So SQL has some additional operators that let us operate on SETs. The ones we've already seen are union intersect and except.

However, there are others such as exists, not exists, in, unique …
In addition, there are the special key words that let us use normal comparison operators against sets. So single values (attr val or a constant) with sets.

Nested Queries

```
SELECT  S.sid
FROM    Sailors S
WHERE   S.sid IN (SELECT   R.sid
                  FROM     Reserves R
                  WHERE    R.bid = 101)
```

Many clauses can contain SQL queries
WHERE, FROM, HAVING, SELECT

Conceptual model:

```
for each Sailors tuple
   run the subquery and evaluate qualification
```

What if we replaced IN with NOT IN?
Logically very similar to JOIN, but not exactly the same. We'll see why…

How many duplicates?
Zero – because it's a fliter on the sailors table. It only returns at most once for each sailor
Duplicate semantics of this is different than for join, where can have mulitple return records for a given sailor

Although that's the conceptual model, we could precompute the subquery once and reuse the precomputed table

Nested Correlated Queries

```
SELECT  S.sid
FROM    Sailors S
WHERE   EXISTS (SELECT  *
                FROM    Reserves R
                WHERE   R.bid = 101 AND
                        S.sid = R.sid)
```

Outer table referenced in nested query

Conceptual model:

```
for each Sailors tuple
   run the subquery and evaluate qualification
```

This is a correlated query, meaning that this outer sailor S is being referenced in the nested query.

So, unlike the previous query, we CAN'T precompute the nested query and re-use its results for every sailor row. We actually have to run this queryfor every record. This is a terrible idea, and thankfully, query optimizers are usulally smart enough to rewrite this as a join -- however need to be careful to preserve the duplicate semantics.

Nested Correlated Queries

```
SELECT  S.sid
FROM    Sailors S
WHERE   UNIQUE (SELECT  *
                FROM    Reserves R
                WHERE   R.bid = 101 AND
                        S.sid = R.sid)
```

UNIQUE checks that there are no duplicates

What does this do?

This just returns every sailor, because each row in reserves will be unique
So this does nothing

Nested Correlated Queries

```
SELECT  S.sid
FROM    Sailors S
WHERE   UNIQUE (SELECT  R.sid
                FROM    Reserves R
                WHERE   R.bid = 101 AND
                        S.sid = R.sid)
```

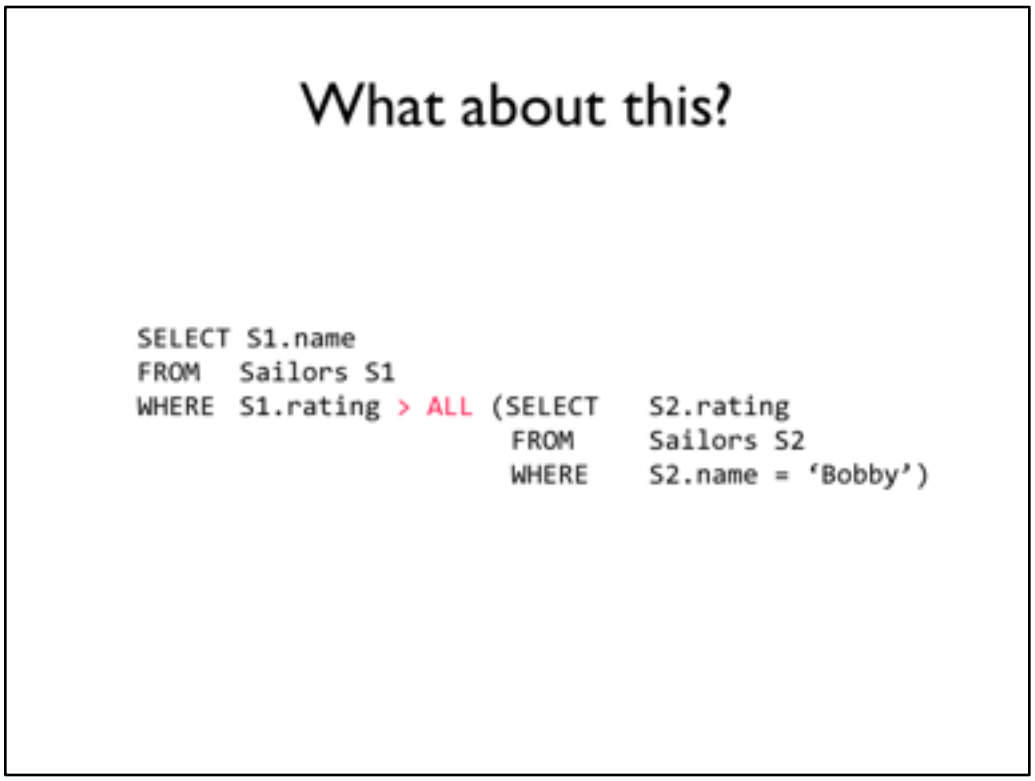
UNIQUE checks that there are no duplicates

What does this do?

So you need to be careful with sets – and pick s.sid
s.Sid is going to be unique, so it's just sailors than reserved boat 101

Sailors whose rating is greater than any sailor named "Bobby"

```
SELECT S1.name
FROM   Sailors S1
WHERE  S1.rating > ANY (SELECT   S2.rating
                        FROM     Sailors S2
                        WHERE    S2.name = 'Bobby')
```



What about this?

Rewrite INTERSECT using IN

```
SELECT S.sid
FROM   Sailors S
WHERE  S.rating > 2

INTERSECT

SELECT R.sid
FROM   Reserves R
```

```
SELECT S.sid
FROM   Sailors S
WHERE  S.rating > 2 AND
       S.sid IN (
    SELECT R.sid
    FROM   Reserves R
)
```

Similar trick for EXCEPT → NOT IN

What if want *names* instead of sids?

If want sailor names, then should we just replace all the sids with names?
NO. Be careful, because names is not a key, so

- On the left: need to have an outer query that translates the sids to names
- On the right, just change the SELECT clause to S.name

Sailors that reserved all boats (Division)

Hint: double negation

reserved all boats == no boat w/out reservation

```
SELECT    S.name
FROM      Sailors S
WHERE     NOT EXISTS (

   (SELECT   B.bid FROM   Boats B)

   EXCEPT

   (SELECT   R.bid
    FROM Reserves R
    WHERE R.sid = S.sid)
)
```

HW1 bugs

Conflicting CHECK constraints

```
Prof(
  type text,
  check(text in ('junior', 'senior')),
  check(text = 'junior' and hired is not null),
  check(text = 'senior' and tenure_year is not null)
```

conflicting

HW1 bugs

At most once *per semester* translated as at most once

```
CREATE TABLE Offers (
   deptid text,
   courseid text,
   semester text,
   year int,
   . . .
   PRIMARY KEY(deptid, courseid)
);
```

Wrong

HW1 bugs

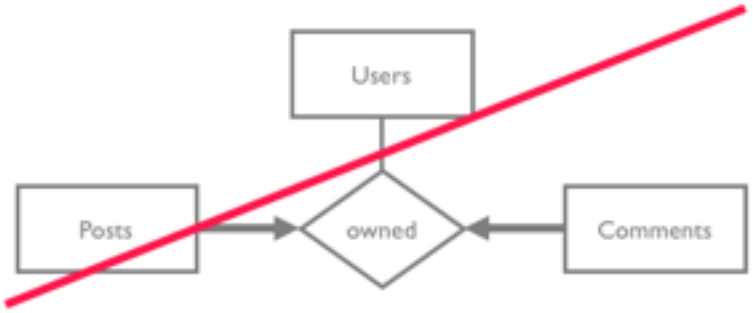
At most once *per semester* translated as at most once

```
CREATE TABLE Offers (
   deptid text,
   courseid text,
   semester text,
   year int,
   . . .
   PRIMARY KEY(deptid, courseid, semester, year)
);
```

HW1 bugs

Reddit:

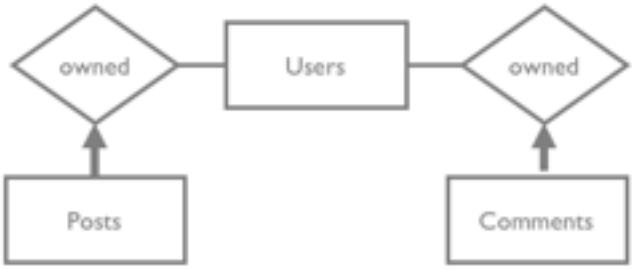
- Comments owned by one user
- Posts owned by one user



HW1 bugs

Reddit:

- Comments owned by one user
- Posts owned by one user



Probably also want to make posts and comments weak entities

Sailors that reserved all boats (Division)

Hint: double negation

reserved all boats == no boat w/out reservation

```
SELECT S.name
FROM   Sailors S
WHERE  NOT EXISTS (
```

Sailors S such that

There's no boat without

A reservation by S

Sailors that reserved all boats (Division)

Hint: double negation

reserved all boats == no boat w/out reservation

```
SELECT S.name
FROM   Sailors S
WHERE  NOT EXISTS (SELECTB.bid
                   FROM  Boats B
                   WHERE NOT EXISTS (
```

Sailors S such that

There's no boat without

A reservation by S

Sailors that reserved all boats (Division)

Hint: double negation

reserved all boats == no boat w/out reservation

```
SELECT S.name
FROM   Sailors S
WHERE  NOT EXISTS (SELECT B.bid
                   FROM  Boats B
                   WHERE NOT EXISTS (SELECT R.bid
                                     FROM Reserves R
                                     WHERE R.sid = S.sid))
```

Sailors S such that

There's no boat without

A reservation by S

NULL

Field values sometimes unknown or inapplicable
SQL provides a special value *null* for such situations.

The presence of null complicates many issues e.g.,
Is age = null true or false?
Is null = null true or false?
Is null = 8 OR 1 = 1 true or false?

Special syntax "IS NULL" and "IS NOT NULL"
3 Valued Logic (true, false, unknown)

How does WHERE remove rows?
if qualification doesn't evaluate to true
New operators (in particular, outer joins) possible/needed.

We're going to switch over to talking about joins, which are really just useful convenience functions over what we have talked about
But before, we need to understand NULLs

Equality is meaningless when talking about nulls, need special syntax
Logic also needs to be extended to deal with these

There are a number of ways it complicates/enriches SQL
For example, this entire time we've said that WHERE's job is to keep records where the qualification doesn't evaluate to TRUE
But what if it evaluates to null? So we need to make a decision about this.

We'll also find that new operators – in particular, outer joins – are useful now that we have a concept of nulls

Notic this is something that relational algebra didn't talk about

NULL

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

Some truth tables

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

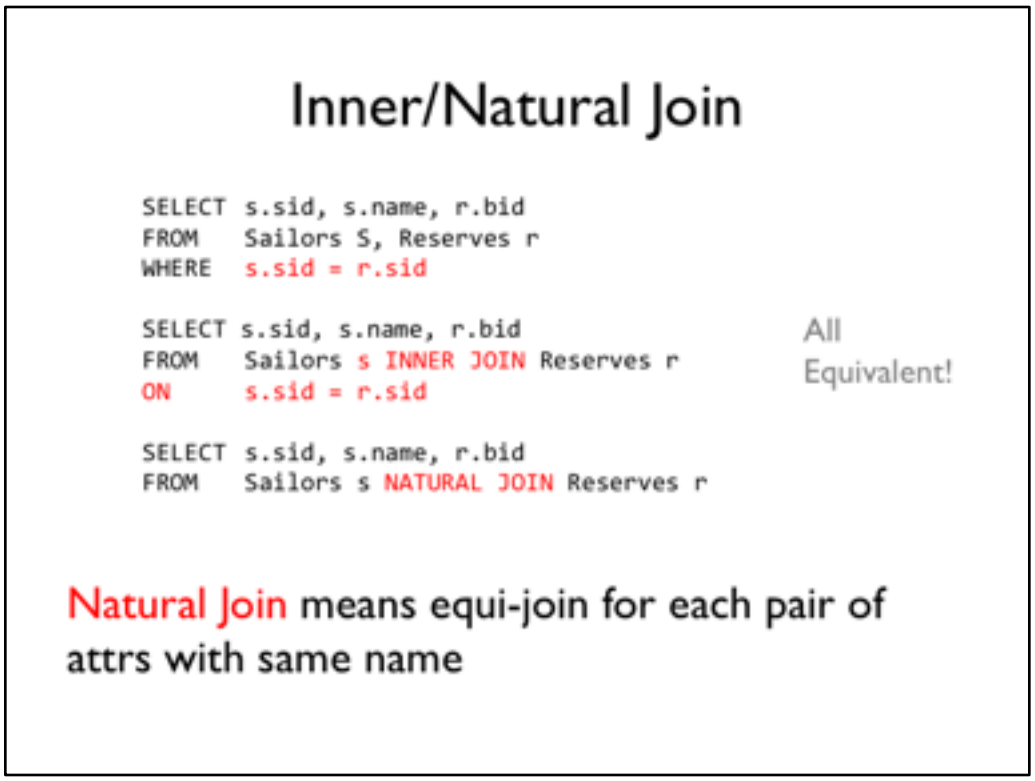
```
SELECT [DISTINCT] target_list
FROM table_name
    [INNER | {LEFT |RIGHT | FULL } {OUTER}] JOIN table_name
    ON qualification_list
WHERE ...
```

INNER is default

Difference in how to deal with NULL values

PostgreSQL documentation:
http://www.postgresql.org/docs/9.4/static/tutorial-join.html

By default, I mean that the way we have been doing joins so far, where the join condidion has been part of the WHERE clause, is equivalent to an inner join. also, note that inner join is like a THETA-join operator in rel alg



But natural join does something special, which is remove the duplicated sid if you use *

Often I use the first one, but it's a matter of preference.

Recall why natural join can be dangerous, so it's generally discouraged. Instead, it's a good idea to explicitly list the columns in the SELECT clause and only do SELECT * if you're debugging

Sailor names and their reserved boat ids

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

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

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

Result

sid	name	bid
1	Eugene	102
2	Luis	102

Sailor names and their reserved boat ids

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

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

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

Result

sid	name	bid
1	Eugene	102
2	Luis	102

Prefer INNER JOIN over NATURAL JOIN. Why?

Generally a good idea to use INNER JOIN instead of natural join
Why? (if someone renames an attribute, you'd rather have the query fail than give you strangely different results!)
For example, if I added a bid for birthday id to the sailors table, then NATURAL JOIN would give a different answer.

Failure is better than the wrong answer. A wrong answer is pretty much impossible to debug – you'll see the app givincg weird – WRONG – results and impossible to figure out why!

Sailor names and their reserved boat ids

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

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

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

Result

sid	name	bid
1	Eugene	102
2	Luis	102

Notice: No result for Ken!

There wasn't a reservation row with Ken's sid (due to join key)

What if we wanted all the sailors, and their reservation if possible, but otherwise just fill those attributes with null?

Left Outer Join (or No Results for Ken)

Returns all matched rows *and all unmatched rows from table on left of join clause*

(*at least one* row for each row in left table)

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

All sailors & bid for boat in their reservations

Bid set to NULL if no reservation

Left Outer Join

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

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

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

Result

sid	name	bid
1	Eugene	102
2	Luis	102
3	Ken	NULL

Cool, we now have a row for ken despite him never reserving any boats

Can Left Outer Join be expressed with Cross-Product?

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

sid	bid	day

```
Sailors x Reserves
```

```
Sailors s LEFT OUTER JOIN Reserves r
ON  s.sid = r.sid
```

Result

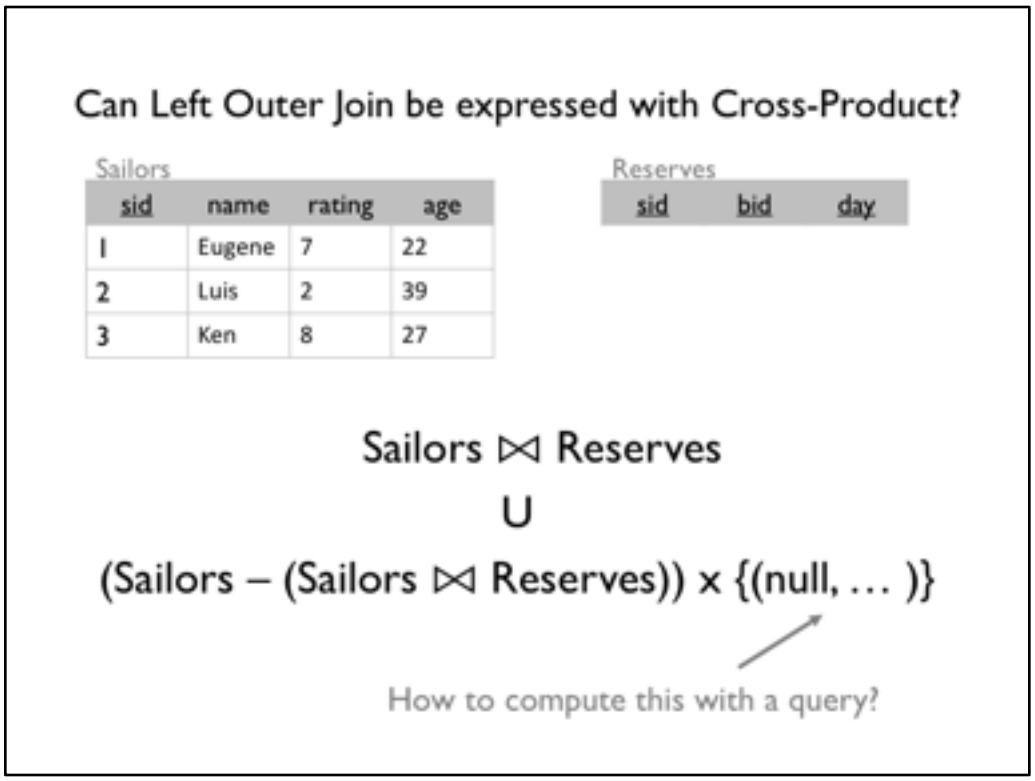
sid	name	bid

Result

sid	name	bid
1	Eugene	NULL
2	Luis	NULL
3	Ken	NULL

OK an important question. We've rewritten except, intersect etc using more fundamental operators.
What about left outer join? Is it a necessary operator, or is it a composite operator???

What's actually happening?
The cross product can only express inner join!
So we need to now fill in the records where sailors are not matched



There's something deeply unsatisfying about this query, can you guess?
So far we have rewritten queries into core operators over the base data but the null tuple is not base data!
The (null, …) needs to have the same number of attributes as the reserves table for everything to work out right? And the types must be compatible with the reserves table.
Other than manually writing down the same numberof nulls as attrs in reserves, is there a way to get this by writing a query?
-- I can't think of a way by manipulating data values.
So it requires a bit of cheating and knowning the metadata information

What does this mean!? You can manually rewrite the query string by working outside of direct data transformations to construct a left outer join.

First order logic basically allows programs to loop over and manipulate data (records)
Second order logic allows programs to iterate over and manipulate metadata (attr names, table names, database names)
Notice how SQL has not been able to dynamically construct a new table e.g., the attribute names and types of a table based on DATA – we've always done it manually.

Right Outer Join

Same as LEFT OUTER JOIN, but guarantees result for rows in table on right side of JOIN

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

FULL OUTER JOIN

Returns all matched *or* unmatched rows from both sides of JOIN

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

FULL OUTER JOIN

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

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

sid	bid	day
1	102	9/12
2	102	9/13
4	109	9/20

Result

sid	name	bid
1	Eugene	102
2	Luis	102
3	Ken	NULL
NULL	NULL	109

Why is sid NULL?

OK, reserves is incorrect, but let's ignore that for now.

Since reserves has an sid, why is the sid in the result null (because it's projected from the sailor's table!)

OK, is full outer join fundamental given left and right outer joins?
No, composite

Serious people can count: Aggregation

```
SELECT COUNT(*)
FROM   Sailors S

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

SELECT COUNT(DISTINCT S.name)
FROM   Sailors S
WHERE  S.name LIKE 'D%'

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

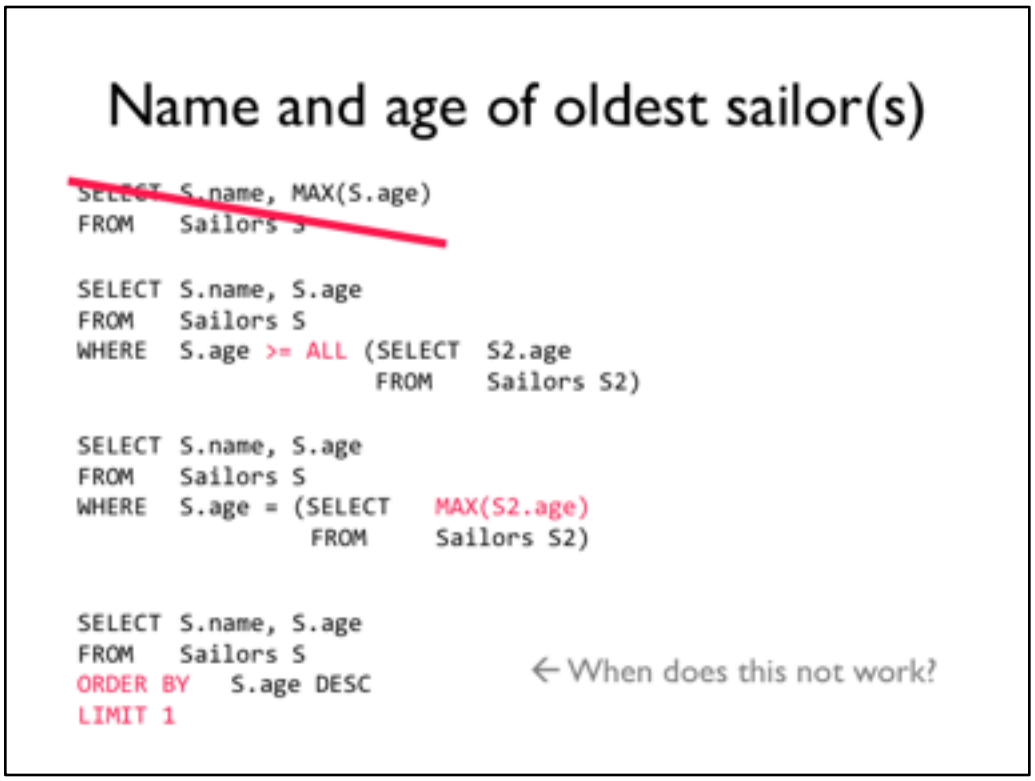
COUNT([DISTINCT] A)
SUM([DISTINCT] A)
AVG([DISTINCT] A)
MAX/MIN(A)
STDDEV(A)
CORR(A,B)

PostgreSQL documentation
http://www.postgresql.org/docs/9.4/static/functions-aggregate.html

Imagine you're the executive in the the Columbia boating agency that rents these boats.
You don't care about individual boats – you're too important for that
You want to know how the *business* is doing
You care about statistics!
How much money am I making, how many users? How many boats? What kinds of users?

How many sailors are using our service?
What's the average age of really good sailors?
I want to name my new baby and want a name starting with D because it's a strong letter.
What are the number of different names of sailors starting with D?
Names of sailors whose ratings are at the top?

I don't recommend using distinct for stddev and corr because that's USUALLY not statistically sound.



We might say, we'll we have the maximum value, so let's just add sailor name to the SELECT clause and call it a day.
This doesn't work because max in this query returns one value for the entire table, whereas S.name hase one value per record, so they are not compatible!

ORDER BY -- When there are ties for oldest

GROUP BY

```
SELECT min(s.age)
FROM   Sailors s
```

Minimum age among all sailors

What if want min age *per rating level*?

We don't even know how many rating levels exist!

If we did, could write (awkward):

```
for rating in [0…10]
   SELECT min(s.age)
   FROM   Sailors s
   WHERE  s.rating = <rating>
```

So in this case, we know that there are a fixed number of possible rating levels say 10, so we could enumerate them

GROUP BY

```
SELECT count(*)
FROM   Reserves R
```

Total number of reservations

What if want reservations per boat?

May not even know all our boats (depends on data)!

If we did, could write (awkward):

```
for boat in [0…10]
    SELECT count(*)
    FROM   Reserves R
    WHERE  R.bid = <boat>
```

But in this case, the number of boats is constantly changing – new boats are being purchased, other boats are being retired so it depends on the data and isn't per-determined!
In this case, we may not even know all our boats without asking the database, so enumerating the boats may be infeasible

But for each of these cases, although you might be able to loop through each of the possible boats or rating levels, you still need to manually kludge each of the query results together, and now you are doing it in the programming language rather than in the database – so you can't just directly query the result of this

GROUP BY

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

grouping-list is a list of expressions that defines groups

set of tuples w/ same value for all attributes in grouping-list

target-list contains

attribute-names ⊆ *grouping-list*

aggregation expressions

To compute statistics over groups we need two things
- Define the groups
- Define the statistics

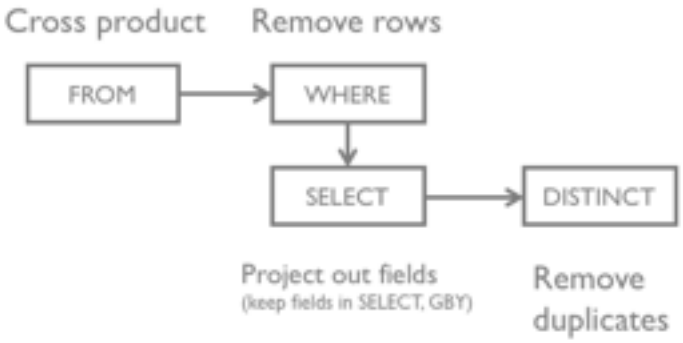
The group by clause defines the list of groups as a list of expressions. By evaluating these expressions on each record, it tells us what bucket to place this record. Once these buckets created, we can compute avgs, mins, maxs on the data in each bucket

HAVING lets us then remove groups that we don't care about – buckets where avg age is too high

When this happens, it restricts the typse of expressions we can have in the SELECT clause.
Data has been placed into buckets defined by the grouping list, so we can only talk abut these buckets, rather than individual records
- Either expressions in the grouping-list, which define the buckets
- Or statistics over the data in the buckets (aggregation expressions)

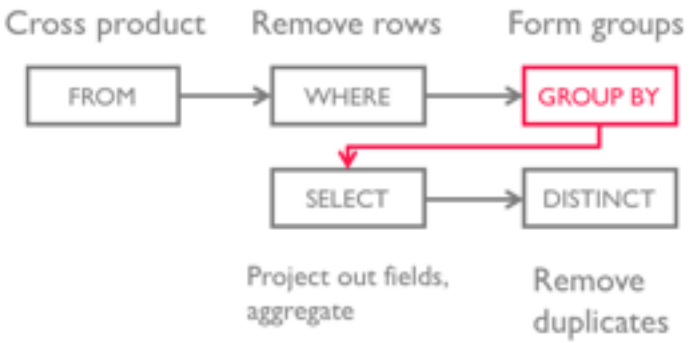
Conceptual Query Evaluation

SELECT [DISTINCT] *target-list*
FROM *relation-list*
WHERE *qualification*
GROUP BY *grouping-list*
HAVING *group-qualification*



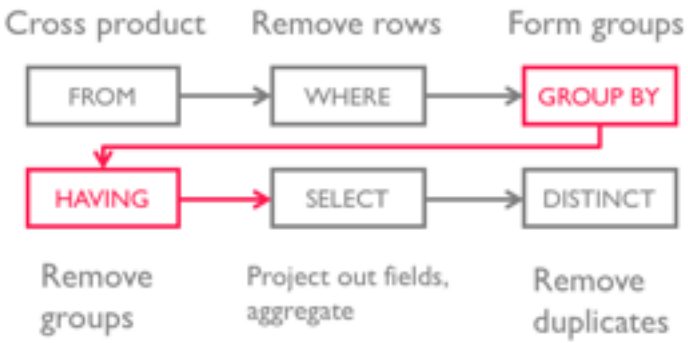
Conceptual Query Evaluation

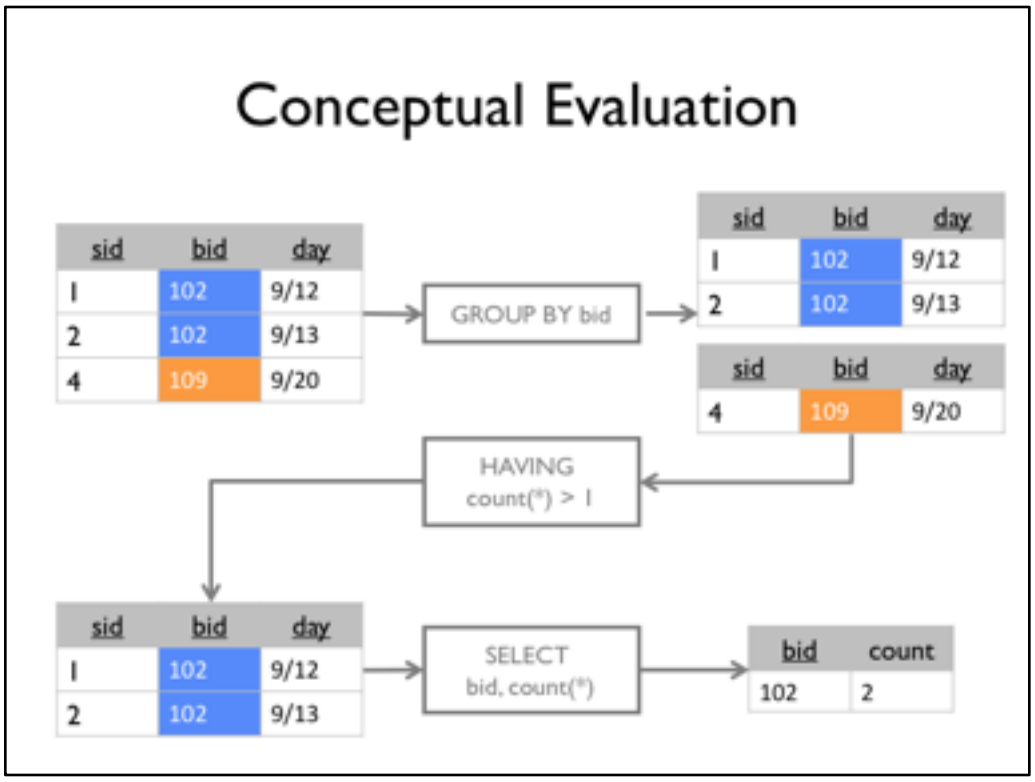
SELECT [DISTINCT] *target-list*
FROM *relation-list*
WHERE *qualification*
GROUP BY *grouping-list*
HAVING *group-qualification*



Conceptual Query Evaluation

SELECT [DISTINCT] *target-list*
FROM *relation-list*
WHERE *qualification*
GROUP BY *grouping-list*
HAVING *group-qualification*





let's say we want the boats reserved more than once and the number of times they've been reserved
What should we group by?
What should we use for having?

Remember that in the SELECT clause, we can ONLY use aggregation functions or grouping expressions (bid). Picking sid would not make sense since its values are different within the bucket.

GROUP BY

```
SELECT    bid, count(*)
FROM      Reserves R
GROUP BY  bid
```

Minimum age for each rating

```
SELECT    bid, count(*)
FROM      Reserves R
GROUP BY  bid
HAVING    count(*) > 1
```

Minimum age for each boat with more than 1 reservation

HAVING

group-qualification used to remove groups

similar to WHERE clause

Expressions must have *one value per group*. Either

An aggregation function or

In *grouping-list*

```
SELECT   bid, count(*)
FROM     Reserves R
GROUP BY bid
HAVING   color = 'red'
```

Because we are not grouping on color, and color is not an aggregation function ,s othere could be

AVG age of sailors reserving red boats, by rating

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

AVG age of sailors reserving red boats, by rating

```
SELECT    S.rating, avg(S.age) AS age
FROM      Sailors S, Boats B, Reserves R
WHERE     S.sid = R.sid AND
          R.bid = B.bid AND
          B.color = 'red'
GROUP BY  S.rating
```

What if move `B.color='red'` to HAVING clause?

You can't just move color to the having clause – needs to be in the groupby clause! In which case we would compute the count for each rating of each color, and only keep those with red boats. So the result would be similar.

However, you could also say, only give me the ratings where the average age is greater than 40. And that's something you can't do in the WHERE clause.

Ratings where the avg age is min over all ratings

```
SELECT S.rating
FROM    Sailors S
WHERE   S.age = (
            SELECT MIN(AVG(S2.age))
            FROM Sailors S2
        )
```

```
SELECT S.rating
FROM    (SELECT S.rating, AVG(S.age) as avgage
           FROM Sailors S
        GROUP BY S.rating) AS tmp
WHERE   tmp.avgage = (
    SELECT MIN(tmp2.avgage) FROM (
        SELECT  S.rating, AVG(S.age) as avgage
          FROM  Sailors S
      GROUP BY  S.rating
    ) AS tmp2
)
```

Q1: ratings of sailors whose age is the average – the min doesn't do anything

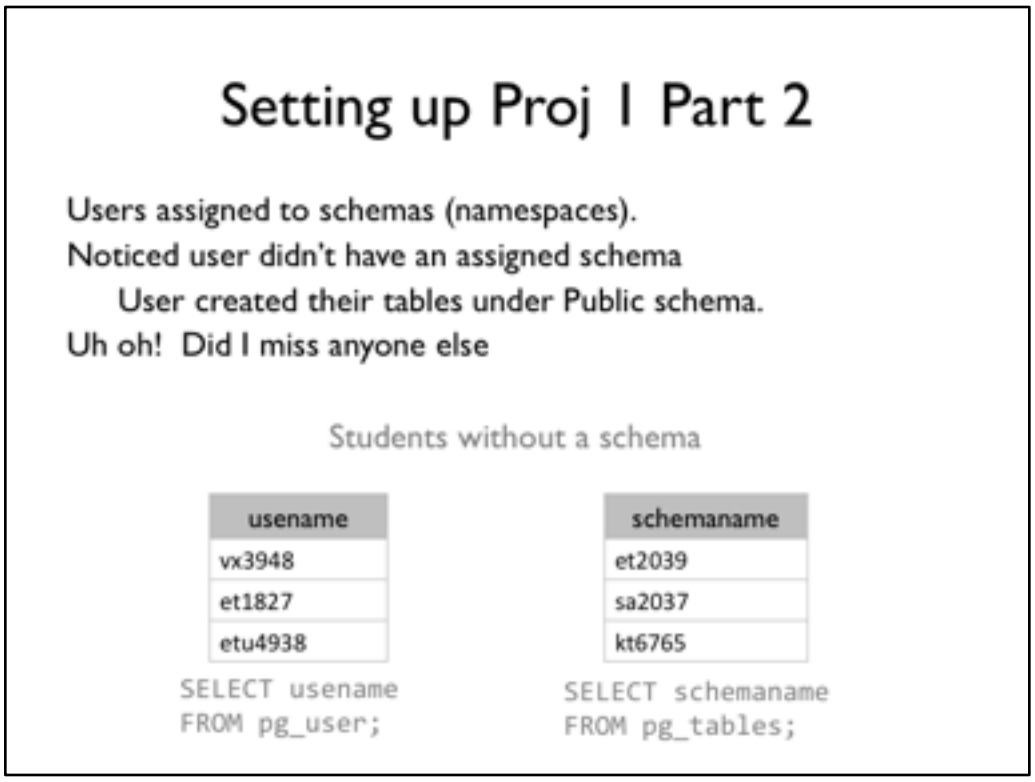
In the second case we're computing the avg age PERRATING, so it works.

Ratings where the avg age is min over all ratings

```
SELECT S.rating
FROM    Sailors S
WHERE   S.age = (
            SELECT MIN(AVG(S2.age))
            FROM Sailors S2
        )
```

```
SELECT S.rating
FROM    (SELECT S.rating, AVG(S.age) as avgage
           FROM Sailors S
        GROUP BY S.rating) AS tmp
WHERE   tmp.avgage <= ALL (
    SELECT tmp2.avgage FROM (
        SELECT  S.rating, AVG(S.age) as avgage
          FROM  Sailors S
      GROUP BY  S.rating
    ) AS tmp2
)
```

Q1: ratings of sailors whose age is the average – the min doesn't do anything



Pg_tables is a list of all the tables that have been created in the database, and along with the tables, it also tracks the schema that they belong to.
(show an output)

Pg_user tracks all users in the database, so there should be about 190 users.

Setting up Proj 1 Part 2

```
FROM    (SELECT usename FROM pg_user) AS U

        (SELECT schemaname FROM pg_tables) AS S
```

usename
vx3948
et1827
etu4938

```
SELECT usename
FROM pg_user;
```

schemaname
et2039
sa2037
kt6765

```
SELECT schemaname
FROM pg_tables;
```

Setting up Proj 1 Part 2

```
FROM    (SELECT usename FROM pg_user) AS U
        LEFT OUTER JOIN
        (SELECT schemaname FROM pg_tables) AS S
        ON U.usename = S.schemaname
```

usename
vx3948
et1827
etu4938

```
SELECT usename
FROM pg_user;
```

schemaname
et2039
sa2037
kt6765

```
SELECT schemaname
FROM pg_tables;
```

Setting up Proj 1 Part 2

```
SELECT U.usename, S.schemaname
FROM   (SELECT usename FROM pg_user) AS U
       LEFT OUTER JOIN
       (SELECT schemaname FROM pg_tables) AS S
       ON U.usename = S.schemaname
WHERE  S.schemaname is null;
```

usename
vx3948
et1827
etu4938

```
SELECT usename
FROM pg_user;
```

schemaname
et2039
sa2037
kt6765

```
SELECT schemaname
FROM pg_tables;
```

Setting up Proj 1 Part 2

```
SELECT count(*)
FROM (
    SELECT U.usename, S.schemaname
    FROM   (SELECT usename FROM pg_user) AS U
           LEFT OUTER JOIN
           (SELECT schemaname FROM pg_tables) AS S
           ON U.usename = S.schemaname
    WHERE  S.schemaname is null;
) AS nestedquery
```

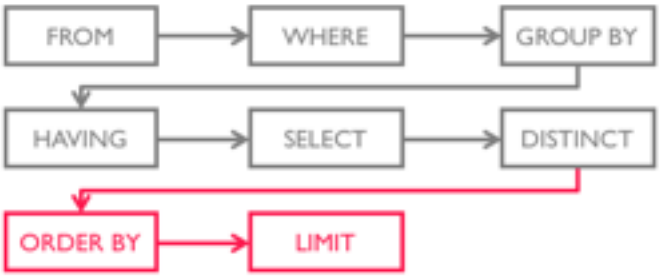
124

but 190 users!

190 – 124 ~= 66 * 2 ~= 132

ORDER BY, LIMIT

```
SELECT    [DISTINCT] target-list
FROM      relation-list
WHERE     qualification
GROUP BY  grouping-list
HAVING    group-qualification
ORDER BY  order-list
LIMIT     limit-expr [OFFSET offset-expr]
```



OK we are going over topics that confused me as well, so ask tons of questions. I will be out of town immediately before the exam, so it is in your interest to ask questions early.

ORDER BY

```
SELECT    S.name
FROM      Sailors S
ORDER BY  (S.rating/2)::int ASC,
          S.age DESC
```

List of *order-list* expressions dictates ordering precedence

- Sorted in ascending by age/rating ratio
- If ties, sorted high to low rating

ASC for ascending, meaning the smallest item is first, then the second smallest. DESC is the opposite

ties are arbitrarily decided, or by the next expression in the order-list

ORDER BY

```
SELECT    S.name, (S.rating/2)::int, S.age
FROM      Sailors S
ORDER BY  (S.rating/2)::int ASC,
          S.age DESC
```

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

name	int4	age
Luis	1	39
Ken	4	27
Eugene	4	22

First sorted by the middle column, so it's 1, 4, 4.

But since 4 and 4 are the same value, the second expression inth e order by says to order them by age

(otherwise if no S.age DESC, ken and eugene could be in arbitrary order – since not specified)

ORDER BY

```
SELECT    S.name, (S.rating/2)::int, S.age
FROM      Sailors S
ORDER BY  (S.rating/2)::int ASC,
          S.age ASC
```

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

name	int4	age
Luis	1	39
Eugene	4	22
Ken	4	27

What about ASC?

LIMIT

```
SELECT    S.name, (S.rating/2)::int, S.age
FROM      Sailors S
ORDER BY  (S.rating/2)::int ASC,
          S.age DESC
LIMIT     2
```

Only the first 2 results

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

name	int4	age
Luis	1	39
Ken	4	27

This is an example where the second expression in the order by matters. If it were Asc, then Eugene would be in the result, but in this case, tis' desc by age, so ken is returned as the second element.

LIMIT is really common for example on google search results, they only show the first 15 or 20 search results because technically the entire web is the search result (many sites have raking scores of zero), but you only care about the top ones.
Any time you're showing a list of the data in your database, you'll want to either summarize the database, or show the top number of items.

LIMIT

```
SELECT    S.name, (S.rating/2)::int, S.age
FROM      Sailors S
ORDER BY  (S.rating/2)::int ASC,
          S.age DESC
LIMIT     2 OFFSET 1
```

Only the first 2 results

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

name	int4	age
Ken	4	27
Eugene	4	22

So what happens when you go to the second page? You can use the offset to say how mayn items you want to skip

LIMIT

```
SELECT    S.name, (S.rating/2)::int, S.age
FROM      Sailors S
ORDER BY  (S.rating/2)::int ASC,
          S.age DESC
LIMIT     (SELECT count(S2.*) / 2
           FROM Sailors AS S2)
```

Can have expressions instead of constants

Result

name	int4	age
Luis	1	39

3 / 2 = 1.5 rounded down is 1 so there is a single result.

Integrity Constraints

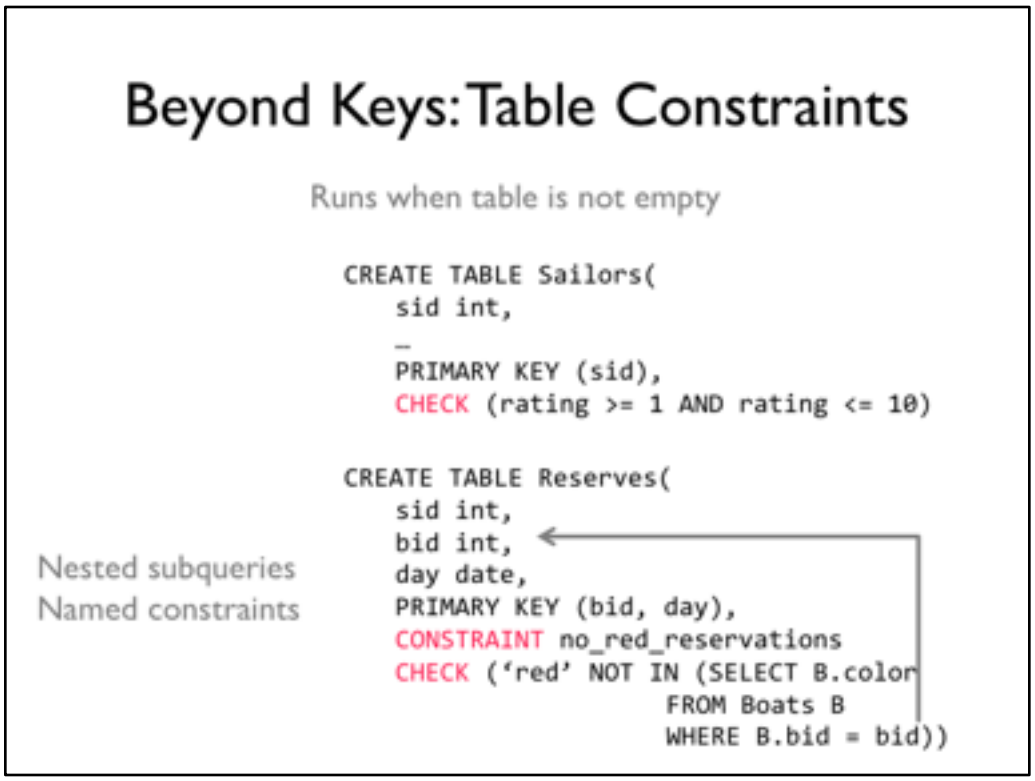
Conditions that every legal instance must satisfy

- Inserts/Deletes/Updates that violate ICs rejected
- Helps ensure app semantics or prevent inconsistencies

We've discussed

- domain/type constraints, primary/foreign key
- general constraints ←

Any time make change to database – I/D/U
It's a way for the programmer to tell the database some info about the application logic and expectations about the data so it can be automatically enforced.



We've already seen and used simple per attribute table constraints, these are ones that are bound to a table, and fire whenever the table is not empty.

But you can have more general table constraints that are
1) Named
2) Run nested queries

Cant reserve red boats

Subqueries are not supported in postgres check constraints

Multi-Relation Constraints

of sailors + # of boats should be less than 100

```
CREATE TABLE Sailors (
    sid int,
    bid int,
    day date,
    PRIMARY KEY (bid, day),
    CHECK (
        (SELECT COUNT(S.sid) FROM Sailors S)
        +
        (SELECT COUNT(B.bid) FROM Boats B)
        < 100
    )
```

What if Sailors is empty?
Only runs if Sailors has rows (ignores Boats)

That was an example of multi-relation constraints that run queries on another table. For example, this ensures that the total number of sailors and boats is less than 100 It's an unnamed CHECK constraint

If sailors empty, the CHECK will never run because check constraints are for tables that are non-empty

and can have arbitrary number of boats in our club.
Real estate is expensive, so this is going to ruin us

ASSERTIONS: Multi-Relation Constraints

```
CREATE ASSERTION small_club
CHECK (
    (SELECT COUNT(*) FROM Sailors S)
    +
    (SELECT COUNT(*) FROM Boats B)
    < 100
)
```

ASSERTIONs are not associated with any table

What's the whole point of Ics?
Assertions and constraints are a form of safety – when can incorrect things happen (wrt user intention)
Think of all possible databases for a given schema without constraints (types, relationships, etc)
My database could be ANYTHING!

This is the point of most languages – to make it easier to do certain types of tasks – scripting, multi-threaded programming etc – while making it hard to shoot yourself in the foot.
The way you do that is reduce the set of allowable BAD programs that can exist. In this case the behavior is the data in the database

Likelihood that errors happen reduces AND
The overhead of checking that everything is OK is removed

Think of the police. Without an authority for public safety, I may need to keep a body guard around me at all times, and check over my shoulder because the environment is not safe. That's both expensie because I have to pay for the bodyguard, and really ineficient because I spend half of my brain cycles looking around to make sure I won't be robbed.

But by delegating certain declarative goals to the police (or DBMS) such as keep robberies low, hurting people is not good, then I can focus on my own stuff.

Note that unfortunately, posgresql doesn't support any of this stuff beyond simple non-nested query check constraints.
Regardless, good to know

WHAT!

So many things we can't express or don't work!

Assertions

Nested queries in CHECK constraints

Let's talk about a couple language features that can help us with SOME small versions of these.

We can solve the first two, and partial setup for solving the at-least-one problem

Advanced Stuff

User defined functions

Triggers

WITH

Views

User Defined Functions (UDFs)

Custom functions that can be called in database

Many languages: SQL, python, C, perl, etc

```
CREATE FUNCTION function_name(p1 type, p2 type, …)
RETURNS type
```

What if I want to run my own functions?
What if abs() sucks and I want my own version?
What if I repeat the same operations and want to abstract them into a func?

The types can either be primitive types that we have seen,
Or abstract data types that we may cover later
Or records – in whiche case the type is the name of the relation (since that defines the structure of the row!)
Or ever sets of rows (relations)!!

We'll look at simple cases of returning scalar values and triggers
The psotgresql documentation is thorough and goes into excruciating detail

User Defined Functions (UDFs)

Custom functions that can be called in database

Many languages: SQL, python, C, perl, etc

```
CREATE FUNCTION function_name(p1 type, p2 type, …)
RETURNS type
AS $$

-- logic

$$ LANGUAGE language_name;
```

After the signature, we define th the actual logic. In this case, we need to define the actual code (logic) and also tell the dbms what language the logic is written in!

User Defined Functions (UDFs)

Custom functions that can be called in database

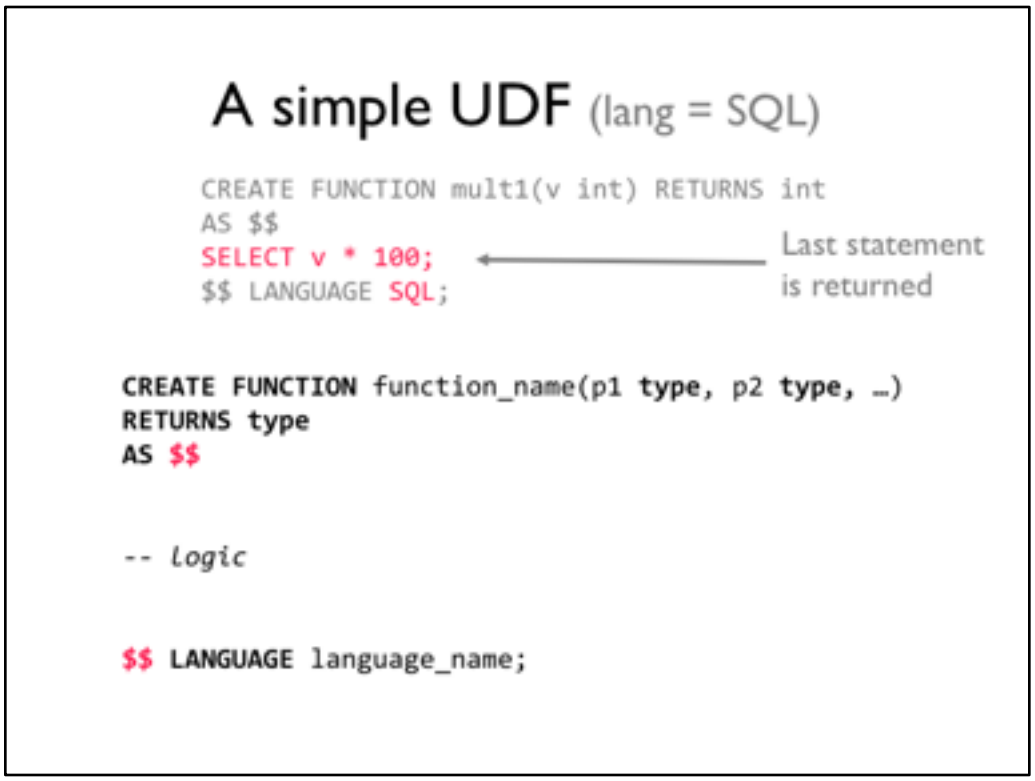
Many languages: SQL, python, C, perl, etc

```
CREATE FUNCTION function_name(p1 type, p2 type, …)
RETURNS type
AS $$

-- logic

$$ LANGUAGE language_name;
```

The $$ are just special strings that are not in the program to distinguish when the code starts and stops. For example, if we used parenthesizes, and we used a close paren in the code, then how does postgresql know to stop?

Remember that UDFs support many different langages so it needs to be flexible!

Almost all modern databases have powerful UDFS supoort.
We will see three example languages with which you can write UDFs. And project 2 will rquire some of this.



Here's a simple example using the SQL language. Basically a list of SQL statements, here the last one is returned.
So we can write crazy selcet statements to return, or insert delet and otherwise manipulate the database in this UDF

A simple UDF (lang = SQL)

```
CREATE FUNCTION mult1(v int) RETURNS int
AS $$
SELECT v * 100;
$$ LANGUAGE SQL;

SELECT mult1(S.age)
FROM   sailors AS S
```

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

int4
220
390
270

http://www.postgresql.org/docs/9.1/static/xfunc-sql.html

A simple UDF (lang = SQL)

```
CREATE FUNCTION mult1(v int) RETURNS int
AS $$
SELECT $1 * 100;
$$ LANGUAGE SQL;

SELECT mult1(S.age)
FROM   sailors AS S
```

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

int4
220
390
270

http://www.postgresql.org/docs/9.1/static/xfunc-sql.html

Instead of the keyword arguments, you can also use positional arguments

Process a Record (lang = SQL)

```
CREATE FUNCTION mult2(x sailors) RETURNS int
AS $$
SELECT (x.sid + x.age) / x.rating;
$$ LANGUAGE SQL;

SELECT mult2(S.*)
FROM   sailors AS S
```

Sailors

sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

int4
3.285
20.5
3.75

http://www.postgresql.org/docs/9.1/static/xfunc-sql.html

Here we gave the first argument the variable name x

Process a Record (lang = SQL)

```
CREATE FUNCTION mult2(sailors) RETURNS int
AS $$
SELECT ($1.sid + $1.age) / $1.rating;
$$ LANGUAGE SQL;

SELECT mult2(S.*)
FROM   sailors AS S
```

Sailors

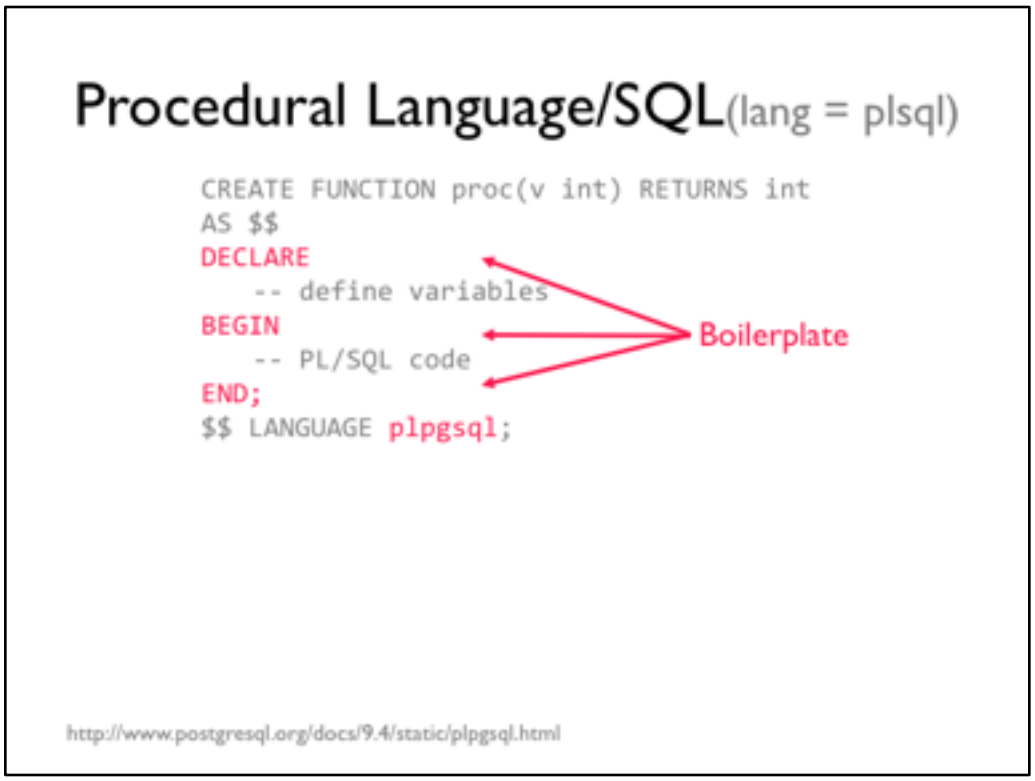
sid	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

int4
3.285
20.5
3.75

http://www.postgresql.org/docs/9.1/static/xfunc-sql.html

x – need to make sure it doesn't collide with any existing table names!
So usually easiest and most reliable to simply say the first arg's type is sailors and use positional references

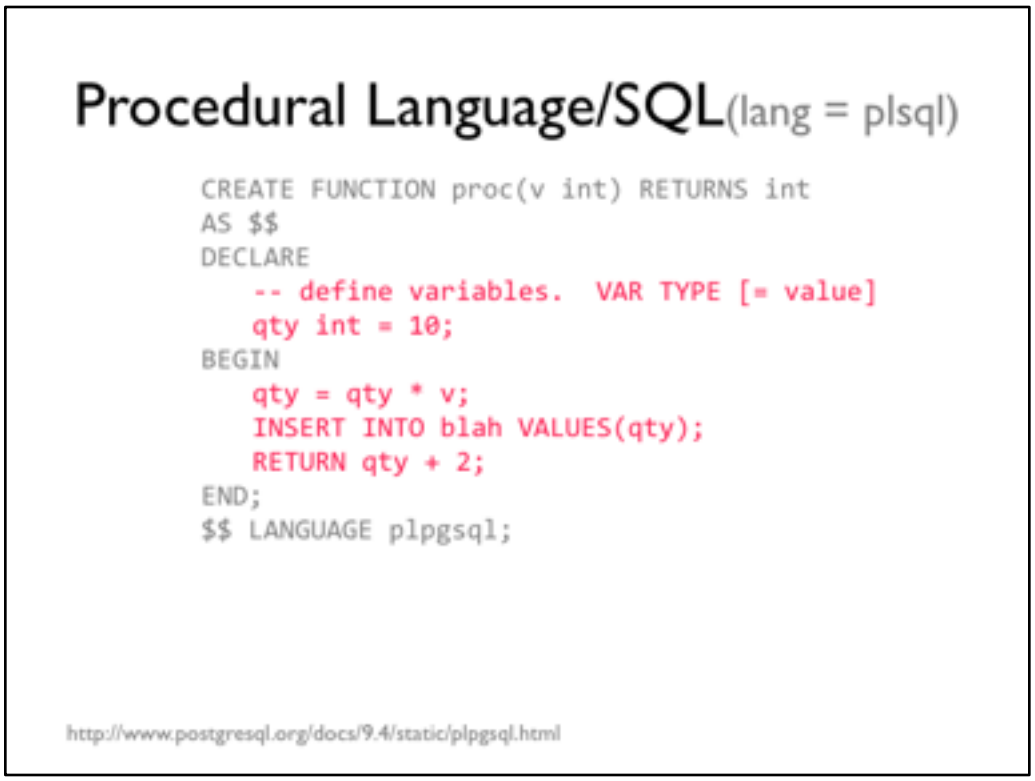


A lot of people complain – writing PROGRAMS in SQL is a pain because we can't seem to do if else statements, exceptions, other logic very easily.
PLSQL is an extension to SQL that adds procedures – procedural language SQL
Really complicated, talk about some basic features

There are two regions where you write code – declare variables, and write actual logic.
Unlike SQL, you need explicit return statements.

Note that the entire DECLARE/BEGIN/END block can be thought of as a single statement, so there's only one semicolon at the end of the entire thing.
Also, BEGIN/END is just syntax, it has NO RELATION to begin/commit/abort of transactions

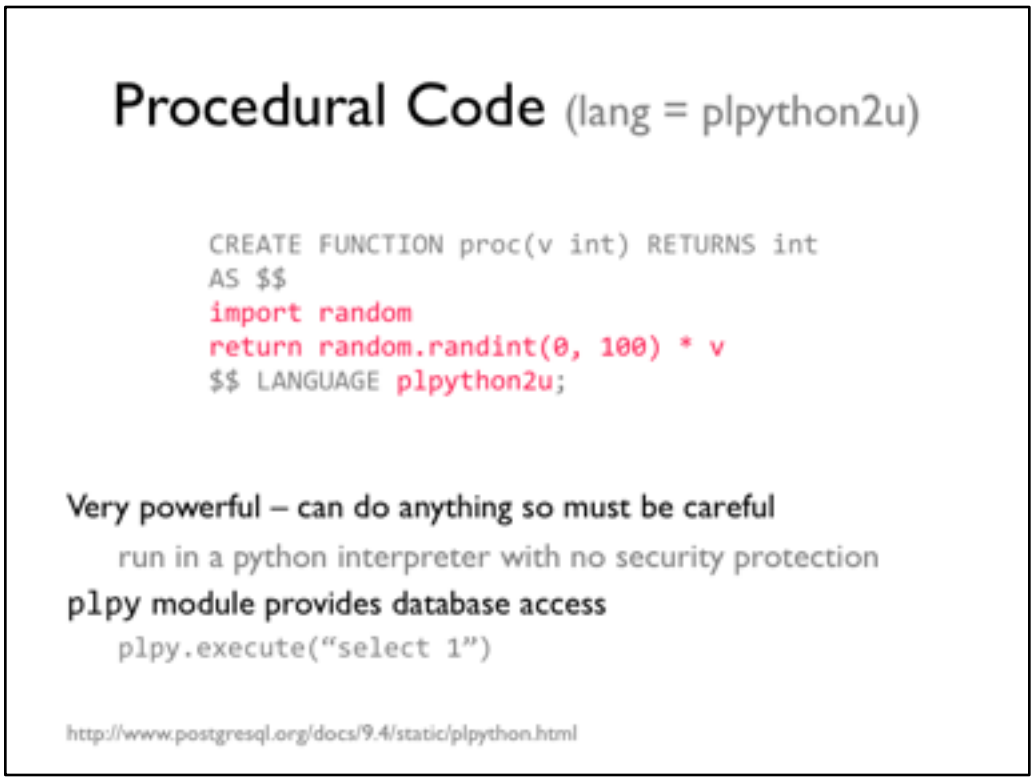
Plpgsql is pl with postgresql (pg) variant of sql



This program first defines a variable (variable type an optionally set the value)

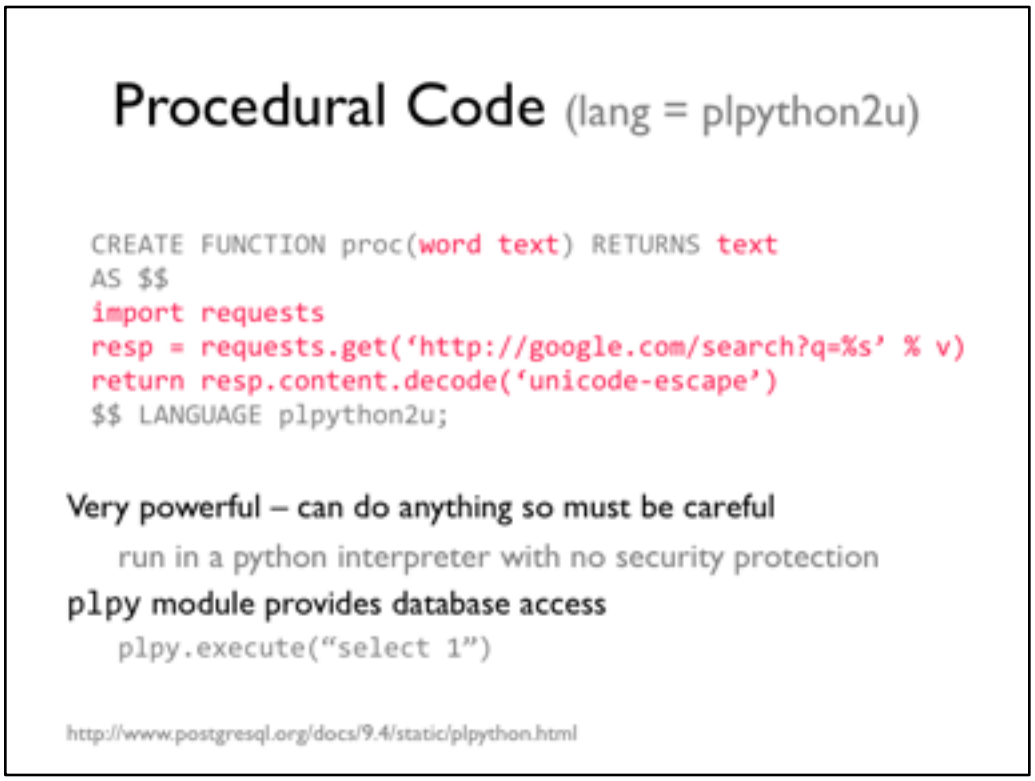
Statements end with semicolons

The BEGIN block can contain both SQL as well as conditionals and other procedural code. We will see more examples of this

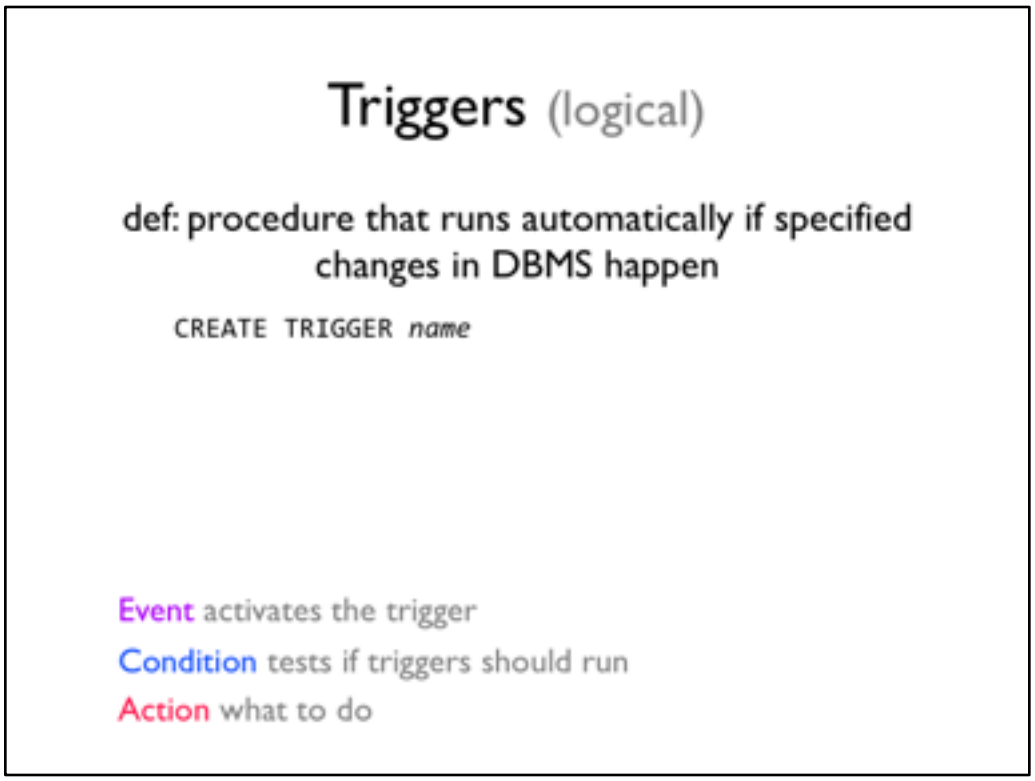


The 2 is for python 2
The u is for untrusted – you can do pretty much ANYTHING – it just runs a python interpretor!

For example, you can import modules such as random
And you can call functions.
You can also use a special python module called plpy that lets you access the database to ask about the schemas, to execute queries, etc



As an example of it's power, you could imagine writing a UDF that takes as input text (such as the sailor name),
and returns the content of the research results page from querying google.

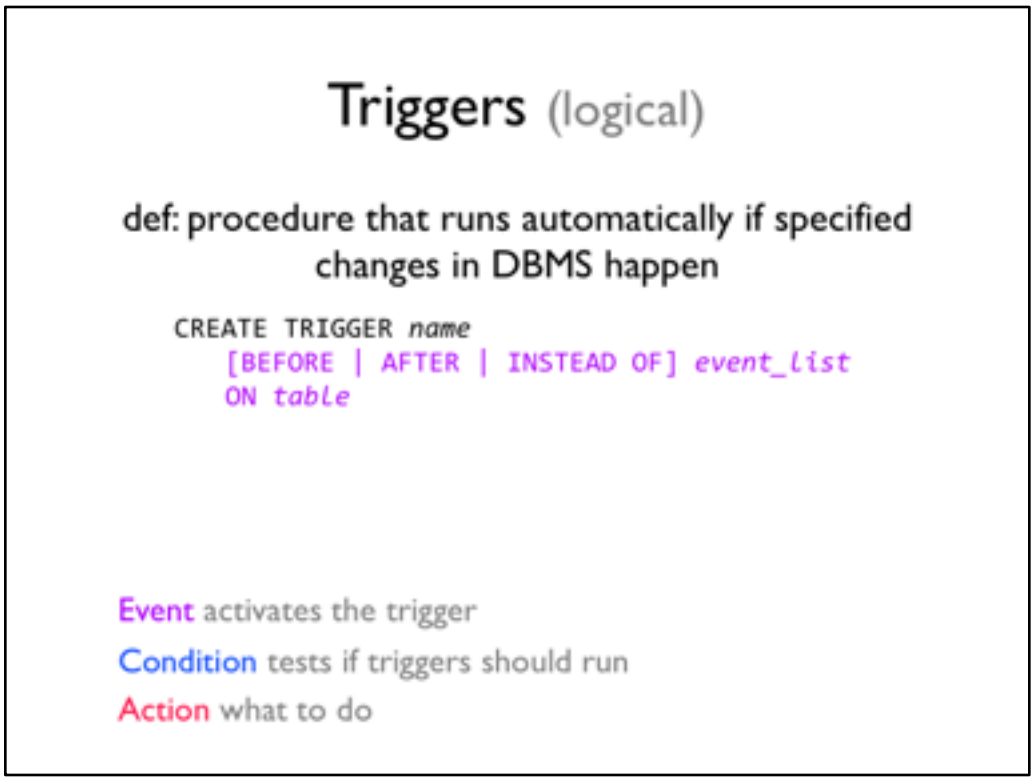


I want to emphasize that this is logical
We won't go into excruciating details, but I will discuss the logical ideas behind triggers, and give you the basic of how to implement them in postgresql

The text book for example, focuses on the main properties of triggers which are very important to know. And doesn't actually show you how to run a trigger in practice.

But let's step back, what are properties we would like to have from triggers that run procedures on changes to a DBMS. If you were to design this, what would you want? Or what things would you like to be able to do with triggers?

Common one is implement assertions
Another is to implement constraints
Or to fetch or populate tables based on data from users for example, user inputs the full address and you parse and clean it up for the normalized address, the state, city, etc and make sure they're correct.



Event: change to DB that activates a trigger
An event is for example, when I insert a record into a table, or when I delet a record, or when I change the value of a record.

If before, you can also preempt and avoid running the event, or replace the statement
Instead of means, whenever this event like an insert happens, replace it with this procedure.
If it's after, then clearly the procedure runs after the insert happens, so all of its changes are visible to the trigger.

Triggers (logical)

def: procedure that runs automatically if specified changes in DBMS happen

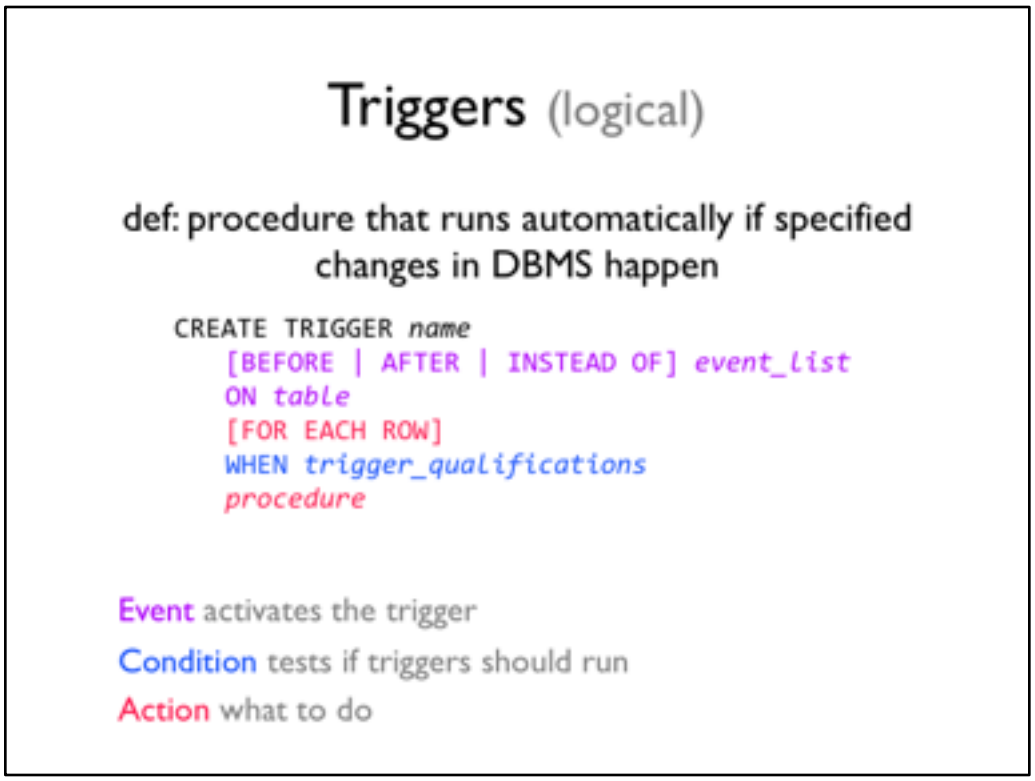
```
CREATE TRIGGER name
    [BEFORE | AFTER | INSTEAD OF] event_list
    ON table

    WHEN trigger_qualifications
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

Condition: boolean expression OR a select query that is true if result is nonempty and false otherwise



Action: procedure to run – has access to condition answers, old and new versions of records being modified (added, rmed, changed) can pretty much do anything including adding new tables. It can run funtions that run arbitrary pieces of code – so it is VERY powerful

Remember the key diff: how many times the procedure is run! Row level, could be a million times if I insert a million records.
If statement, it happens only ONCE!

By default, the trigger procedure is run once for the entire statement – even if no rows are modified, or optionally once for each row that is being inserted/deleted/updated.
The statement level one typically has a way to provide access to the

Copy new young sailors into special table

(logical)

```
CREATE TRIGGER youngSailorUpdate
   AFTER INSERT ON SAILORS
REFERENCING NEW TABLE NewInserts
FOR EACH STATEMENT
   INSERT
      INTO YoungSailors(sid, name, age, rating)
      SELECT sid, name, age, rating
      FROM NewInserts N
      WHERE N.age <= 18
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

AFTER INSERT ON SAILORs is the event, and says that the event should fire after the insertion actually happens.
There are cases where you can prevent inserts by using BEFORE INSERT instead.

NEW TABLE is the table containing the new inserted values
Special syntax for referencing before and after for both the table and the row.

FOR EACH Statement allows both queries to be run – an insert query in this case, or to run an arbitrary function
Here we just look for young sailors in the newinserts table and add them to our yound sailors table

Copy new young sailors into special table

(logical)

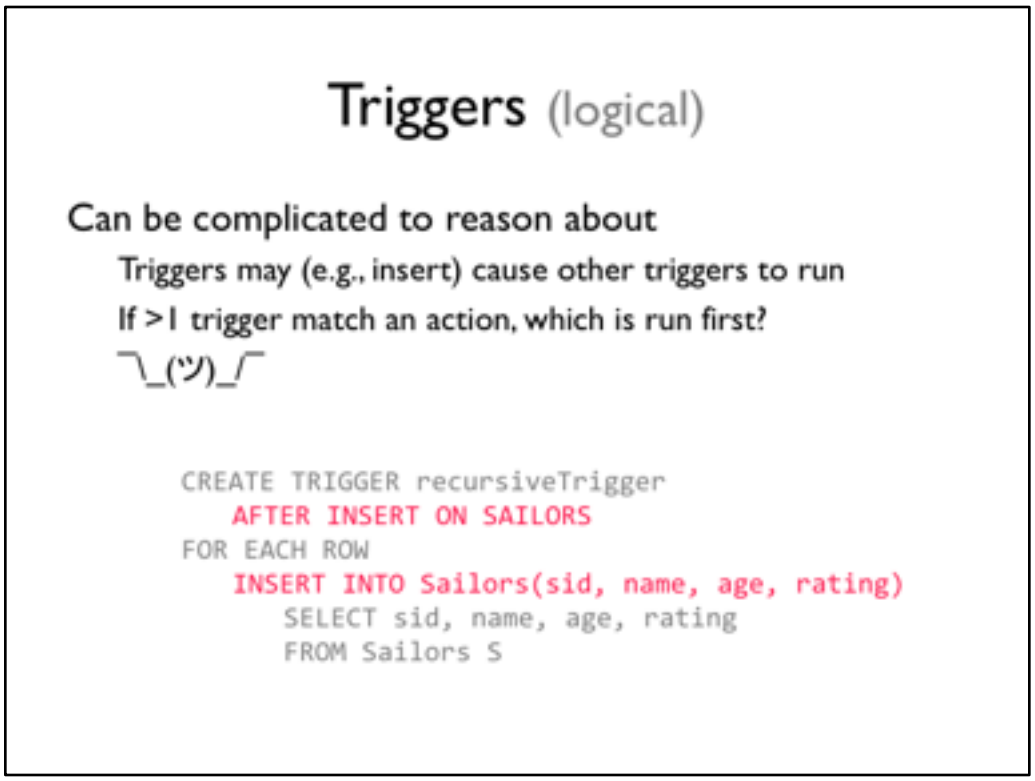
```
CREATE TRIGGER youngSailorUpdate
    AFTER INSERT ON SAILORS
FOR EACH ROW
    WHEN NEW.age <= 18
    INSERT
        INTO YoungSailors (sid, name, age, rating)
        VALUES (NEW.sid, NEW.name, NEW.age, NEW.rating)
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

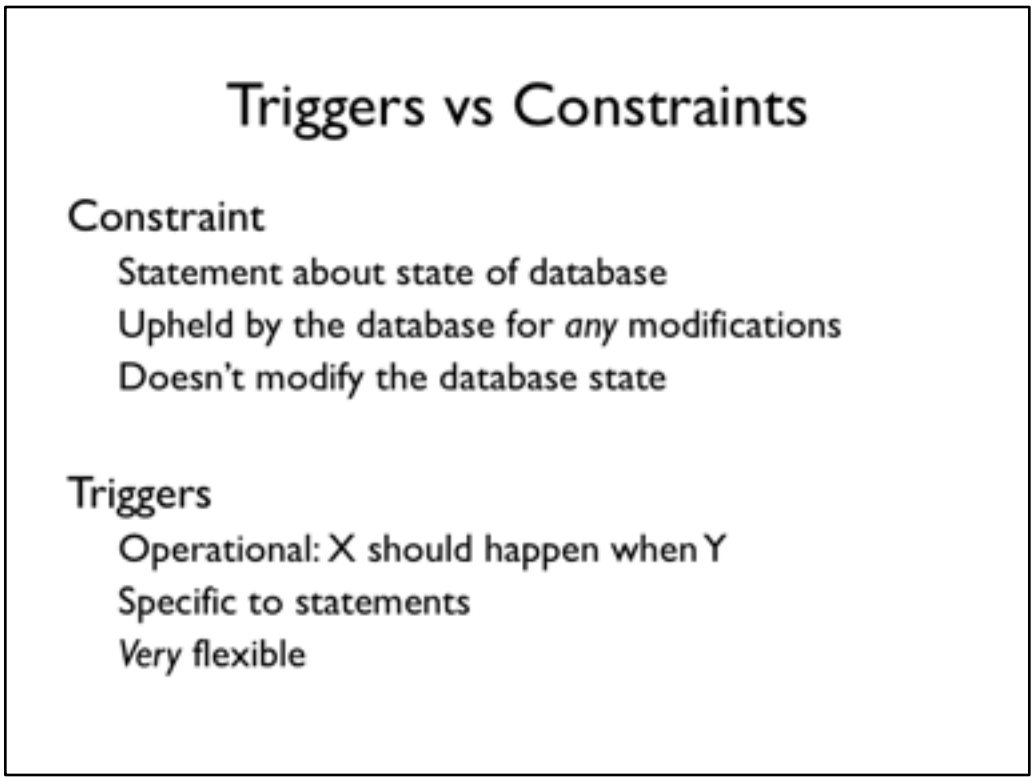
Alternatively, we could use the for each row, not run the trigger action when the row has age <= 18, and insert the rows one at a time



What if there are two BEFORE insert statements, and one cancels the statement while the other inserts a copy into another table? What happens?
What if there are two triggers, one that inserts a record, and one that removes the same record?
-- this is the classic problem with any event based system that performs call backs
-- javascript and user events – which callback is executed first// who knows?
-- what if there's a feedback loop between the callback and the events (callback can generate events)?

What if there are RECURSIVE triggers?

As far as I can tell, statement level triggers are useful for logging, notification, and running queries. I haven't seen a way to access the modified DATA at the statement level – in postgresql.
What I care about
- Overall idea of triggers
- Difference between before after, etc type trigers
- What the pros and cons are
- How to write simple triggers

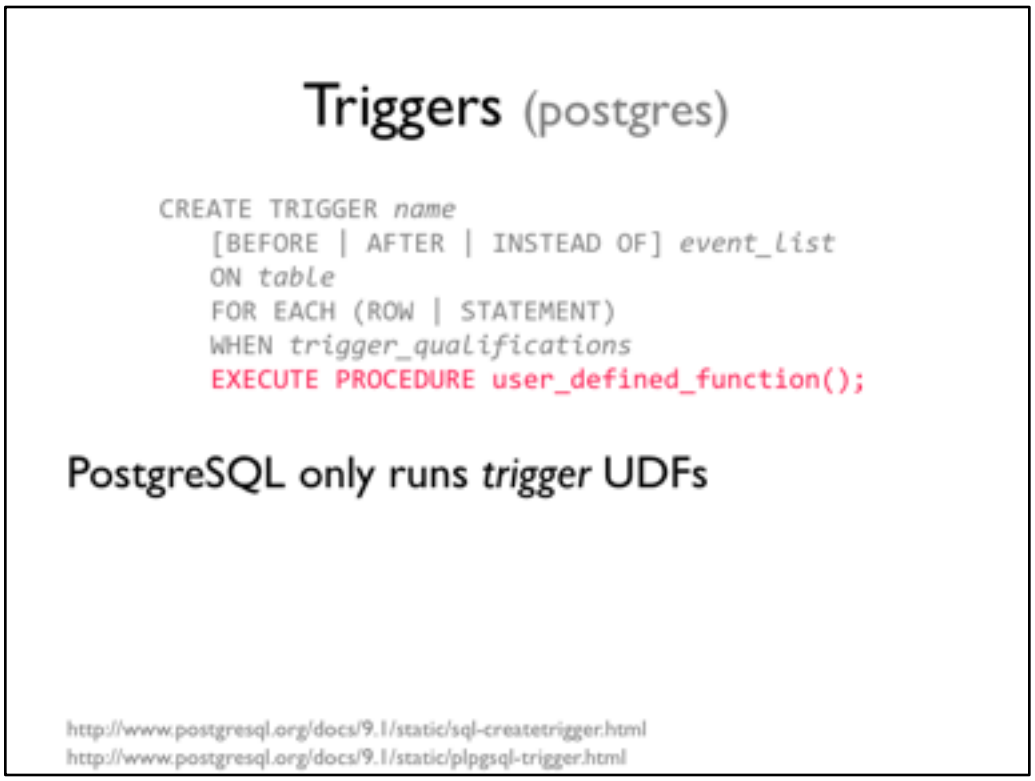


Triggers can be USED to maintain integrity as we have seen. A simple case is using a BEFORE trigger to implement at most one relationships by prohibiting the insertion of more than one relationship (say inserting a second owner for a post).

Let's say we are able to repaint boat colors, so we might update a boat's color and change it.
Then we could use a trigger to keep a copy of each reservation along with the color of the boat at the time of reservation

Let's say we charge sailors for using our boats. Then we could use triggers to provide discounts, or if we are a socialist boat company, we might want to make sure no sailor spends more than 150% more than any other sailor. These are all things we can't do with constraints.

On the otherhand, constraints are simple to reason about, and don't have the complicated mess like ordering and recursion.



Very similar right? The main difference is it runs execute procedure rather than actual statements inline.
This is why we needed to learn about UDFs

BTW, sqlite3 supports triggers as well! It's so powerful. Only row level triggers. The details are VERY vendor specific

Trigger Example

```
CREATE FUNCTION copyrecord() RETURNS trigger
AS $$
BEGIN
    INSERT INTO blah VALUES(NEW.a);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Signature: no args, return type is trigger

Returns NULL or same record structure as modified row

Special variables: OLD, NEW

```
CREATE TRIGGER t_copyinserts BEFORE INSERT ON a
    FOR EACH ROW
        EXECUTE PROCEDURE copyrecord();
```

http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html
http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html

These are special triggers that have a particular signature
No arguments, return type is trigger

The special variables only apply to FOR EACH ROW triggers, because you have access to the entire row.
Notice that if you're doing INSERT, the OLD row isn't avilable (makes sense)
Also, if you are doing a BEFORE UPDATE trigger, the OLD row would also not be available.

Total boats and sailors < 100

```
CREATE FUNCTION checktotal() RETURNS trigger
AS $$
BEGIN
    IF ((SELECT COUNT(*) FROM sailors) +
        (SELECT COUNT(*) FROM boats) < 100) THEN
        RETURN NEW
    ELSE
        RETURN null;
    END IF;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER t_checktotal BEFORE INSERT ON sailors
    FOR EACH ROW
        EXECUTE PROCEDURE checktotal();
```

http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html
http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html

This is how we would write the assertion

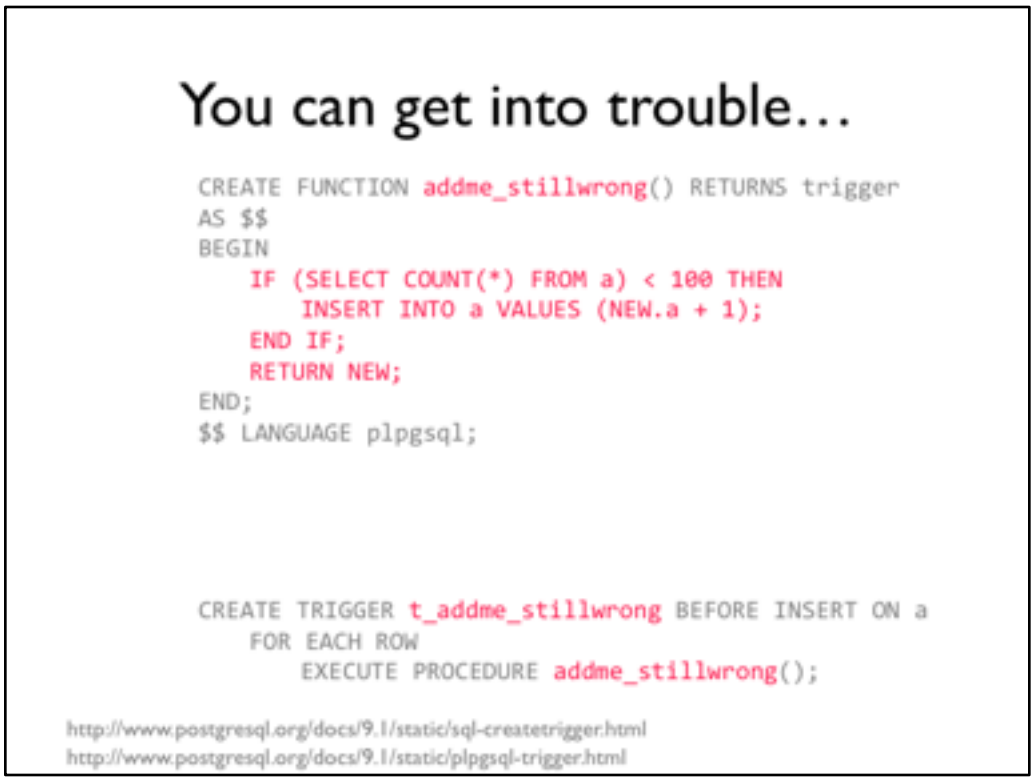
You can get into trouble…

```
CREATE FUNCTION addme_bad() RETURNS trigger
AS $$
BEGIN
    INSERT INTO a VALUES (NEW.*);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER t_addme_bad BEFORE INSERT ON a
    FOR EACH ROW
        EXECUTE PROCEDURE addme_bad();
```

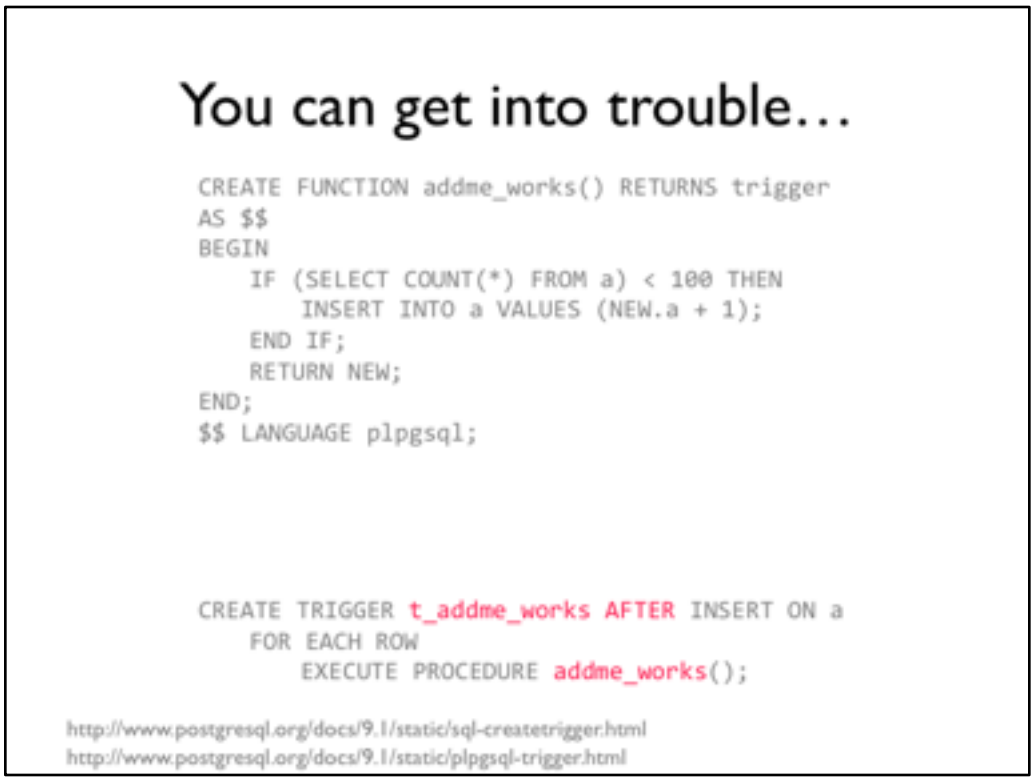
http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html
http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html

This is how we would write the assertion



We'll call this addme_ok , Maybe this could make it better, we'll just limit it to 100 rows
Uh oh, it still hase the same problem. WHY?

Before inserting a record, we run this trigger, so it's an infinite loop still!



We should run it after the insert, so we added the record, then the trigger runs.
We check the count, if it's less than 100 we insert anoher record, then AFTERwards it runs the next trigger.
So it actually stops.

WITH

```
WITH RedBoats(bid, count) AS
   (SELECT   B.bid, count(*)
    FROM     Boats B, Reserves R
    WHERE    R.bid = B.bid AND B.color = 'red'
   GROUP BY  B.bid)
SELECT    name, count
FROM      Boats AS B,  RedBoats AS RB
WHERE     B.bid = RB.bid AND count < 2
```

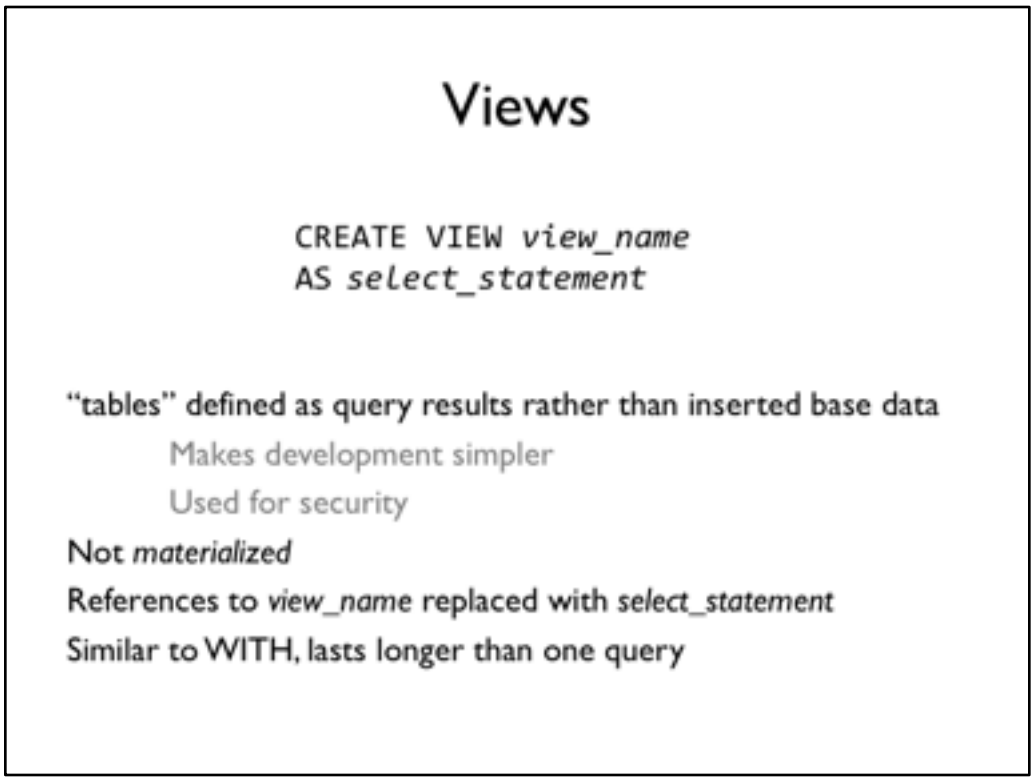
Names of unpopular boats

WITH

```
WITH RedBoats(bid, count) AS
   (SELECT   B.bid, count(*)
    FROM     Boats B, Reserves R
    WHERE    R.bid = B.bid AND B.color = 'red'
   GROUP BY  B.bid)
SELECT    name, count
FROM      Boats AS B,  RedBoats AS RB
WHERE     B.bid = RB.bid AND count < 2
```

```
WITH tablename(attr1, …) AS (select_query)
   [,tablename(attr1, …) AS (select_query)]
main_select_query
```

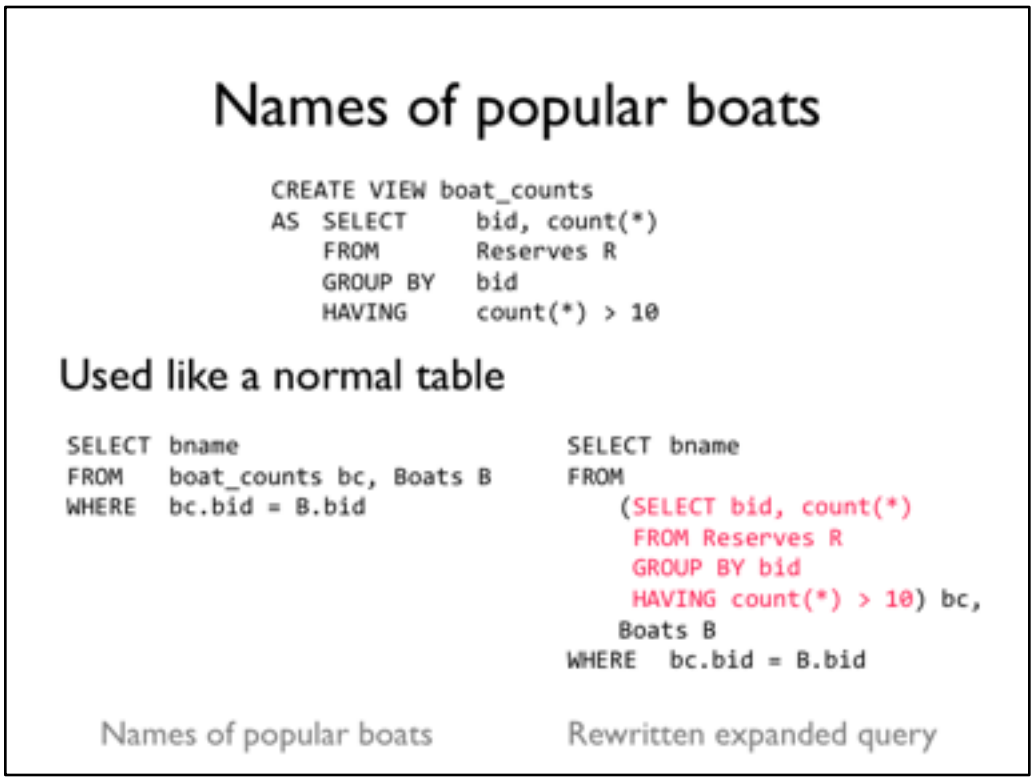
With also allows you to write recursive queries wheer the WITH clause can refer to itself! We won't get into this.



It's like creating a temporary table, but you don't need to waste space, so it's very light weight and has a similar flavor to using the WITH clause before
How is this different than WITH?

Security – you can only give users access to views, so they can ONLY see a subset of the database. For example, if I have different types of users some are admins, some are lowly professors, then professors can only see listings of other professors, but admins can see listings of everyone. You can accomplish this using views.

UPDATES to views tend to be difficult, and there are complex rules about what types of views are updatable – meaning you can run insert/update/delete queries over them.



when you query a view, it equivalent to if you subsituted the viw definition query wherever you referenced the view.
Fore xample if we wanted to know the names of the popular boats.

We have seen numerous ways to write this query, and using views is another way

CREATE TABLE

```
CREATE TABLE <table_name> AS
    <SELECT STATEMENT>
```

Guess the schema:

```
CREATE TABLE used_boats1 AS
    SELECT r.bid
    FROM   Sailors s,
           Reservations r
    WHERE  s.sid = r.sid
```

used_boats1(bid int)

```
CREATE TABLE used_boats2 AS
    SELECT r.bid as foo
    FROM   Sailors s,
           Reservations r
    WHERE  s.sid = r.sid
```

used_boats2(foo int)

How is this different than views?

What if we insert a new record into Reservations?

It's a tradeoff between space and time and a bit more.

On one hand you could create a new tabel based on the result of the select statement and manually maintain it to keep it in sync as the base tables are updated. For example if the select staetment computes popular boats, and I add new reservations that makes the bubba boat really opular, then I would want it to be reflected in the new table.

On the other hand, views simply rewrite any references with the select statement, so it doesn't take any storage space, and will always be in sync, but may thus be slower to run.

What if create table as select 1? And don't give column name? Default name
Create table b as select 1;
Create table b as select 1, 2;

Summary

SQL is pretty complex

- Superset of Relational Algebra SQL99 turing complete!
- Human readable

More than one way to skin a horse

- Many alternatives to write a query
- Optimizer (theoretically) finds most efficient plan

Especially with UDFs, you can do really crazy stuff and we only scratched the surface of UDFs.
Tries to be human readable – great for small queries, crazy for largue 10 table queries.
Many things like views, WITH, etc try to make things a bit simpler