

SQL SQL SQL SQL SQL SQL SQL

Eugene Wu



▲ SQL: One of the most valuable skills (craigkerstiens.com)

816 points by duck 6 days ago | hide | past | web | favorite | 378 comments

▲ slap_shot 6 days ago [-]

SQL is one the most amazing concepts I've ever experienced. It's nearly 5 decades old and there is no sign of a replacement. We've created countless other technologies to store and process data, and we always seem to try to re-create SQL in those technologies (e.g. Hive, Presto, KSQL, etc).

I run a
Craigs p

SQL, and

SQL: One of the Most Valuable Skills

I've learned a lot of skills over the course of my career, but no technical skill more useful than SQL. SQL stands out to me as the most valuable skill for a few reasons:

1. It is valuable across different roles and disciplines
2. Learning it once doesn't really require re-learning
3. You seem like a superhero. *You seem extra powerful when you know it because of the amount of people that aren't fluent*

<https://news.ycombinator.com/item?id=19149792>

<https://mystery.knightlab.com/>

SQL Murder Mystery

Can you find out whodunnit?



Didn't Lecture 3 Go Over SQL?

The SQL 2003 spec is >1400 pages!

Didn't Lecture 3 Go Over SQL?

Two sublanguages

DDL Data Definition Language

define and modify schema (physical, logical, view)

CREATE TABLE, Integrity Constraints

DML Data Manipulation Language

get and modify data

simple SELECT, INSERT, DELETE

human-readable language

Gritty Details

DDL

NULL, Views

DML

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

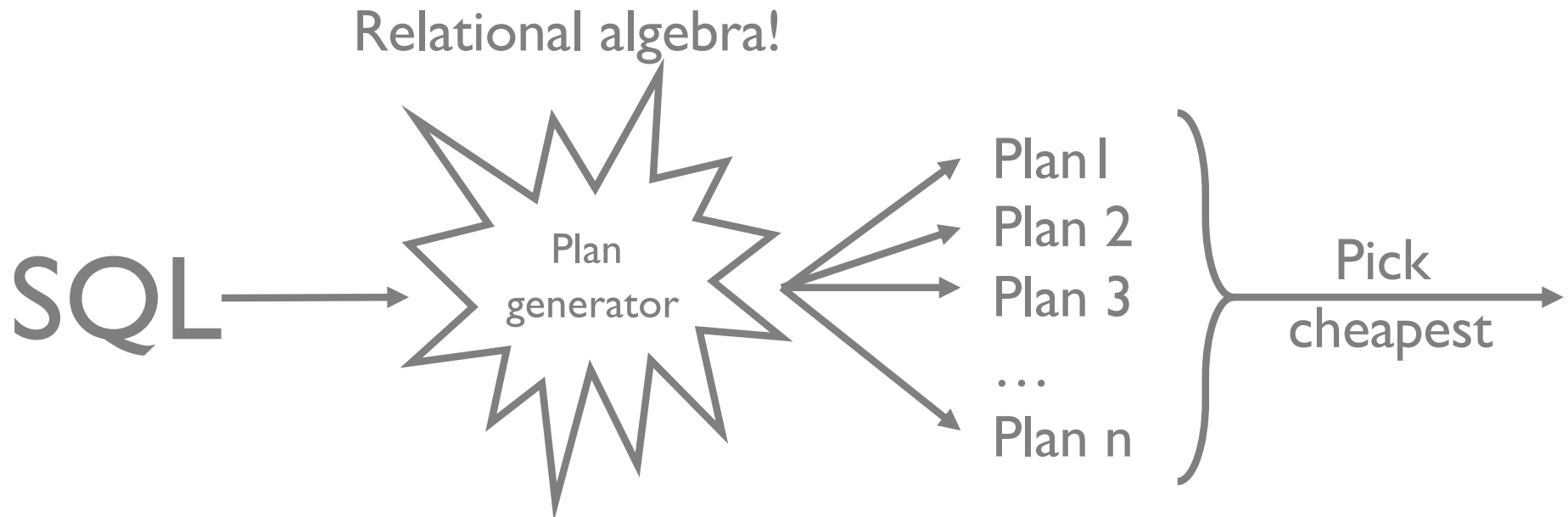
Why a declarative language (SQL)?

DBMS makes it run efficiently

Key: precise query semantics

Reorder/modify queries while answers stay same

DBMS estimates costs for different evaluation plans



SQL Extends Relational Algebra

More expressive power than Rel Alg

- Multisets (bags) rather than sets (allow duplicates)

- Ordering

- NULLs

- Aggregates

- ...

Today's Database

Sailors

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Boats

<u>bid</u>	name	color
101	Legacy	red
102	Melon	blue
103	Mars	red

Reserves

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

Is Reserves table correct?

Today's Database

Sailors

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Boats

<u>bid</u>	name	color
101	Legacy	red
102	Melon	blue
103	Mars	red

Reserves

<u>sid</u>	<u>bid</u>	<u>day</u>
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!

w4111.github.io



Announcements

- 9/23: Notebook examples used in class [can be found here](#). Click on “Open in Colab” on the line with ertosql_private.ipynb

Schedule

Date	Topic	Assigned	Due
------	-------	----------	-----

<30 year old sailors

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

<u>sid</u>	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

<30 year old sailors

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

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

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

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

Multiple Relations

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

$\pi_{\text{name}} (\sigma_{\text{bid}=102}(\text{Sailors} \bowtie_{\text{sid}} \text{Reserves}))$

Sailors

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

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

Structure of a SQL Query

DISTINCT

Optional, answer should not have duplicates
Default: duplicates not removed (multiset)

target-list

List of expressions over attrs of tables in relation-list

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

relation-list

List of relation names
Can define range-variable “AS X”

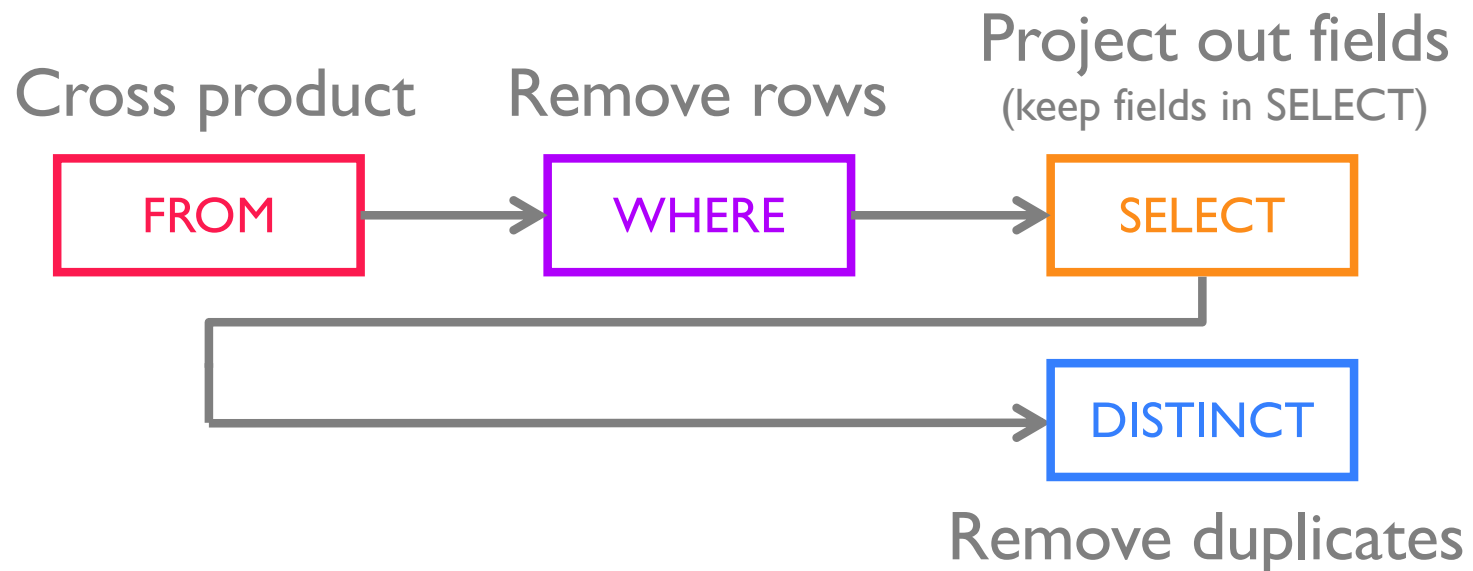
qualification

Boolean expressions

- Combined w/ AND, OR, NOT
- $\text{attr } op \text{ const}$
- $\text{attr}_1 \text{ } op \text{ attr}_2$
- op is =, <, >, !=, etc

Conceptual Query Evaluation

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



Not how actually executed! Above is likely very slow

DISTINCT (vol.I)

Reserves

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

SELECT bid
FROM Reserves

<u>bid</u>
102
102
103

SELECT DISTINCT bid
FROM Reserves

<u>bid</u>
102
103

Sailors that reserved 1+ boats

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

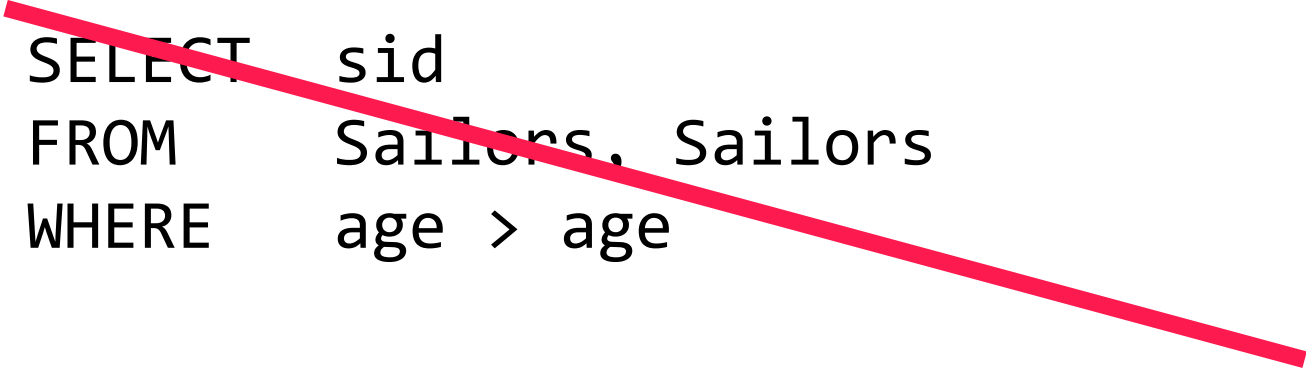
Would DISTINCT change anything in this query?

What if SELECT clause was SELECT S.name?

Range Variables

Disambiguate relations

same table used multiple times (self join)



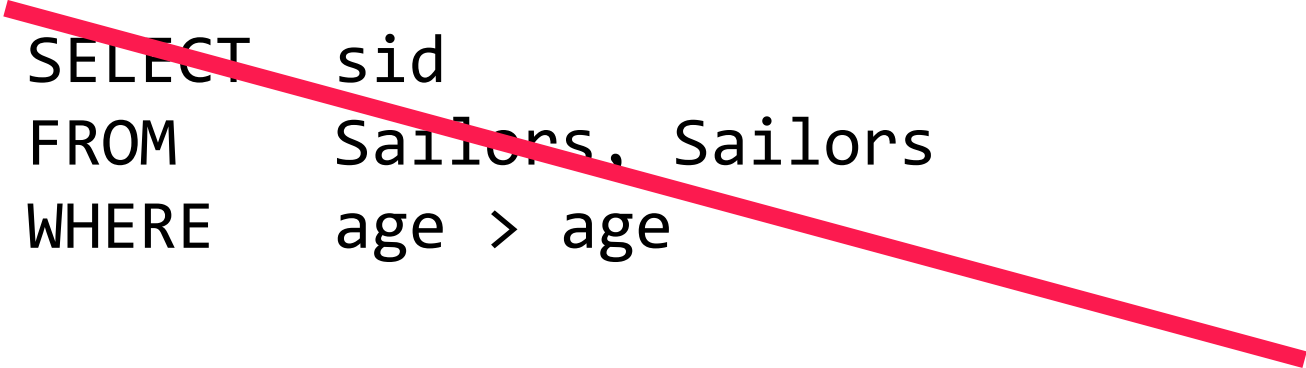
```
SELECT  sid
FROM    Sailors, Sailors
WHERE   age > age
```

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

Range Variables

Disambiguate relations

same table used multiple times (self join)



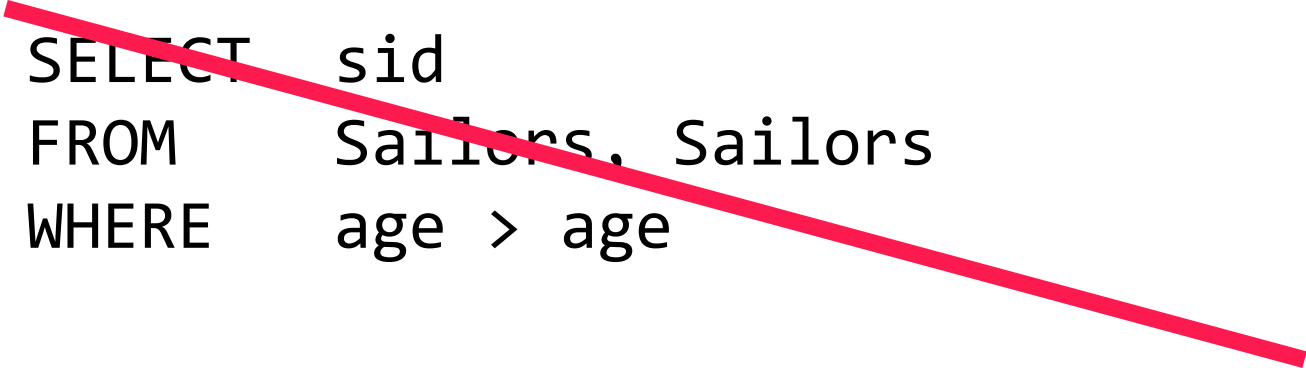
```
SELECT  sid
FROM    Sailors, Sailors
WHERE   age > age
```

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

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 S1, Sailors S2
WHERE   S1.age > S2.age
```

The “AS” is optional

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
```

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>

Expressions (Date/Time)

```
SELECT  R.sid  
FROM    Reserves AS R  
WHERE   now() - R.date < interval '1 day'
```

TIMESTAMP, DATE, TIME types

now() returns timestamp at start of transaction

DBMSes provide rich time manipulation support

exact support may vary by vender

Postgresql Documentation

<http://www.postgresql.org/docs/9.1/static/functions-datetime.html>

Basic Expressions

Constant	2
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')
```

same as...

```
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'
```

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')
```

same as...

```
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'
```

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'
```

INTERSECT

```
CREATE TABLE R(a int)
```

```
CREATE TABLE S(a int)
```

```
INSERT INTO R VALUES (1),(1),(2)
```

```
INSERT INTO S VALUES (1),(2),(2), (3),(1),(1)
```

```
R intersect S = (1), (2)           // remove duplicates
```

```
R intersect ALL S = (1), (1), (2)  // keep common copies
```

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'
```

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'
```

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'
```


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'
```

Midterm Logistics

Exam: expected 75min in length

In person, in class

Extra credit question.

Midterm Logistics

Open book and physical notes.

Writing tool allowed

No electronic resources.

No breaks except emergencies

Professor Wu will answer Qs in the **FIRST 60** minutes of exam.

Midterm Content

All material including SQL lectures

Columbia CS academic honesty policy

Excludes

- Details of hierarchical/network models (should know their pros and cons)

Example of the Classes of Questions

Definitions.

$$A \bowtie_c B = \sigma_c(A \times B)$$

Apply 1 concept.

Find users older than another user.

Apply >1 concepts.

Find tall users older than 3 short users.

Inference based on concept(s).

Max # of result rows?

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 ALL actually takes duplicates into account (multi-set cardinality) will then matter.

SET Comparison Operators

Binary: Relation *op* Relation

op: UNION, INTERSECT, EXCEPT [ALL]

Binary: Tuple *op* Relation

op: IN, NOT IN

Unary: OP Relation

op: [NOT] EXISTS, UNIQUE

Turn Scalar *op* into Set *op*:

op ANY, *op* ALL

op $\in \{ <, >, =, \leq, \geq, \neq, \dots \}$

Many rely on Nested Query Support

Nested Queries

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

a “Subquery”

Nested Queries

```
SELECT  S.sid  
FROM    Sailors S  
WHERE
```



boolean_function(S)

```
for S in Sailors  
    if boolean_function(S):  
        yield S.sid
```

Nested Queries

```
SELECT  S.sid  
FROM    Sailors S  
WHERE   S.sid IN
```



subquery()

```
for S in Sailors  
    if S.sid in subquery():  
        yield S.sid
```

Nested Queries

```
SELECT  S.sid  
FROM    Sailors S  
WHERE   S.sid IN
```



subquery()

```
sq = subquery()  
for S in Sailors  
    if S.sid in sq:  
        yield S.sid
```

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

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

Nested Correlated Queries

```
SELECT  S.sid  
FROM    Sailors S  
WHERE   EXISTS
```



subquery(S.sid)

```
sq = subquery(S.sid)
```

```
for S in Sailors
```

```
    if EXISTS sq:
```

```
        yield S.sid
```

Nested Correlated Queries

```
SELECT  S.sid  
FROM    Sailors S  
WHERE   EXISTS
```




subquery(S.sid)

```
for S in Sailors  
    if EXISTS subquery(S.sid):  
        yield S.sid
```

Nested Correlated Queries

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

bid, sid, date



UNIQUE checks that there are no duplicates

What does this do?

R(bid, sid, date). all attributes are returned from subquery, so we know each record is unique.

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?

Not implemented in PostgreSQL :(

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?

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

Sailors whose rating is greater than
ALL sailors named “Bobby”

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?

Rewrite INTERSECT using IN

```
SELECT S2.name
FROM Sailors S2, (
    SELECT S.sid
    FROM   Sailors S
    WHERE  S.rating > 2

    INTERSECT

    SELECT R.sid
    FROM   Reserves R
) as tmp
WHERE tmp.sid = S2.sid
```

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

Translation harder for INTERSECT ALL

Sailors that reserved all boats (Division)

Hint: double negation

S reserved all boats == no boat that S didn't reserve

```
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)
)
```

Sailors that reserved all boats (Division)

Hint: double negation

S reserved all boats $\equiv \nexists \text{ boat } B \nexists \text{ Sailor } S \text{ reserved } B$

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

Sailors S such that

there's no boat B without

a reservation by S

Sailors that reserved all boats (Division)

Hint: double negation

S reserved all boats $\equiv \nexists$ boat B \nexists Sailor S reserved B

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

Sailors S such that

there's no boat B without

a reservation by S

Sailors that reserved all boats (Division)

Hint: double negation

S reserved all boats == \nexists boat B \nexists Sailor S reserved B

```
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
                                     AND R.bid = B.bid ))
```

Sailors S such that

there's no boat B 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.

NULL

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

Some truth tables

AND	T	F
T	T	F
F	F	F

OR	T	F
T	T	T
F	T	F

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

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

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>

Inner/Natural Join

```
SELECT s.sid, s.name, r.bid  
FROM   Sailors S, Reserves r  
WHERE  s.sid = r.sid
```

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

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

All
Equivalent
for example
tables

Natural Join means equi-join for each pair of
attrs with same name

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

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

<u>sid</u>	<u>bid</u>	<u>day</u>
1	102	9/12
2	102	9/13

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

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

<u>sid</u>	<u>bid</u>	<u>day</u>
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

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

<u>sid</u>	<u>bid</u>	<u>day</u>
1	102	9/12
2	102	9/13

Result

sid	name	bid
1	Eugene	102
2	Luis	102

Notice: No result for Ken!

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 result 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

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

<u>sid</u>	<u>bid</u>	<u>day</u>
1	102	9/12
2	102	9/13

Result

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

Can Left Outer Join be expressed with Cross-Product?

Sailors

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

<u>sid</u>	<u>bid</u>	<u>day</u>
------------	------------	------------

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

Can Left Outer Join be expressed with Cross-Product?

Sailors

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

<u>sid</u>	<u>bid</u>	<u>day</u>
------------	------------	------------

Sailors \bowtie Reserves

U

$(\text{Sailors} - \pi_{\text{sid,name,rating,age}}(\text{Sailors} \bowtie \text{Reserves})) \times \{(\text{null}, \text{null}, \text{null})\}$



How to compute this with a query?

Joins as For Loops

```
for s in Sailors:
```

```
    for r in Reserves:
```

```
        if s.sid = r.sid:
```

```
            yield s, r
```

Inner Join

Joins as For Loops

```
for s in Sailors:
    bmatched = False
    for r in Reserves:
        if s.sid = r.sid:
            yield s, r
            bmatched = True

if not bmatched:
    yield s, null
```

Left
Outer
Join

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

Sailors

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Reserves

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

Result

sid	name	sid	bid
1	Eugene	1	102
2	Luis	2	102
3	Ken	NULL	NULL
NULL	NULL	4	109

Are Functions Special?

DBMSes support custom functions

e.g., `sum(a,b)`, `floor(a)`.

Do functions fit within the relational model?

Functions as Joins

What is $f(x) = x * 2$?

What is $f(8)$?

x
8



x	f(x)
1	1*2 = 2
2	2*2 = 4
3	3*2 = 6
...	...

How big is this relation?

Serious people can count: Aggregation

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

COUNT([DISTINCT] A)

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

SUM([DISTINCT] A)

AVG([DISTINCT] A)

MAX/MIN(A)

STDDEV(A)

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

CORR(A,B)

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

PostgreSQL documentation

<http://www.postgresql.org/docs/9.4/static/functions-aggregate.html>

Name and age of oldest sailor(s)

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

```
SELECT S.name, S.age  
FROM   Sailors S  
WHERE  S.age >= ALL (SELECT S2.age  
                     FROM   Sailors S2)
```

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

```
SELECT S.name, S.age  
FROM   Sailors S  
ORDER BY S.age DESC  
LIMIT 1
```

← When does this not work?

GROUP BY

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

Minimum age among all sailors

What if want minimum 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>
```

What is a list in relational terms?


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

for rating in ratings

SELECT min(s.age)

FROM Sailors s

WHERE s.rating = <rating>

<u>rating</u>
0
1
2
3
4
5
6
7
8
9
10

min()

<u>sid</u>	name	rating	age
1	Eugene	7	22

<u>sid</u>	name	rating	age
5	Darcy	7	32

```
for rating in ratings
```

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

```
    FROM   Sailors s
```

```
    WHERE  s.rating = <rating>
```

rating	
0	
1	
2	
3	
4	
5	
6	
7	age
8	22
9	
10	

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 [100...131]  
    SELECT count(*)  
    FROM   Reserves R  
    WHERE  R.bid = <boat>
```

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*

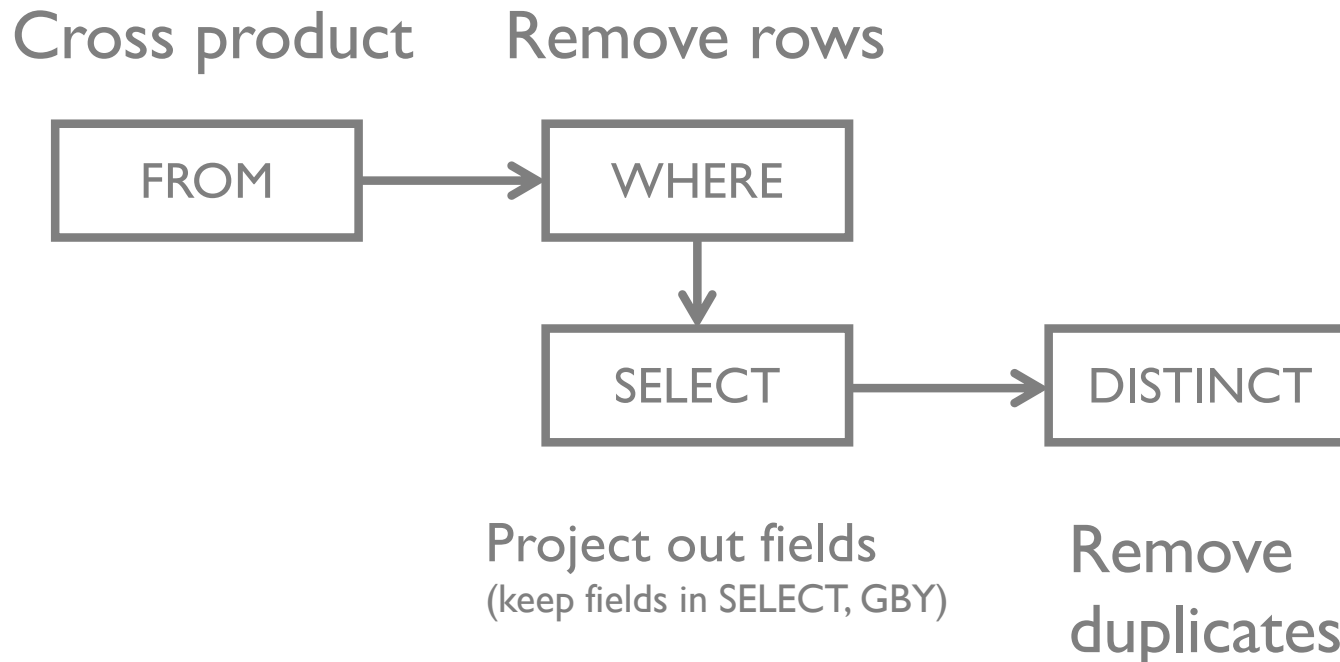
target-list contains

attribute-names \subseteq *grouping-list*

aggregation expressions

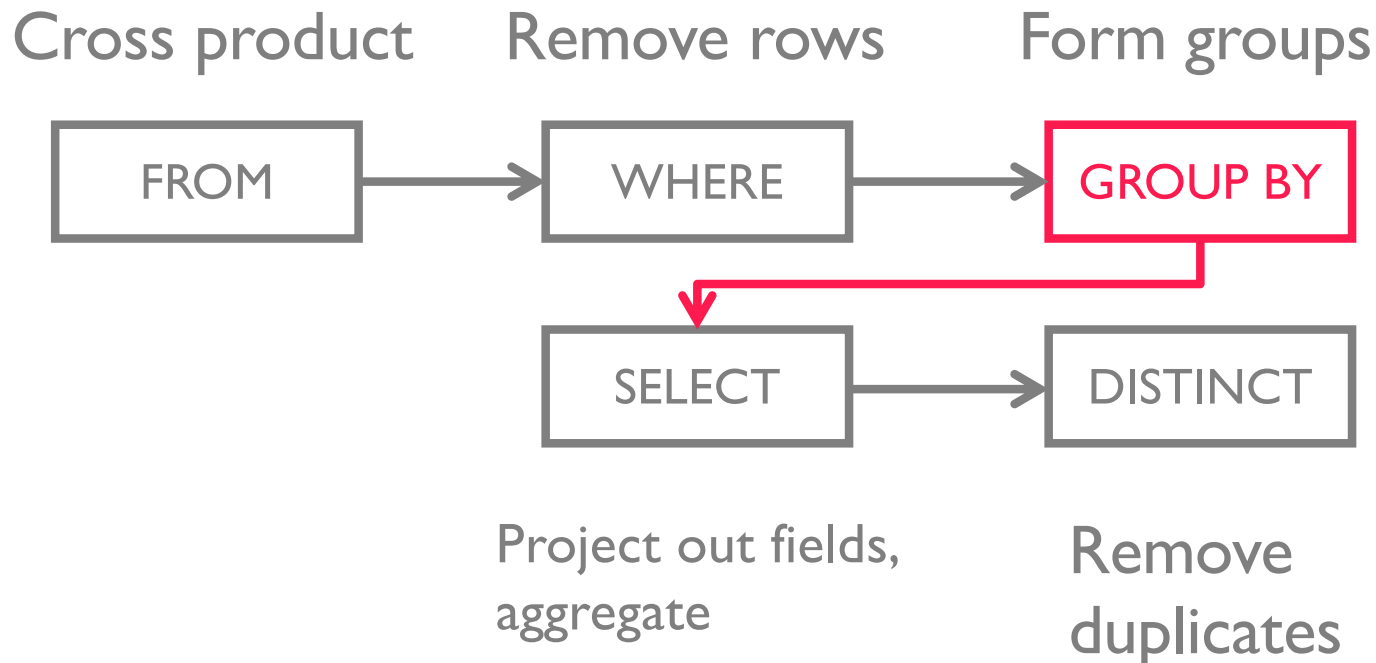
Conceptual Query Evaluation

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



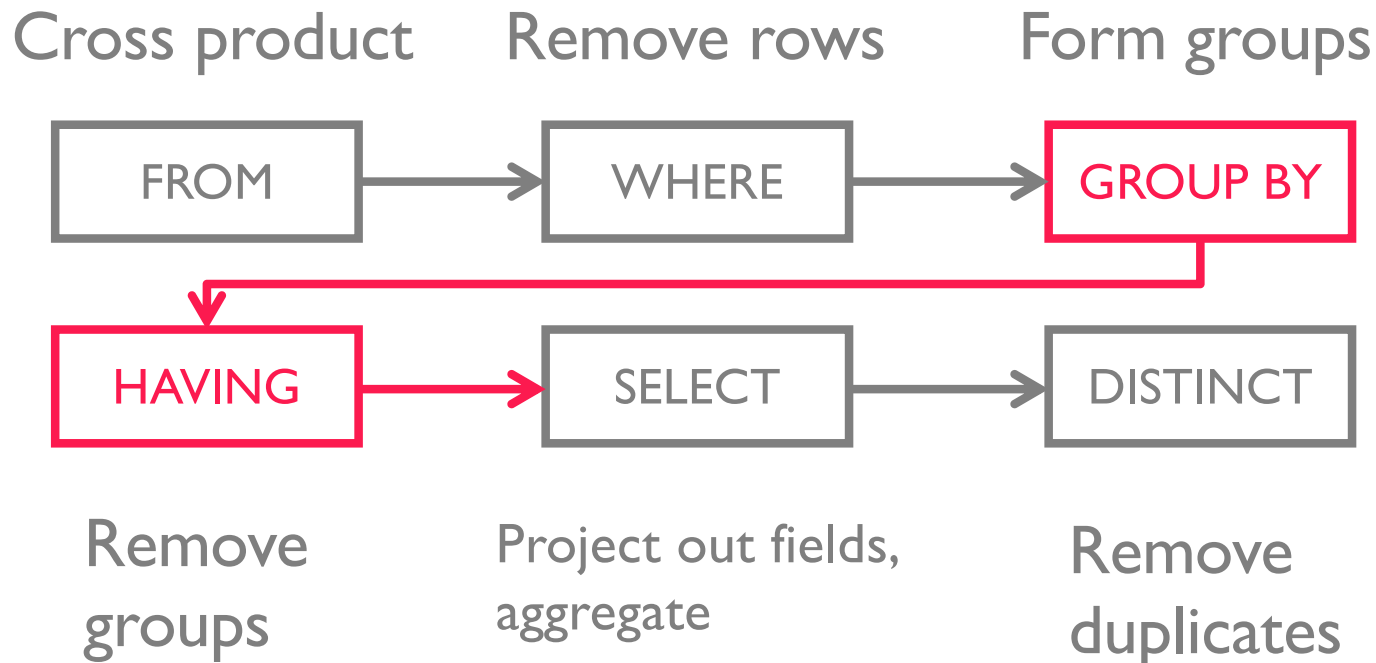
Conceptual Query Evaluation

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

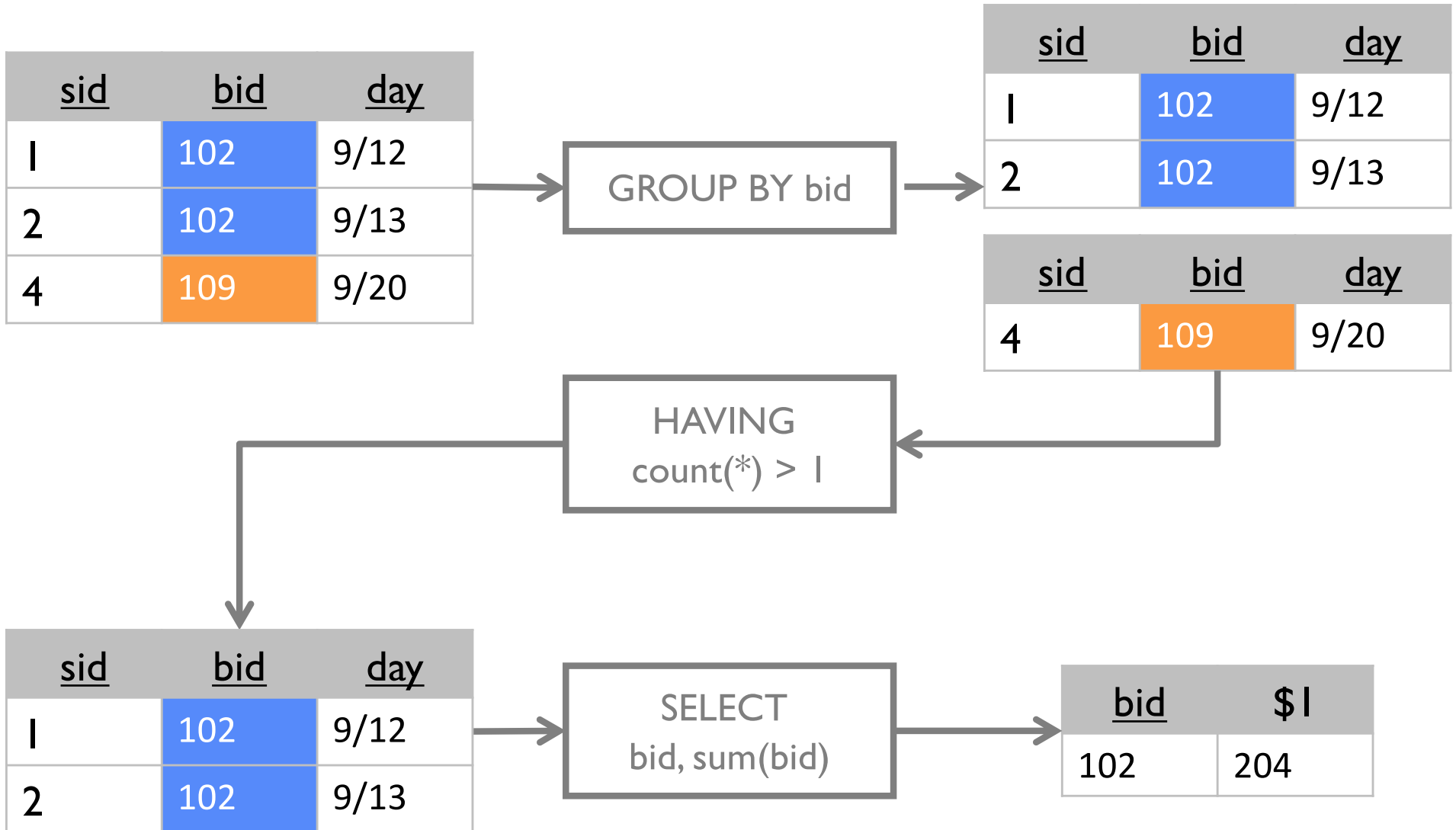


Conceptual Query Evaluation

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



Conceptual Evaluation



GROUP BY

```
SELECT    rating, min(age)
FROM      Sailors
GROUP BY  rating
```

Minimum age for each rating

```
SELECT    min(age)
FROM      Reserves R, Sailors S
WHERE     S.sid = R.sid
GROUP BY  bid
HAVING    count(*) > 2
```

Minimum sailor age

for each boat that has >2 reservations

HAVING

group-qualification used to remove groups
similar to WHERE clause

Its expressions must return *one value per group*.

Either a value in the *grouping-list* or an aggregation function

```
SELECT S.rating, count(*)  
FROM Sailors S  
GROUP BY S.rating  
HAVING name = 'bobby'
```

name may have >1 value in a group!

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
```

Common Mistakes

Aggregation function over a scalar

```
SELECT S.rating, count(*)  
FROM Sailors S  
WHERE sum(age) > 10  
GROUP BY S.rating
```

Scalar expression over a group

```
SELECT S.rating, count(*)  
FROM Sailors S  
GROUP BY S.rating  
HAVING age > 10
```

Ratings where the avg age is the least of all ratings



```
SELECT S.rating
FROM   Sailors S
WHERE  AVG(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
      )
```

Ratings where the avg age is the least of all ratings



```
SELECT S.rating
FROM   Sailors S
WHERE  AVG(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
      )
```


Example 1: Is the following correct?

Query:

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

Alternative:

```
CREATE TABLE tmp(rating, a);
```

```
SELECT tmp.rating
FROM    tmp
WHERE   tmp.a = min(tmp.a)
```

Conceptual evaluation:

```
for t in tmp
    if t.a == min(t.a)    // min over a single value t.a
        yield t
```

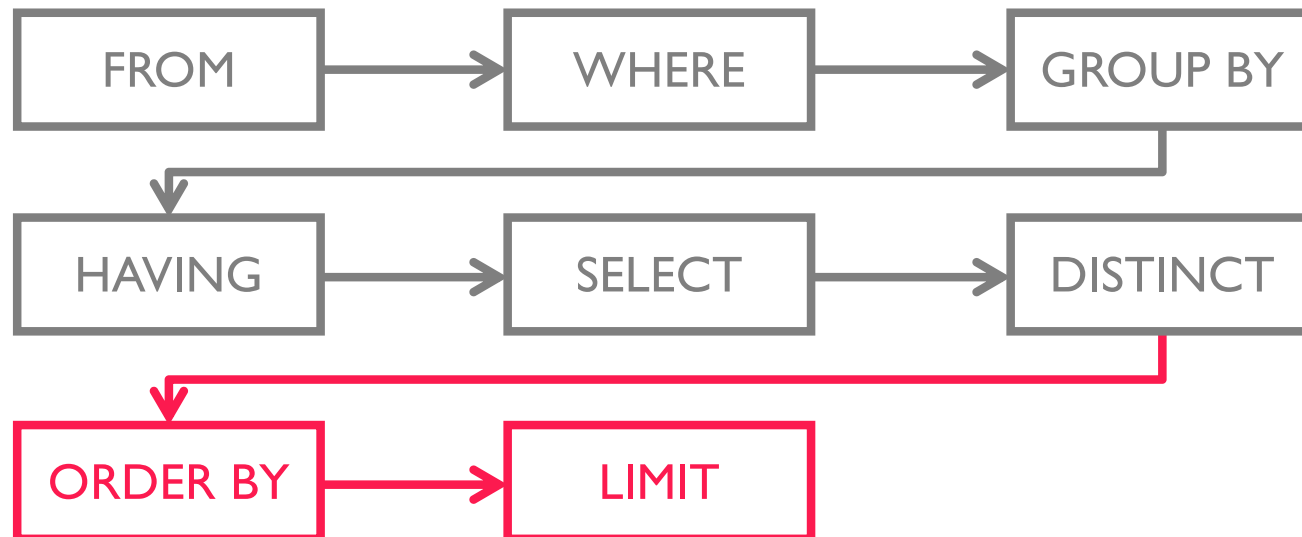
Example 2: Does output cardinality change?

```
SELECT min(age)
  FROM Reserves R JOIN Sailors S
    ON R.sid = S.sid
GROUP BY bid
HAVING count(*) > 2
```

```
SELECT bid, min(age)           // only changes output schema
  FROM Reserves R JOIN Sailors S
    ON R.sid = S.sid
GROUP BY bid
HAVING count(*) > 2
```

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*]



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

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

<u>sid</u>	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

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

<u>sid</u>	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

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

<u>sid</u>	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

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

<u>sid</u>	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

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

WITH, Views, Tables

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

similar to "rename" operator in relational algebra

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
```

(Optional) Recursive WITH

```
WITH RECURSIVE TABLENAME(ATTR NAMES) AS (  
    INITIAL QUERY Q  
    UNION [ALL]  
    RECURSIVE QUERY Q' CAN REFER TO Q  
)  
main_select_query
```

Logical execution model:

Run Q

Run Q' on output of Q

While Q' produced *new* records S

run Q' on S

Run main_select_query

(Optional) Recursive WITH

```
WITH RECURSIVE t(n) AS (  
    VALUES (1)  
    UNION [ALL]  
    SELECT n+1 FROM t  
)  
SELECT sum(n) FROM t;
```

Is there a problem with this query?

(Optional) Recursive WITH

```
WITH RECURSIVE t(n) AS (  
    VALUES (1)  
    UNION [ALL]  
    SELECT n+1 FROM t WHERE n < 10  
)  
SELECT sum(n) FROM t;
```

Fibonacci Series up to 50

```
WITH RECURSIVE fib(n,m) AS (  
    VALUES (0,1)  
    UNION  
    ???  
  
)  
SELECT distinct n  
    FROM fib  
    WHERE n < 50;
```


Fibonacci Series up to 50

```
WITH RECURSIVE fib(n,m) AS (  
    VALUES (0,1)  
    UNION ALL  
    SELECT m,n+m FROM fib  
  
)  
SELECT distinct n  
    FROM fib  
    WHERE n < 50  
ORDER BY n;
```

Fibonacci Series up to 50

```
WITH RECURSIVE fib(n,m) AS (  
    VALUES (0,1)  
    UNION  
    SELECT m,n+m FROM fib  
    WHERE n < 50  
)  
SELECT distinct n  
    FROM fib  
    WHERE n < 50  
ORDER BY n;
```

Views

```
CREATE VIEW view_name AS (  
    select_statement  
)
```

“tables” defined as query results rather than inserted base data

Makes development simpler

Used for security

Not *materialized*

References to *view_name* replaced with *select_statement*

Similar to WITH, lasts longer than one query

Often used for access control

Names of popular boats

```
CREATE VIEW boat_counts AS
  SELECT bid, count(*)
  FROM Reserves R
  GROUP BY bid
  HAVING count(*) > 10
```

Used like a normal table

```
SELECT bname
FROM   boat_counts bc, Boats B
WHERE  bc.bid = B.bid
```

```
SELECT bname
FROM
  (SELECT bid, count(*)
   FROM Reserves R
   GROUP BY bid
   HAVING count(*) > 10) bc,
  Boats B
WHERE  bc.bid = B.bid
```

Names of popular boats

Rewritten expanded query

Views for Access Control

Views used to restrict access to subsets of attrs and rows of the database

Can perform arbitrary transformations, such as encryption

Users/apps query views instead of base relations

```
CREATE TABLE grades(
  studentid text,
  sex text,
  gpa float,
  ispublic bool
)
```

[illegible]

Views as Virtual Databases

```
# Send
import duckdb
db = duckdb.connect("weird_rides.db")
db.sql("""
    CREATE VIEW weird_rides
    AS SELECT pickup_at, dropoff_at, trip_distance, total_amount
    FROM 's3://robotaxi-inc/daily-ride-data/*.parquet'
    WHERE fare_amount > 100 AND trip_distance < 10.0
""")
db.close()
```

```
# Receive
import duckdb
conn = duckdb.connect()
conn.sql("""
ATTACH 's3://robotaxi-inc/virtual-datasets/weird_rides.db'
AS rides_db (READ_ONLY)
""")
```

upload as
weird_rides.db

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?

Not reflected in used_boats

Advanced SQL Topics

General Constraints

User defined functions

Triggers

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 ←

Beyond Keys: Table Constraints

Runs when table is not empty

```
CREATE TABLE Sailors(  
    sid int,  
    ...  
    PRIMARY KEY (sid),  
    CHECK (rating >= 1 AND rating <= 10)
```

```
CREATE TABLE Reserves(  
    sid int,  
    bid int, ←  
    day date,  
    PRIMARY KEY (bid, day),  
    CONSTRAINT no_red_reservations  
    CHECK ('red' NOT IN (SELECT B.color  
                        FROM Boats B  
                        WHERE B.bid = bid))
```

Nested subqueries
Named 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)

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

In practice, most DBMSes don't support:

- General assertions
- Nested queries in CHECK constraints
- Total participation

Why?

- Easy to write something very expensive.
- Requires application logic

Can be done using UDFs + Triggers

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
```

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;
```

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; SQL, PL/SQL, Python, ...
```


A simple UDF (lang = SQL)

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

Schema!



Last statement
is returned



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

```
-- logic
```

```
$$ LANGUAGE language_name;
```

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

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

int4
220
390
270

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

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

int4
220
390
270

Process a Record (lang = SQL)

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

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

Sailors

<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

v
3.285
20.5
3.75

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

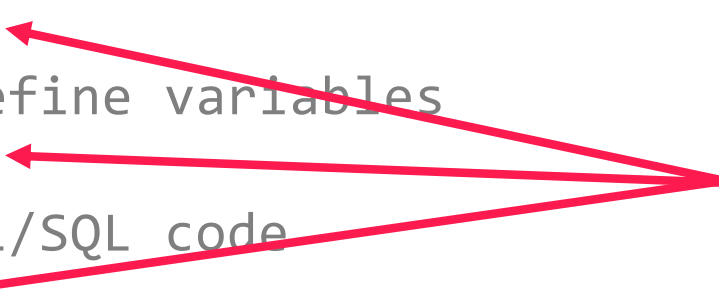
<u>sid</u>	name	rating	age
1	Eugene	7	22
2	Luis	2	39
3	Ken	8	27

Result

int4
3.285
20.5
3.75

Procedural Language/SQL_(lang = plsql)

```
CREATE FUNCTION proc(v int) RETURNS int
AS $$
DECLARE
    -- define variables
BEGIN
    -- PL/SQL code
END;
$$ LANGUAGE plpgsql;
```



Boilerplate

Procedural Language/SQL(lang = plsql)

```
CREATE FUNCTION proc(v int) RETURNS int
AS $$
DECLARE
    -- define variables.  VAR TYPE [= value]
    qty int = 10;
BEGIN
    qty = qty * v;
    IF (SELECT COUNT(*) FROM foo) > 10 THEN
        INSERT INTO blah VALUES(qty);
    END IF;
    RETURN qty + 2;
END;
$$ LANGUAGE plpgsql;
```

Procedural Code (lang = plpython2u)

```
CREATE FUNCTION proc(v int) RETURNS int
AS $$
import random
return random.randint(0, 100) * v
$$ LANGUAGE plpython2u;
```

Very powerful – can do anything so must be careful

run in a python interpreter with no security protection

plpy module provides database access

```
plpy.execute("select 1")
```


Procedural Code (lang = plpython2u)

```
CREATE FUNCTION proc(word text) RETURNS text
AS $$
import requests
resp = requests.get('http://google.com/search?q=%s' % v)
return resp.content.decode('unicode-escape')
$$ LANGUAGE plpython2u;
```

Very powerful – can do anything so must be careful

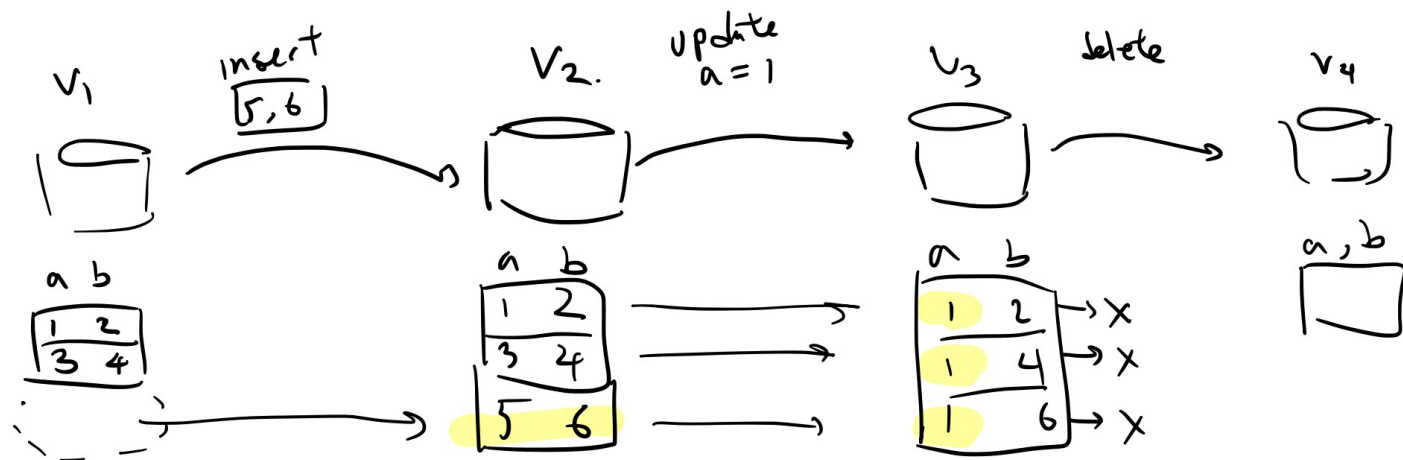
run in a python interpreter with no security protection

plpy module provides database access

```
plpy.execute("select 1")
```

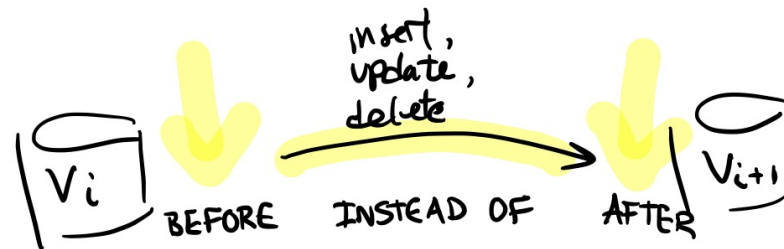
Triggers (background)

- Recall that a database instance is the database Schema + the specific records
- Changing a DB instance essentially creates a new DB instance because the records are different. Let's call each instance a "version" of the DB
- Let's say we made 3 separate changes

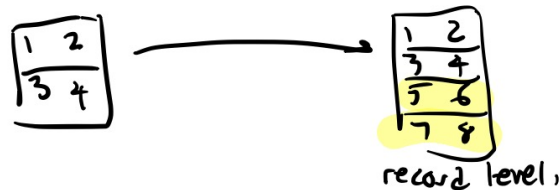


Triggers (background)

- When/where can we add trigger logic?



- At what granularity?



Does a SELECT query create a new version?

No

Triggers (conceptual)

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

```
CREATE TRIGGER name
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

Triggers (conceptual)

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

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



INSERT,
DELETE,
UPDATE

Event activates the trigger

Condition tests if triggers should run

Action what to do

Triggers (conceptual)

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

```
CREATE TRIGGER name  
    [BEFORE | AFTER | INSTEAD OF] event  
    ON table  
  
    WHEN trigger_qualifications
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

Triggers (conceptual)

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

```
CREATE TRIGGER name  
    [BEFORE | AFTER | INSTEAD OF] event  
    ON table  
    [FOR EACH ROW]  
    WHEN trigger_qualifications  
    procedure
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

Copy new young sailors into special table (conceptual)

```
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
```

NewInserts is a virtual table containing new records.

Virtual tables for deleted records, and update records too

Copy new young sailors into special table

(conceptual)

```
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)
```

NEW is special keyword that references each inserted record

Cancel Bulk Inserts

(conceptual)

```
CREATE TRIGGER youngSailorUpdate  
  INSTEAD OF INSERT ON SAILORS  
REFERENCING NEW TABLE NewInserts  
  WHEN (SELECT count(*) FROM NewInserts) < 100  
  INSERT INTO Sailors SELECT * FROM NewInserts
```

Triggers (conceptual)

Can be complicated to reason about

Triggers may (e.g., insert) cause other triggers to run

If >1 trigger match an action, which is run first?

¬_(ツ)_/

```
CREATE TRIGGER recursiveTrigger
  AFTER INSERT ON SAILORS
FOR EACH ROW
  INSERT INTO Sailors(sid, name, age, rating)
    SELECT sid, name, age, rating
    FROM Sailors S
```

Triggers vs Constraints

Constraint

Statement about state of database

Upheld by the database for *any* modifications

Doesn't modify the database state

Safe

Triggers

Operational: X should happen when Y

Specific to statements

Modifies database state

Very flexible, unpredictable

(Optional) Triggers (postgres)

```
CREATE TRIGGER name  
    [BEFORE | AFTER] event_list ON table  
    FOR EACH (ROW | STATEMENT)  
    WHEN trigger_qualifications  
    EXECUTE PROCEDURE user_defined_function();
```

PostgreSQL only runs *trigger* UDFs

<http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html>

<http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html>

(Optional) 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>

(Optional)

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>

(Optional)

You can get into trouble...

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

Will enter infinite loop when you insert a row into a.

```
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>

(Optional)

You can get into trouble...

```
CREATE FUNCTION addme_stillwrong() RETURNS trigger
AS $$
BEGIN
    IF (SELECT COUNT(*) FROM a) < 100 THEN
        INSERT INTO a VALUES (NEW.a + 1);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER t_addme_stillwrong BEFORE INSERT ON a
FOR EACH ROW
EXECUTE PROCEDURE addme_stillwrong();
```

<http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html>

<http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html>

(Optional)

You can get into trouble...

```
CREATE FUNCTION addme_works() RETURNS trigger
AS $$
BEGIN
    IF (SELECT COUNT(*) FROM a) < 100 THEN
        INSERT INTO a VALUES (NEW.a + 1);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER t_addme_works AFTER INSERT ON a
FOR EACH ROW
EXECUTE PROCEDURE addme_works();
```

<http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html>

<http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html>

Summary

SQL is pretty complex

Superset of Relational Algebra SQL99 turing complete!

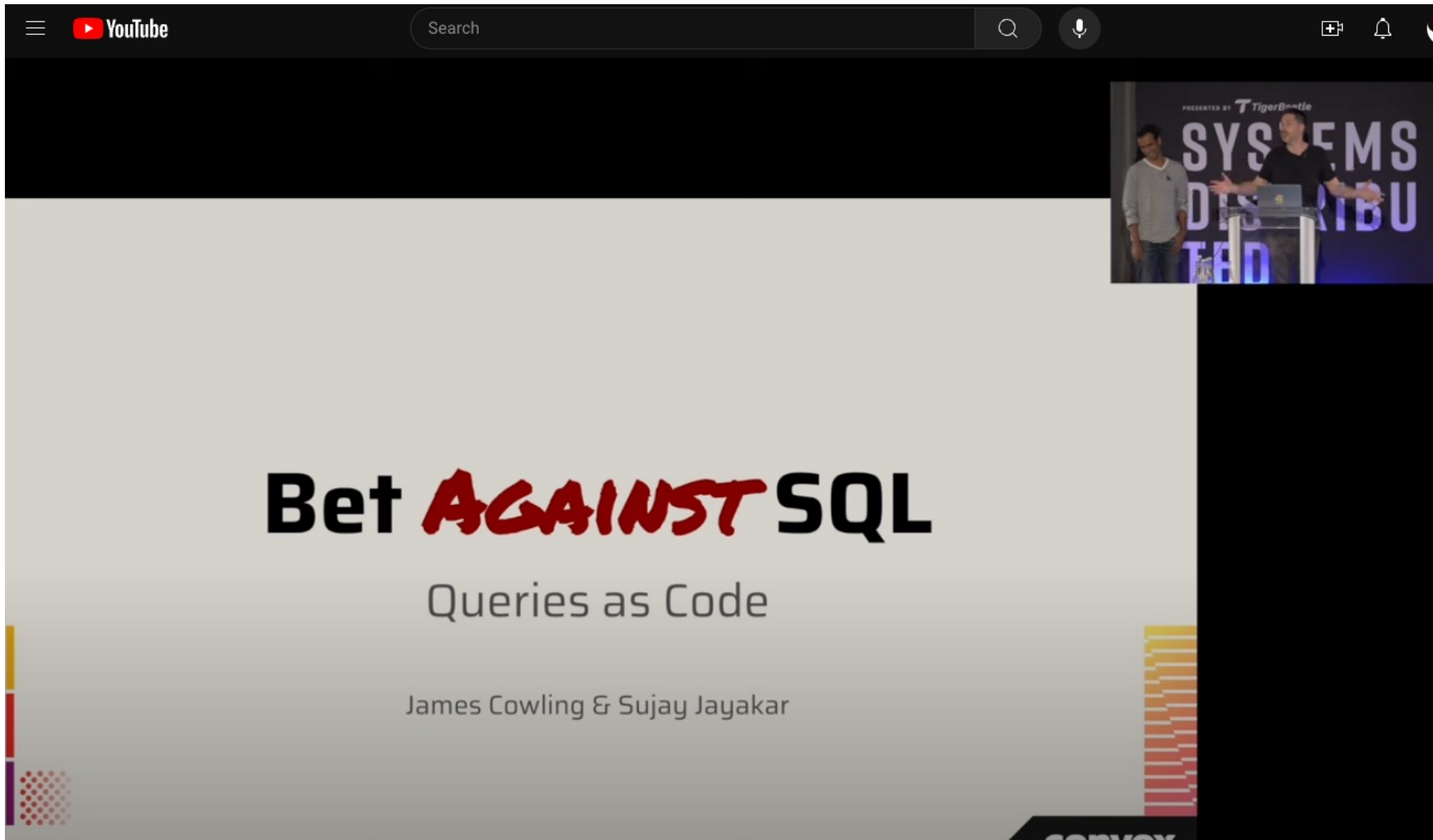
Human readable

More than one way to skin a horse

Many alternatives to write a query

Optimizer (theoretically) finds most efficient plan





<https://www.youtube.com/watch?v=dS9jtih4dI4>

SQL and Beyond

SQL is confusing

- PRQL, Google's Pipe Syntax, SaneQL

SQL Has Problems. We Can Fix Them: Pipe Syntax In SQL

Jeff Shute
Google, Inc.

Shannon Bales
Google, Inc.

Matthew Brown
Google, Inc.

Jean-Daniel Browne
Google, Inc.

Brandon Dolphin
Google, Inc.

Romit Kudtarkar
Google, Inc.

Andrey Litvinov
Google, Inc.

Jingchi Ma
Google, Inc.

John Morcos
Google, Inc.

Michael Shen
Google, Inc.

David Wilhite
Google, Inc.

Xi Wu
Google, Inc.

Lulan Yu
Google, Inc.

additional slides

Some Tricky Queries

Lets write some tricky queries

social graph analysis

statistics

Social Network

-- A directed friend graph. Store each link once

```
CREATE TABLE Friends(  
    fromID int,  
    toID int,  
    since date,  
    PRIMARY KEY (fromID, toID),  
    FOREIGN KEY (fromID) REFERENCES Users,  
    FOREIGN KEY (toID) REFERENCES Users,  
    CHECK (fromID < toID));
```

-- Return edges in both directions

```
CREATE VIEW BothFriends AS  
    SELECT * FROM Friends  
    UNION  
    SELECT F.toID, F.fromID, F.since  
    FROM Friends F;
```


friends of friends of friends do I have?

```
SELECT      count(distinct F3.toID)
FROM        BothFriends F1,
            BothFriends F2,
            BothFriends F3
WHERE       F1.toID = F2.fromID AND
            F2.toID = F3.fromID AND
            F1.fromID = <myid>;
```

friends of friends for each user?

```
SELECT    F1.fromID, count(distinct F3.toID)
FROM      BothFriends F1,
          BothFriends F2,
          BothFriends F3
WHERE     F1.toID = F2.fromID AND
          F2.toID = F3.fromID
GROUP BY  F1.fromID;
```

Median

Given n values in sorted order, value at idx $n/2$
if n is even, can take lower of middle 2

Robust statistics compared to avg

- if want avg to equal 0, what fraction of values need to be corrupted?
- if want median to be 0, what fraction?

Breakdown point of a statistic
crucial if there are outliers
helps with over-fitting

Median

Given n values in sorted order, value at $\text{idx } n/2$

```
SELECT      T.c
FROM        T
ORDER BY    T.c
LIMIT      1
OFFSET      (SELECT COUNT(*)/2
             FROM T AS T2)
```

Median

Given n values in sorted order, value at $\text{idx } n/2$

```
SELECT  c AS median
FROM    T
WHERE
    (SELECT COUNT(*) FROM T AS T1
     WHERE T1.c < T.c)
=
    (SELECT COUNT(*) FROM T AS T2
     WHERE T2.c > T.c);
```

Faster Median

```
SELECT  x.c as median
FROM    T x, T y
GROUP BY x.c
HAVING
    SUM((y.c <= x.c)::int) >= (COUNT(*)+1)/2
    AND
    SUM((y.c >= x.c)::int) >= (COUNT(*)/2)+1
```

Window Functions (not on exam)

How to run queries over ordered data

Partition over a sequence of rows

Each row can be in multiple partitions

```
aggregation OVER (  
    [PARTITION BY attrs]  
    [ORDER BY attrs]  
)
```

Window Functions (not on exam)

1,1,2,3,4,4,5,6

```
SELECT count(*) OVER (PARTITION BY c)
FROM T
```

```
for row in T
    partition = (SELECT * FROM T
                  WHERE T.c = row.c)
    count = len(partition)
    # add count to output row
```


Window Functions (not on exam)

1,1,2,3,4,4,5,6

```
SELECT row_number() OVER (ORDER BY c)
FROM T
```

```
for row in T
    partition = (SELECT * FROM T ORDER BY C)
    row_num = idx of row in partition
    # add rank to output row
```

Window Functions (not on exam)

1,1,2,3,4,4,5,6

```
SELECT row_number() OVER (PARTITION BY C ORDER BY c)
FROM T
```

```
for row in T
    partition = (SELECT * FROM T ORDER BY C)
    row_num = idx of row in partition
    # add rank to output row
```

Window Functions (Median)

How to run queries over ordered data

$O(n \log n)$

Works with even # of items

```
CREATE VIEW twocounters AS
(SELECT    c,
           ROW_NUMBER() OVER (ORDER BY c ASC) AS RowAsc,
           ROW_NUMBER() OVER (ORDER BY c DESC) AS RowDesc
FROM T);
```

```
SELECT AVG(c)
FROM twocounters
WHERE RowAsc IN (RowDesc, RowDesc - 1, RowDesc + 1);
```