

# Administrivia

AAI Grades released

Part I grades released tonight

HW2 due next Tuesday

TWO Exam locations!!

501 NWC if the last UNI digit is 0, 1, 2, 3, 4, or 5

Pupin 329 if the last UNI digit is 6, 7, 8, or 9

# 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
```

# 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?

# 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  
    SELECT m,n+m FROM fib  
  
)  
SELECT distinct n  
    FROM fib  
    WHERE n < 50;
```

# 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;
```



# 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

# 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

# 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?

# 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



**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  $\text{idx } n/2$   
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);
```