Fun With SQL

Joshua Tolley
End Point Corporation

September 29, 2009

"The degree of normality in a database is inversely proportional to that of its DBA." - Anon, twitter

Why Not Do Stuff in SQL

- Databases are harder to replicate, if you **really** need to scale out

Why Not Do Stuff in SQL

- Databases are harder to replicate, if you **really** need to scale out
  - Often, one complex SQL query is more efficient than several simple ones

Why Not Do Stuff in SQL

- Databases are harder to replicate, if you **really** need to scale out
  - Often, one complex SQL query is more efficient than several simple ones
  - Sometimes, indeed, it's useful to reduce the load on the database by moving logic into the application. Be careful doing this

Why Not Do Stuff in SQL

- Databases are harder to replicate, if you **really** need to scale out
  - Often, one complex SQL query is more efficient than several simple ones
  - Sometimes, indeed, it's useful to reduce the load on the database by moving logic into the application. Be careful doing this
    - c.f. Premature Optimization

Why Not Do Stuff in SQL

- Databases are harder to replicate, if you **really** need to scale out
  - Often, one complex SQL query is more efficient than several simple ones
  - Sometimes, indeed, it's useful to reduce the load on the database by moving logic into the application. Be careful doing this
    - c.f. Premature Optimization
- More complex queries are harder to write and debug

Why Not Do Stuff in SQL

- Databases are harder to replicate, if you **really** need to scale out
  - Often, one complex SQL query is more efficient than several simple ones
  - Sometimes, indeed, it's useful to reduce the load on the database by moving logic into the application. Be careful doing this
    - c.f. Premature Optimization
- More complex queries are harder to write and debug
  - True. But so is more complex programming.

Why Not Do Stuff in SQL

- Databases are harder to replicate, if you **really** need to scale out
  - Often, one complex SQL query is more efficient than several simple ones
  - Sometimes, indeed, it's useful to reduce the load on the database by moving logic into the application. Be careful doing this
    - c.f. Premature Optimization
- More complex queries are harder to write and debug
  - True. But so is more complex programming.
- More complex queries are harder for the next guy to maintain

Why Not Do Stuff in SQL

- Databases are harder to replicate, if you **really** need to scale out
  - Often, one complex SQL query is more efficient than several simple ones
  - Sometimes, indeed, it's useful to reduce the load on the database by moving logic into the application. Be careful doing this
    - c.f. Premature Optimization
- More complex queries are harder to write and debug
  - True. But so is more complex programming.
- More complex queries are harder for the next guy to maintain
- Also, good DBAs are often more expensive than good programmers

Why Not Do Stuff in SQL

- Databases are harder to replicate, if you **really** need to scale out
  - Often, one complex SQL query is more efficient than several simple ones
  - Sometimes, indeed, it's useful to reduce the load on the database by moving logic into the application. Be careful doing this
    - c.f. Premature Optimization
- More complex queries are harder to write and debug
  - True. But so is more complex programming.
- More complex queries are harder for the next guy to maintain
- Also, good DBAs are often more expensive than good programmers
  - These are both true. But complex programming is also hard for the next guy to maintain

Why Not Do Stuff in SQL

- Databases are harder to replicate, if you **really** need to scale out
  - Often, one complex SQL query is more efficient than several simple ones
  - Sometimes, indeed, it's useful to reduce the load on the database by moving logic into the application. Be careful doing this
    - c.f. Premature Optimization
- More complex queries are harder to write and debug
  - True. But so is more complex programming.
- More complex queries are harder for the next guy to maintain
- Also, good DBAs are often more expensive than good programmers
  - These are both true. But complex programming is also hard for the next guy to maintain
  - Of all the reasons not to write fluent SQL, this is probably the most widely applicable

Why do stuff in SQL?

- The database is more efficient than your application for processing big chunks of data

Why do stuff in SQL?

- The database is more efficient than your application for processing big chunks of data
  - ...especially if your code is in an interpreted language

Why do stuff in SQL?

- The database is more efficient than your application for processing big chunks of data
  - ...especially if your code is in an interpreted language
- The database is better tested than your application

Why do stuff in SQL?

- The database is more efficient than your application for processing big chunks of data
  - ...especially if your code is in an interpreted language
- The database is better tested than your application
  - Applications trying to do what SQL should be doing often get big and complex quickly

Why do stuff in SQL?

- The database is more efficient than your application for processing big chunks of data
  - ...especially if your code is in an interpreted language
- The database is better tested than your application
  - Applications trying to do what SQL should be doing often get big and complex quickly
  - ...and also buggy quickly

Why do stuff in SQL?

- The database is more efficient than your application for processing big chunks of data
  - ...especially if your code is in an interpreted language
- The database is better tested than your application
  - Applications trying to do what SQL should be doing often get big and complex quickly
  - ...and also buggy quickly
- That's what the database is there for

Why do stuff in SQL?

- The database is more efficient than your application for processing big chunks of data
  - ...especially if your code is in an interpreted language
- The database is better tested than your application
  - Applications trying to do what SQL should be doing often get big and complex quickly
  - ...and also buggy quickly
- That's what the database is there for
- SQL is designed to express relations and conditions on them. Your application's language isn't.

Why do stuff in SQL?

- The database is more efficient than your application for processing big chunks of data
  - ...especially if your code is in an interpreted language
- The database is better tested than your application
  - Applications trying to do what SQL should be doing often get big and complex quickly
  - ...and also buggy quickly
- That's what the database is there for
- SQL is designed to express relations and conditions on them. Your application's language isn't.
- A better understanding of SQL allows you to write queries that perform better

Why do stuff in SQL?

In short, the database exists to manage data, and your application exists to handle business logic. Write software accordingly.

So let's get started...

Tables we'll use

```
# SELECT * FROM a;
 id | value
----+-------
  1 | a1
  2 | a2
  3 | a3
  4 | a4
(4 rows)
```

```
# SELECT * FROM b;
 id | value
----+-------
  5 | b5
  4 | b4
  3 | b3
  6 | b6
(4 rows)
```

JOINs

- If you want data from multiple tables, you probably want a join

JOINs

- If you want data from multiple tables, you probably want a join
  - ...but see also Subqueries, later on

JOINs

- If you want data from multiple tables, you probably want a join
  - ...but see also Subqueries, later on
- There are several different kinds of joins

JOINs

```
<table1> [alias1]
    [ [ [NATURAL] [ [FULL | RIGHT | LEFT] [OUTER] |
    INNER] ] | CROSS ] JOIN
<table2> [alias2]
    [USING (...) |
    ON (<value1> <op> <value2>
        [,<value3> <op> <value4>...] ) ]
```

CROSS JOIN

- SELECT <... >FROM table1 JOIN table2

CROSS JOIN

- SELECT <... >FROM table1 JOIN table2
- With no explicit join type and no join qualifiers (an ON clause, WHERE clause involving both relations, etc.) this is a CROSS JOIN

Joshua Tolley
End Point Corporation

CROSS JOIN

- SELECT <... >FROM table1 JOIN table2
- With no explicit join type and no join qualifiers (an ON clause, WHERE clause involving both relations, etc.) this is a CROSS JOIN
- Equivalent to
  - SELECT <... >FROM table1, table2

CROSS JOIN

- SELECT <... >FROM table1 JOIN table2
- With no explicit join type and no join qualifiers (an ON clause, WHERE clause involving both relations, etc.) this is a CROSS JOIN
- Equivalent to
  - SELECT <... >FROM table1, table2
  - SELECT <... >FROM table1 CROSS JOIN table2

CROSS JOIN

- SELECT <... >FROM table1 JOIN table2
- With no explicit join type and no join qualifiers (an ON clause, WHERE clause involving both relations, etc.) this is a CROSS JOIN
- Equivalent to
  - SELECT <... >FROM table1, table2
  - SELECT <... >FROM table1 CROSS JOIN table2
- "Cartesian product" of the two relations

CROSS JOIN

- SELECT <... >FROM table1 JOIN table2
- With no explicit join type and no join qualifiers (an ON clause, WHERE clause involving both relations, etc.) this is a CROSS JOIN
- Equivalent to
  - SELECT <... >FROM table1, table2
  - SELECT <... >FROM table1 CROSS JOIN table2
- "Cartesian product" of the two relations
  - Combines every row of table1 with every row of table2

CROSS JOIN

- SELECT <... >FROM table1 JOIN table2
- With no explicit join type and no join qualifiers (an ON clause, WHERE clause involving both relations, etc.) this is a CROSS JOIN
- Equivalent to
  - SELECT <... >FROM table1, table2
  - SELECT <... >FROM table1 CROSS JOIN table2
- "Cartesian product" of the two relations
  - Combines every row of table1 with every row of table2
  - Makes **LOTS** of rows, and can thus be very slow

CROSS JOIN

```
# SELECT * FROM a, b;
 id | value | id | value
----+-------+----+-------
  1 | a1    |  5 | b5
  1 | a1    |  4 | b4
<snip>
  3 | a3    |  3 | b3
  3 | a3    |  6 | b6
  4 | a4    |  5 | b5
  4 | a4    |  4 | b4
  4 | a4    |  3 | b3
  4 | a4    |  6 | b6
(16 rows)
```

INNER JOIN

- SELECT <...>FROM table1 INNER JOIN table2 ON (table1.field = table2.field ...)

INNER JOIN

- SELECT <...>FROM table1 INNER JOIN table2 ON (table1.field = table2.field ...)
- Only returns rows satisfying the ON condition

INNER JOIN

- SELECT <...>FROM table1 INNER JOIN table2 ON (table1.field = table2.field ...)
- Only returns rows satisfying the ON condition
- Equivalent to a CROSS JOIN with a WHERE clause

INNER JOIN

```
# SELECT * FROM a INNER JOIN b USING (id);
 id | value | value
----+-------+-------
  3 | a3    | b3
  4 | a4    | b4
(2 rows)
```

OUTER JOIN

- Return all rows from one or both relations

OUTER JOIN

- Return all rows from one or both relations
- LEFT: Return all rows from the relation on the left

OUTER JOIN

- Return all rows from one or both relations
- LEFT: Return all rows from the relation on the left
- RIGHT: Return all rows from the relation on the right

OUTER JOIN

- Return all rows from one or both relations
- LEFT: Return all rows from the relation on the left
- RIGHT: Return all rows from the relation on the right
- FULL: Return all rows from both relations

OUTER JOIN

- Return all rows from one or both relations
- LEFT: Return all rows from the relation on the left
- RIGHT: Return all rows from the relation on the right
- FULL: Return all rows from both relations
- Returns nulls for values from one relation when it contains to match with the other relation

OUTER JOIN

- Return all rows from one or both relations
- LEFT: Return all rows from the relation on the left
- RIGHT: Return all rows from the relation on the right
- FULL: Return all rows from both relations
- Returns nulls for values from one relation when it contains to match with the other relation
- The OUTER keyword is redundant

OUTER JOIN

- Return all rows from one or both relations
- LEFT: Return all rows from the relation on the left
- RIGHT: Return all rows from the relation on the right
- FULL: Return all rows from both relations
- Returns nulls for values from one relation when it contains to match with the other relation
- The OUTER keyword is redundant
- Requires ON or USING clause

LEFT JOIN

```
# SELECT * FROM a LEFT JOIN b USING (id);
 id | value | value
----+-------+-------
  1 | a1    |
  2 | a2    |
  3 | a3    | b3
  4 | a4    | b4
(4 rows)
```

RIGHT JOIN

```
# SELECT * FROM a RIGHT JOIN b USING (id);
 id | value | value
----+-------+-------
  3 | a3    | b3
  4 | a4    | b4
  5 |       | b5
  6 |       | b6
(4 rows)
```

FULL JOIN

```
# select * from a full join b using (id);
 id | value | value
----+-------+-------
  1 | a1    |
  2 | a2    |
  3 | a3    | b3
  4 | a4    | b4
  5 |       | b5
  6 |       | b6
(6 rows)
```

Applications

Find rows with no match in table b:

```
# SELECT * FROM a LEFT JOIN b USING (id)
    WHERE b.value IS NULL;
 id | value | value
----+-------+-------
  1 | a1    |
  2 | a2    |
(2 rows)
```

NATURAL JOIN

- NATURAL is syntactic sugar to match all columns with the same name

NATURAL JOIN

```
# SELECT * FROM a NATURAL FULL JOIN b;
 id | value
----+-------
  1 | a1
  2 | a2
  3 | a3
  3 | b3
  4 | a4
  4 | b4
  5 | b5
  6 | b6
(8 rows)
```

This looked for matches in both the *id* and *value* columns, so no rows matched. It returned all rows of both relations because it's a FULL JOIN.

Self Joins

- "Self joins" are particularly counterintuitive

Self Joins

- "Self joins" are particularly counterintuitive
- Joins one table to itself

Self Joins

- "Self joins" are particularly counterintuitive
- Joins one table to itself
- It helps to give the table two different aliases

Self Joins

Find all employees' names, and each employee's manager

```
SELECT
    e.first || ' ' || e.last,
    (SELECT
        m.first || ' ' || m.last
     FROM employee m
     WHERE m.id = e.manager);
```

... will generally be much faster rewritten as ...

```
SELECT
    e.first || ' ' || e.last,
    m.first || ' ' || m.last
FROM
    employee e
    JOIN employee m ON (e.manager = m.id)
```

More useful operations...

Subqueries

- Embeds one query within another

Subqueries

- Embeds one query within another
- Examples (some bad, some good)

Subqueries

- Embeds one query within another
- Examples (some bad, some good)
  - SELECT id FROM table WHERE field = (SELECT MAX(field) FROM table)

Subqueries

- Embeds one query within another
- Examples (some bad, some good)
  - SELECT id FROM table WHERE field = (SELECT MAX(field) FROM table)
  - SELECT id, (SELECT COUNT(*) FROM table2 WHERE id = table1.id) FROM table1

Subqueries

- Embeds one query within another
- Examples (some bad, some good)
  - SELECT id FROM table WHERE field = (SELECT MAX(field) FROM table)
  - SELECT id, (SELECT COUNT(*) FROM table2 WHERE id = table1.id) FROM table1
  - SELECT a, b FROM (SELECT a, COUNT(*) AS c FROM table1) t1 JOIN (SELECT b, COUNT(*) AS c FROM table2) t2 on (t1.c = t2.c)

Subqueries

- Embeds one query within another
- Examples (some bad, some good)
  - SELECT id FROM table WHERE field = (SELECT MAX(field) FROM table)
  - SELECT id, (SELECT COUNT(*) FROM table2 WHERE id = table1.id) FROM table1
  - SELECT a, b FROM (SELECT a, COUNT(*) AS c FROM table1) t1 JOIN (SELECT b, COUNT(*) AS c FROM table2) t2 on (t1.c = t2.c)
    - You can join subqueries just like you'd join tables

Set Operations

- INTERSECT

Set Operations

- INTERSECT
  - Returns the intersection of two sets

Set Operations

- INTERSECT
  - Returns the intersection of two sets
  - Doesn't exist in MySQL

Set Operations

- INTERSECT
  - Returns the intersection of two sets
  - Doesn't exist in MySQL
  - SELECT (SELECT a, b FROM table1) INTERSECT (SELECT c, d FROM table2)

Set Operations

- INTERSECT
  - Returns the intersection of two sets
  - Doesn't exist in MySQL
  - SELECT (SELECT a, b FROM table1) INTERSECT (SELECT c, d FROM table2)
- UNION

Set Operations

- INTERSECT
  - Returns the intersection of two sets
  - Doesn't exist in MySQL
  - SELECT (SELECT a, b FROM table1) INTERSECT (SELECT c, d FROM table2)
- UNION
  - Appends one set of rows to another set with matching column types

Set Operations

- INTERSECT
  - Returns the intersection of two sets
  - Doesn't exist in MySQL
  - SELECT (SELECT a, b FROM table1) INTERSECT (SELECT c, d FROM table2)
- UNION
  - Appends one set of rows to another set with matching column types
  - SELECT a FROM table1 UNION SELECT b FROM table2

Set Operations

- INTERSECT
  - Returns the intersection of two sets
  - Doesn't exist in MySQL
  - SELECT (SELECT a, b FROM table1) INTERSECT (SELECT c, d FROM table2)
- UNION
  - Appends one set of rows to another set with matching column types
  - SELECT a FROM table1 UNION SELECT b FROM table2
- EXCEPT

Set Operations

- INTERSECT
  - Returns the intersection of two sets
  - Doesn't exist in MySQL
  - SELECT (SELECT a, b FROM table1) INTERSECT (SELECT c, d FROM table2)
- UNION
  - Appends one set of rows to another set with matching column types
  - SELECT a FROM table1 UNION SELECT b FROM table2
- EXCEPT
  - Returns rows in one SELECT that aren't in another SELECT

Set Operations

- INTERSECT
  - Returns the intersection of two sets
  - Doesn't exist in MySQL
  - SELECT (SELECT a, b FROM table1) INTERSECT (SELECT c, d FROM table2)
- UNION
  - Appends one set of rows to another set with matching column types
  - SELECT a FROM table1 UNION SELECT b FROM table2
- EXCEPT
  - Returns rows in one SELECT that aren't in another SELECT
  - SELECT a FROM table1 EXCEPT SELECT b FROM table2

Common Operations

- COALESCE(a, b)

Common Operations

- COALESCE(a, b)
  - If `a` is null, return `b`, else return `a`

Common Operations

- COALESCE(a, b)
  - If a is null, return b, else return a
  - SELECT COALESCE(first, '<NULL>') FROM table

Common Operations

- COALESCE(a, b)
  - If a is null, return b, else return a
  - SELECT COALESCE(first, '<NULL>') FROM table
  - Oracle calls this NVL()

Common Operations

- COALESCE(a, b)
  - If a is null, return b, else return a
  - SELECT COALESCE(first, '<NULL>') FROM table
  - Oracle calls this NVL()
- CASE...WHEN

Common Operations

- COALESCE(a, b)
  - If a is null, return b, else return a
  - SELECT COALESCE(first, '<NULL>') FROM table
  - Oracle calls this NVL()
- CASE...WHEN
  - Conditional operation

Common Operations

- COALESCE(a, b)
  - If a is null, return b, else return a
  - SELECT COALESCE(first, '<NULL>') FROM table
  - Oracle calls this NVL()
- CASE...WHEN
  - Conditional operation
  - SELECT CASE WHEN langused IN ('Lisp', 'OCaml', 'Haskell') THEN 'Functional' ELSE 'Imperative' AS langtype FROM software

Series Generation

- generate_series() in PostgreSQL; might be something else in other databases

Series Generation

- generate_series() in PostgreSQL; might be something else in other databases
- Returns a series of numbers

Series Generation

- generate_series() in PostgreSQL; might be something else in other databases
- Returns a series of numbers
- Can be used like a `for` loop (example given later)

```
# SELECT * FROM generate_series(1, 5);
 generate_series
-----------------
               1
               2
               3
               4
               5
(5 rows)
```

Common Table Expressions

- Abbreviated CTEs

Common Table Expressions

- Abbreviated CTEs
- Fairly advanced; not available in all databases

Common Table Expressions

- Abbreviated CTEs
- Fairly advanced; not available in all databases
  - Not in PostgreSQL before v. 8.4, or any version of MySQL
- It's just like defining a one-time view for your query

Common Table Expressions

- Abbreviated CTEs
- Fairly advanced; not available in all databases
  - Not in PostgreSQL before v. 8.4, or any version of MySQL
- It's just like defining a one-time view for your query
- One major benefit: CTEs allow recursion

Common Table Expressions

- Abbreviated CTEs
- Fairly advanced; not available in all databases
  - Not in PostgreSQL before v. 8.4, or any version of MySQL
- It's just like defining a one-time view for your query
- One major benefit: CTEs allow recursion
  - Recursing with CTEs is much more efficent than processing recursive data in your application

A Simple CTE Example

```
# SELECT * FROM GENERATE_SERIES(1,3)
CROSS JOIN
    (SELECT * FROM GENERATE_SERIES(8,9)) AS f;
 generate_series | generate_series
-----------------+-----------------
               1 |               8
               1 |               9
               2 |               8
               2 |               9
               3 |               8
               3 |               9
(6 rows)
```

A Simple CTE Example

```
# WITH t AS (
    SELECT * FROM GENERATE_SERIES(8,9)
)
SELECT * FROM GENERATE_SERIES(1,3)
CROSS JOIN t;
 generate_series | generate_series
-----------------+-----------------
               1 |               8
               1 |               9
               2 |               8
               2 |               9
               3 |               8
               3 |               9
(6 rows)
```

That last example was a bit cheesy, but the technique can be useful for complex queries in several parts

Recursion

Start with this:

```
# SELECT * FROM employee;
 first  |   last    | id | manager
--------+-----------+----+---------
 john   | doe       |  1 |
 fred   | rogers    |  2 |       1
 speedy | gonzales  |  3 |       1
 carly  | fiorina   |  4 |       1
 hans   | reiser    |  5 |       2
 johnny | carson    |  6 |       5
 martha | stewart   |  7 |       3
(7 rows)
```

Recursion

Recursive CTE to retrieve management hierarchy:

```
# WITH RECURSIVE t (id, managernames) AS (
    SELECT e.id, first || ' ' || last
        AS managernames
    FROM employee e WHERE manager IS NULL
        UNION ALL
    SELECT e.id,
    first || ' ' || last || ', ' || managernames
        AS managernames
    FROM employee e
    JOIN t ON (e.manager = t.id)
    WHERE manager IS NOT NULL
)
SELECT e.id, first || ' ' || last AS name,
    managernames
FROM employee e JOIN t ON (e.id = t.id);
```

Recursion

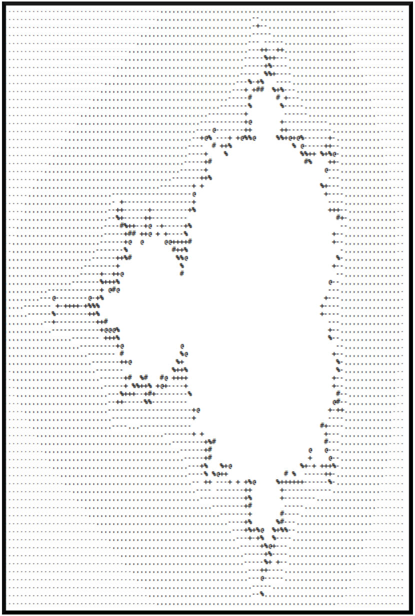
...and get this...

```
 id |       name       |  managernames
----+------------------+-----------------------------------
  1 | john doe         | john doe
  2 | fred rogers      | fred rogers, john doe
  3 | speedy gonzales  | speedy gonzales, john doe
  4 | carly fiorina    | carly fiorina, john doe
  5 | hans reiser      | hans reiser, fred rogers,
    |                  |  john doe
  6 | johnny carson    | johnny carson, hans reiser,
    |                  |  fred rogers, john doe
  7 | martha stewart   | martha stewart, speedy
    |                  |  gonzales, john doe
(7 rows)
```

Fractals in SQL

```
WITH RECURSIVE x(i) AS
(VALUES(0) UNION ALL SELECT i + 1 FROM x WHERE i < 101),
Z(Ix, Iy, Cx, Cy, X, Y, I)
AS (
    SELECT Ix, Iy, X::float, Y::float, X::float, Y::float, 0
    FROM (SELECT -2.2 + 0.031 * i, i FROM x) AS xgen(x,ix)
        CROSS JOIN
    (SELECT -1.5 + 0.031 * i, i FROM x) AS ygen(y,iy)
        UNION ALL
    SELECT
        Ix, Iy, Cx, Cy, X * X - Y * Y + Cx AS X,
        Y * X * 2 + Cy, I + 1
    FROM Z
    WHERE X * X + Y * Y < 16.0 AND I < 27),
Zt (Ix, Iy, I) AS (
    SELECT Ix, Iy, MAX(I) AS I
    FROM Z GROUP BY Iy, Ix
    ORDER BY Iy, Ix
)
SELECT array_to_string(
    array_agg(
        SUBSTRING(' .,,,-----++++%%%%@@@@#### ',
            GREATEST(I,1), 1)
    ),''
)
FROM Zt GROUP BY Iy ORDER BY Iy;
```

(yes, this query is SQL-spec compliant)



Window Functions

- Like CTEs, these are quite advanced

Window Functions

- Like CTEs, these are quite advanced
- Also unavailable in MySQL, and PostgreSQL before 8.4

Window Functions

- Like CTEs, these are quite advanced
- Also unavailable in MySQL, and PostgreSQL before 8.4
- Allow ranking, moving averages

Window Functions

- Like CTEs, these are quite advanced
- Also unavailable in MySQL, and PostgreSQL before 8.4
- Allow ranking, moving averages
- Like a set-returning aggregate function. Window functions return results for each row based on a "window" of related rows

Window Functions

If our employee table had department and salary information...

```
# SELECT first, last, salary, department
    FROM employee;
 first  |   last   | salary |   department
--------+----------+--------+----------------
 fred   | rogers   |  97000 | sales
 carly  | fiorina  |  95000 | sales
 johnny | carson   |  89000 | sales
 speedy | gonzales |  96000 | development
 hans   | reiser   |  93000 | development
 martha | stewart  |  90000 | development
 john   | doe      |  99000 | administration
(7 rows)
```

Window Functions Example

Rank employees in each department by salary

```
SELECT first, last, salary, department,
    RANK() OVER (
        PARTITION BY department
        ORDER BY salary DESC
    )
FROM employee
```

Window Functions Example

... and get this:

```
 first  |   last   | salary |   department   | rank
--------+----------+--------+----------------+------
 john   | doe      |  99000 | administration |    1
 speedy | gonzales |  96000 | development    |    1
 hans   | reiser   |  93000 | development    |    2
 martha | stewart  |  90000 | development    |    3
 fred   | rogers   |  97000 | sales          |    1
 carly  | fiorina  |  95000 | sales          |    2
 johnny | carson   |  89000 | sales          |    3
(7 rows)
```

Real, live queries

Something Simple

The slow version:

```
SELECT DISTINCT(sync) FROM bucardo.bucardo_rate
ORDER BY 1
```

The fast version:

```
SELECT name FROM sync WHERE EXISTS (
    SELECT 1 FROM bucardo_rate
    WHERE sync = name LIMIT 1)
ORDER BY 1
```

Something Simple

- The *bucardo_rate* table is huge, with few distinct values

Something Simple

- The *bucardo_rate* table is huge, with few distinct values
- finding "DISTINCT sync" requires a **long** table scan

Something Simple

- The *bucardo_rate* table is huge, with few distinct values
- finding "DISTINCT sync" requires a **long** table scan
- The *sync* table contains a list of all possible values in the *bucardo_rate.sync* column

Something Simple

- The *bucardo_rate* table is huge, with few distinct values
- finding "DISTINCT sync" requires a **long** table scan
- The *sync* table contains a list of all possible values in the *bucardo_rate.sync* column
- So instead of a big table scan, we scan the small table, and filter out values can't find in *bucardo_rate*

Something Fun

```
SELECT
    id, idname,
    COALESCE(ROUND(AVG(synctime)::NUMERIC, 1), 0) AS avgtime,
    COALESCE(SUM(total), 0) AS count
FROM (
    SELECT slavecommit,
    EXTRACT(EPOCH FROM slavecommit - mastercommit) AS synctime,
    total
    FROM bucardo.bucardo_rate
    WHERE sync = 'RO_everything' AND
    mastercommit > (NOW() - (15 + 1) * INTERVAL '1 HOUR')
) i
RIGHT JOIN (
    SELECT id, idname,
        TO_TIMESTAMP(start - start::INTEGER % 3600) AS start,
        TO_TIMESTAMP(stop - stop::INTEGER % 3600) AS stop
    FROM (
        SELECT id,
            TO_CHAR(NOW() - id * INTERVAL '1 HOUR',
                'Dy Mon DD HH:MI AM') AS idname,
            EXTRACT(EPOCH FROM NOW() - id * INTERVAL '1 HOUR') AS start,
            EXTRACT(EPOCH FROM NOW() - (id - 1) * INTERVAL '1 HOUR') AS stop
        FROM (
            SELECT GENERATE_SERIES(1, 15) AS id
        ) f
    ) g
) h ON (slavecommit BETWEEN start AND stop)
GROUP BY id, idname
ORDER BY id DESC;
```

Something Fun

- The table contains replication data

Something Fun

- The table contains replication data
  - Time of commit on master

Something Fun

- The table contains replication data
  - Time of commit on master
  - Time of commit on slave

Something Fun

- The table contains replication data
  - Time of commit on master
  - Time of commit on slave
  - Number of rows replicated

Something Fun

- The table contains replication data
  - Time of commit on master
  - Time of commit on slave
  - Number of rows replicated
- The user wants a graph of replication speed over time, given a user-determined range of time

Something Fun

We want to average replication times over a series of buckets. The first part of our query creates those buckets, based on `generate_series()`. Here we create buckets for 15 hours

```
SELECT
    id,
    TO_CHAR(NOW() - id * INTERVAL '1 HOUR',
        'Dy Mon DD HH:MI AM') AS idname,
    EXTRACT(EPOCH FROM NOW() - id *
        INTERVAL '1 HOUR') AS start,
    EXTRACT(EPOCH FROM NOW() - (id - 1) *
        INTERVAL '1 HOUR') AS stop
FROM (
    SELECT GENERATE_SERIES(1, 15) AS id
) f
```

Something Fun

This gives us:

```
 id |        idname        |       start       |       stop
----+----------------------+-------------------+-------------------
  1 | Sat Mar 14 10:23 PM | 1237091036.95657 | 1237094636.95657
  2 | Sat Mar 14 09:23 PM | 1237087436.95657 | 1237091036.95657
  3 | Sat Mar 14 08:23 PM | 1237083836.95657 | 1237087436.95657
  4 | Sat Mar 14 07:23 PM | 1237080236.95657 | 1237083836.95657
...
```

Something Fun

Make the buckets end on nice time boundaries:

```
SELECT id, idname,
    TO_TIMESTAMP(start - start::INTEGER % 3600)
        AS start,
    TO_TIMESTAMP(stop - stop::INTEGER % 3600)
        AS stop
FROM (
    -- The bucket query, shown earlier, goes here
) g
```

Something Fun

That gives us this:

```
 id |        idname        |             start             |             stop
----+----------------------+-------------------------------+-------------------------------
  1 | Sat Mar 14 10:23 PM | 2009-03-14 21:59:59.956568-06 | 2009-03-14 22:59:59.956568-06
  2 | Sat Mar 14 09:23 PM | 2009-03-14 20:59:59.956568-06 | 2009-03-14 21:59:59.956568-06
  3 | Sat Mar 14 08:23 PM | 2009-03-14 19:59:59.956568-06 | 2009-03-14 20:59:59.956568-06
  4 | Sat Mar 14 07:23 PM | 2009-03-14 18:59:59.956568-06 | 2009-03-14 19:59:59.956568-06
```

Something Fun

In an different subquery, select everything from the table of the right time period and right sync. Call this the "stats" query:

```
SELECT
    slavecommit,
    EXTRACT(EPOCH FROM slavecommit - mastercommit)
        AS synctime,
    total
FROM bucardo.bucardo_rate
WHERE
    sync = 'RO_everything' AND
    mastercommit > (NOW() - (15 + 1) *
        INTERVAL '1 HOUR')
```

Something Fun

...which gives us this:

```
         slavecommit           |     synctime      | total
-------------------------------+-------------------+--------
 2009-03-14 07:32:00.103759-06 | 5.65614098310471 |      1
 2009-03-14 07:32:04.31508-06  | 5.25827997922897 |      3
 2009-03-14 07:32:04.31508-06  | 5.25827997922897 |      5
 2009-03-14 07:32:08.700184-06 | 7.71899098157883 |      1
 2009-03-14 07:32:08.700184-06 | 8.22490698099136 |      1
 2009-03-14 07:32:12.675518-06 | 7.85176599025726 |      6
 2009-03-14 07:32:12.675518-06 | 7.15798497200012 |      6
...
```

Something Fun

Now, join the two queries:

```
SELECT
    id, idname,
    COALESCE(ROUND(AVG(synctime)::NUMERIC, 1), 0)
        AS avgtime,
    COALESCE(SUM(total), 0) AS count
FROM (
    <STATS QUERY>
)  RIGHT JOIN (
    <CALENDAR QUERY>
) ON (slavecommit BETWEEN start AND stop)
GROUP BY id, idname
ORDER BY id DESC;
```

Something Fun

...and get this:

```
 id |        idname          | avgtime | count
----+------------------------+---------+-------
 15 | Sat Mar 14 08:35 AM |     7.9 | 14219
 14 | Sat Mar 14 09:35 AM |     6.9 | 16444
 13 | Sat Mar 14 10:35 AM |     6.5 | 62100
 12 | Sat Mar 14 11:35 AM |     6.2 | 47349
 11 | Sat Mar 14 12:35 PM |       0 |     0
 10 | Sat Mar 14 01:35 PM |     4.6 | 21348
```

This is the average replication time and total replicated rows per hour. Note that this correctly returns zeroes when no rows are replicated, and still returns a value for that time slot. This prevents some amount of application-side processing.

Something Fun

That query again:

```
SELECT
    id, idname,
    COALESCE(ROUND(AVG(synctime)::NUMERIC, 1), 0) AS avgtime,
    COALESCE(SUM(total), 0) AS count
FROM (
    SELECT slavecommit,
    EXTRACT(EPOCH FROM slavecommit - mastercommit) AS synctime,
    total
    FROM bucardo.bucardo_rate
    WHERE sync = 'RO_everything' AND
    mastercommit > (NOW() - (15 + 1) * INTERVAL '1 HOUR')
) i
RIGHT JOIN (
    SELECT id, idname,
        TO_TIMESTAMP(start - start::INTEGER % 3600) AS start,
        TO_TIMESTAMP(stop - stop::INTEGER % 3600) AS stop
    FROM (
        SELECT id,
            TO_CHAR(NOW() - id * INTERVAL '1 HOUR',
                'Dy Mon DD HH:MI AM') AS idname,
            EXTRACT(EPOCH FROM NOW() - id * INTERVAL '1 HOUR') AS start,
            EXTRACT(EPOCH FROM NOW() - (id - 1) * INTERVAL '1 HOUR') AS stop
        FROM (
            SELECT GENERATE_SERIES(1, 15) AS id
        ) f
    ) g
) h ON (slavecommit BETWEEN start AND stop)
GROUP BY id, idname
ORDER BY id DESC;
```

Key Points

- Understand join types, and use them

Key Points

- Understand join types, and use them
- Know what functions and set operations your database provides

Key Points

- Understand join types, and use them
- Know what functions and set operations your database provides
- Build large queries piece by piece

Questions?