Refactoring SQL

jeremiah.peschka@gmail.com

Brent Ozar PLF
Your technology pain-relief experts.

IT'S ALL ABOUT ME

ME ME ME ME ME ME ME

CorrugatedIron

A FEATURE-RICH .NET CLIENT FOR THE RIAK KV STORE

http://www.flickr.com/photos/bixentro/2091438688/

A quick note

about compilers

http://www.flickr.com/photos/dawgbyte77/3058183665/

Modern compilers
are modular

http://www.flickr.com/photos/dawgbyte77/3058183665/

Modern compilers are modular

they are designed for modular development techniques

http://www.flickr.com/photos/dawgbyte77/3058183665/

Database compilers

are like C compilers

http://www.flickr.com/photos/dawgbyte77/3058183665/

Database compilers

you optimize by inlining

are like C compilers

http://www.flickr.com/photos/dawgbyte77/3058183665/

Before Refactory

Standardize

Testing

Refactoring

http://www.flickr.com/photos/24293932@N00/2752221871/

http://www.flickr.com/photos/noahbulgaria/305783109/

http://www.flickr.com/photos/acrider/4077303411/

Statistics are used by optimizers to determine the types of joins and reads to use.
Statistics are stored as a histogram that's divided into up to 200 segments detailing things like:

- number of rows
- histogram bounds
- distinct values

You want them because, otherwise, the database might very well scan every table on every query

http://www.flickr.com/photos/paulk/2687161030/

Unindexed tables - these always get full scans
Tables without unique indexes - duplicate data, no integrity, optimizer has to make more guesses
Only one index - are you sure you only query this table one way?
Many indexes - a lot of maintenance, could be combined to reduce index merges in memory
Many single-column indexes - as above

Before Refactory

Standardize

Testing

Refactoring

http://www.flickr.com/photos/24293932@N00/2752221871/

http://www.flickr.com/photos/oskay/265899766/

http://www.flickr.com/photos/meanestindian/536475568/

TABLE ALIASES

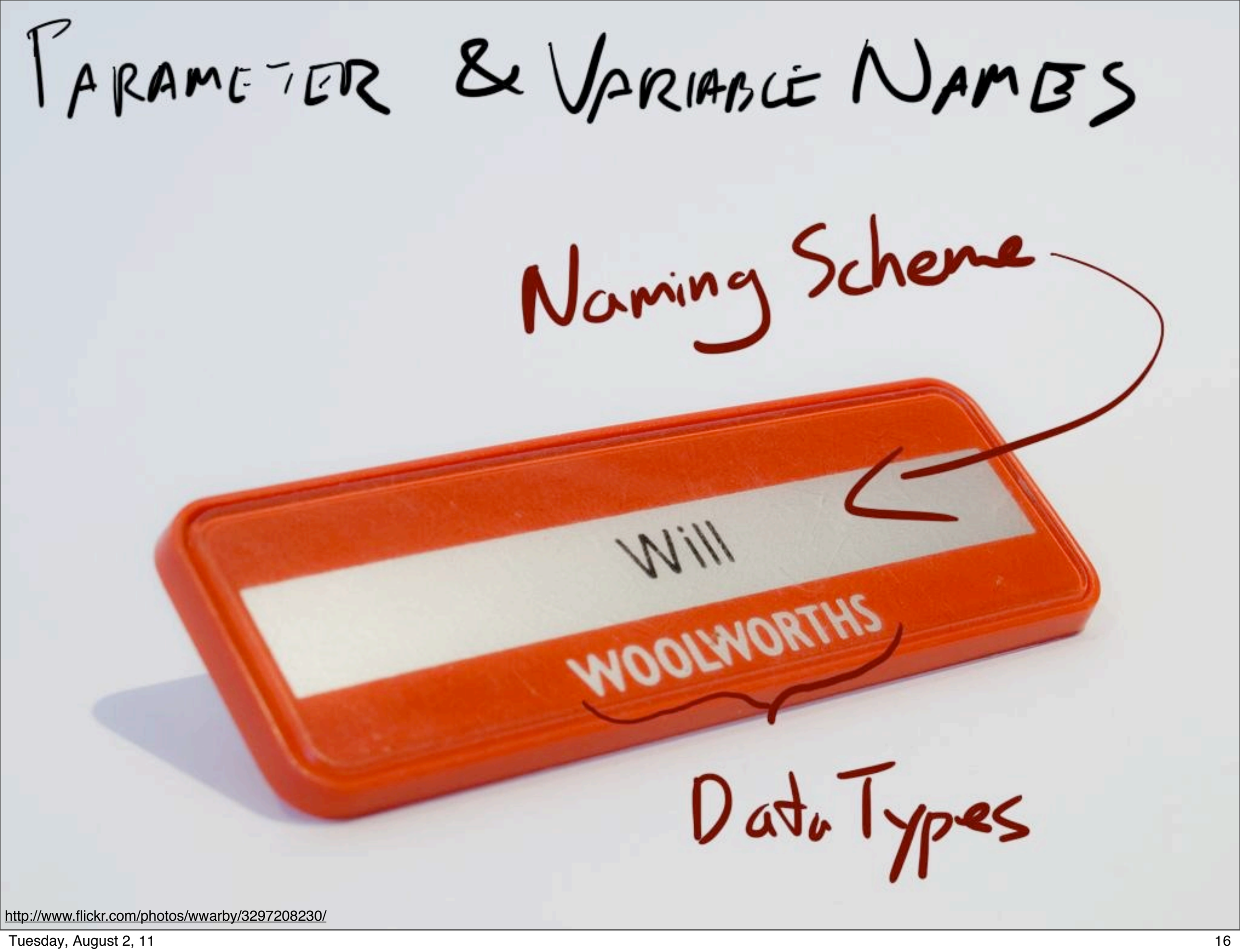
No aliases?

ABC?

Common Abbreviations?

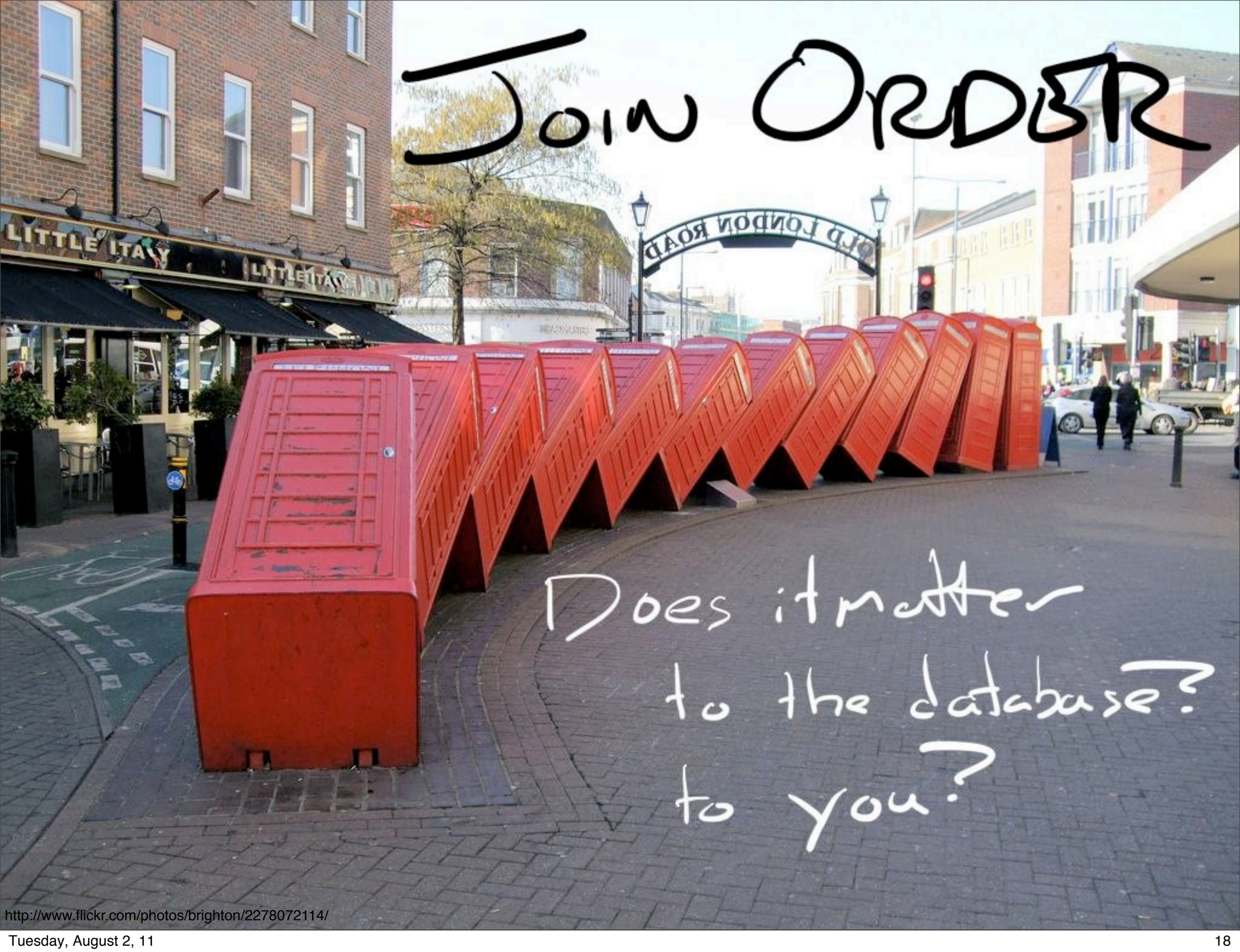
Be Consistent

Parameter & Variable Names



http://www.flickr.com/photos/wwarby/3297208230/

Be Consistent



http://www.flickr.com/photos/brighton/2278072114/

Be Consistent

I bet you've noticed a theme...

Be Consistent

Before Refactory

Standardize

Testing

Refactoring

http://www.flickr.com/photos/24293932@N00/2752221871/

http://www.flickr.com/photos/x-ray_delta_one/3812795111/

http://www.opensourcetesting.org/unit_sql.php has a list of unit test frameworks
Create set ups, tear downs, data loads, whatever.
Create an environment where you can repeatably verify that nothing changes apart from what should change.
Performance improvements shouldn't change your report outputs.

Watch Out!

Testing in dev is NOT Testing in production

http://www.flickr.com/photos/tm-tm/3107095489/

You should Test in production

Your dev database might be small. Your production database might be huge. Enhancements in test are not the same as enhancements to production.
Testing in production means that you need to have a way to test with similar hardware and data sizes (ideally the same).
Ideally you should be able to replay production workloads during your test process or use your production workloads as a benchmark for performance

Before Refactory

Standardize

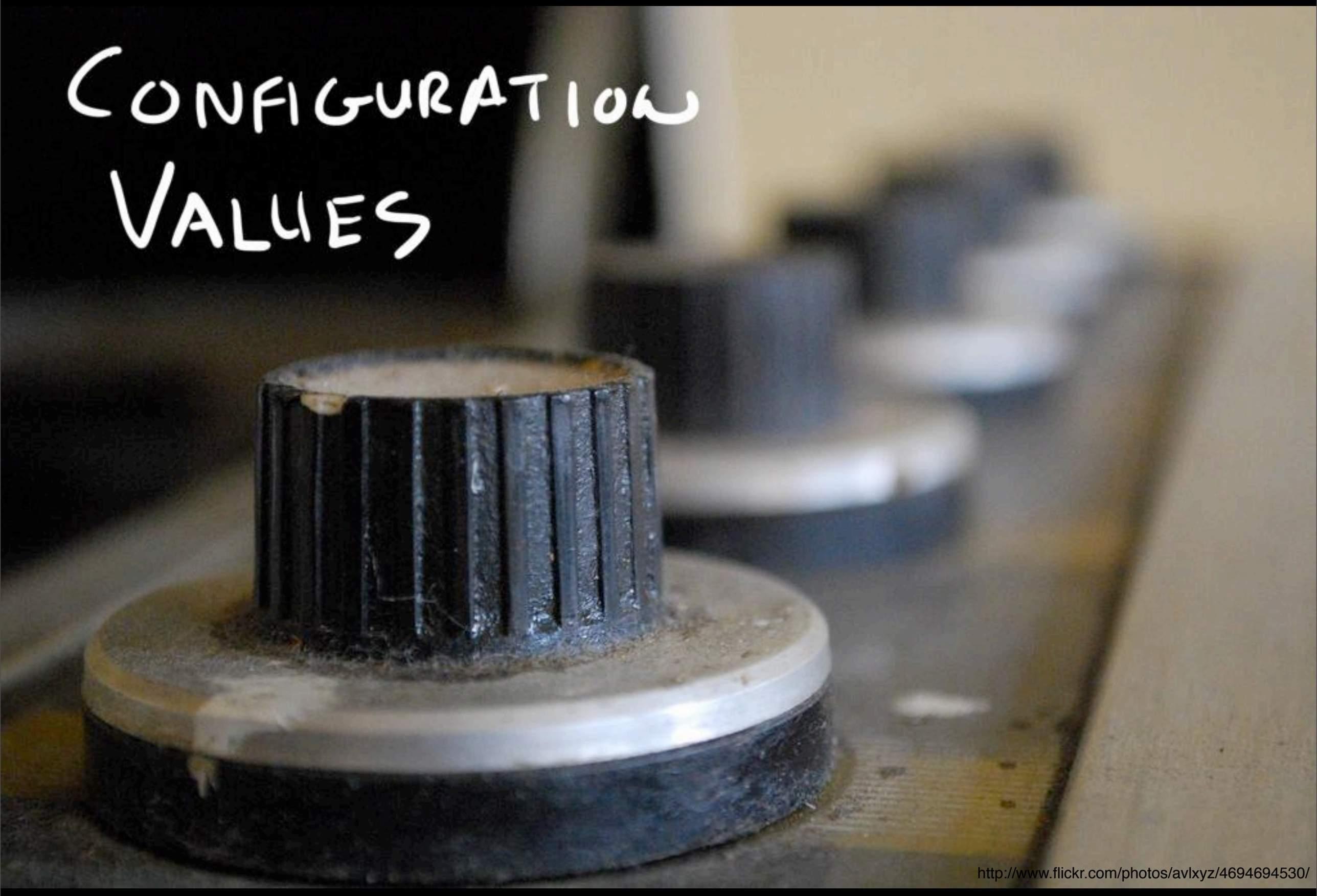
Testing

Refactoring

http://www.flickr.com/photos/24293932@N00/2752221871/

This is a sign of repeated, boilerplate, SQL.
These are places for forgetful coding, sloppy code, and maintenance nightmares.
Difficult for optimizers to produce good code
    (usually only one path will be optimized)
    (This has changed in newer versions of SQL Server, but it's still a maintenance nightmare)
Replace with modular dynamic SQL for better performance - each parameter combo used should receive an individual query plan.
Dynamic SQL is SQL that is built in the database layer (SQL writing SQL is not a form of witchcraft). This lets you keep database native data types and still have flexible code.
Easier to maintain (for some values of easier)

There is a difference between CASE and IF/ELSE - CASE is evaluated once as part of a set-based operation, it isn't a branching operation.



http://www.flickr.com/photos/avlxyz/4694694530/

Remove magic numbers and magic strings from your code
Create lookup tables - use a custom procedure like SQL Server's sp_configure

Potential problems
Can produce read contention on one table (theoretically, this table should be held in memory)
Can produce a difficult to understand "master table" structure

http://www.flickr.com/photos/21804434@N02/3984281276/

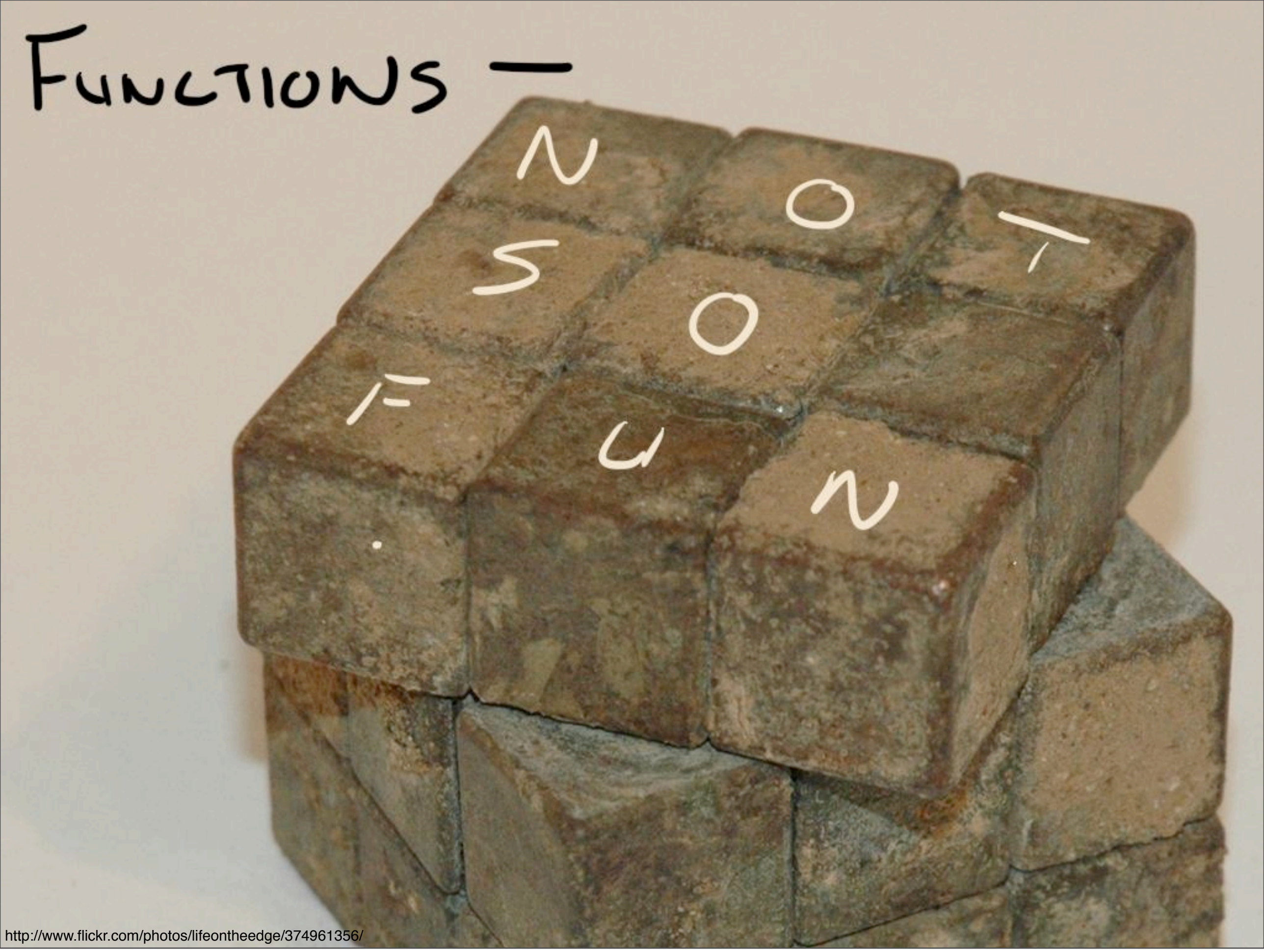
This isn't magic numbers, this is junk from the app tier.
This leads to plan bloat and difficult to debug problems.
Prepare statements using your database drivers
This will re-use the same query with named parameters, reduce plan bloat (and memory consumption), reduce code complexity, and it might even increase plan re-use (hooray for free performance).

The Horrors of Lookup Code

http://www.flickr.com/photos/pinksherbet/3561662932/

Find commonly used lookup code – user security settings, sales tax, shipping & handling.
Replace with table valued functions – load temp tables and use them where you would use lookup code
Problems
– SQL functions aren’t always high performance when inlined (use a temp table!)

Functions —



http://www.flickr.com/photos/lifeontheedge/374961356/

In the WHERE clause? that's bad. Full table scans can result from this.
Rewrite where clauses so that columns do not have functions applied to them - apply functions to constants, parameters, and variables instead.
If you need a table-valued function to be inline, consider replacing it with a CROSS APPLY or OUTER APPLY

An Example!

```
SELECT *
FROM   payment
WHERE  payment_date + '4 years'::interval > now();
```

An Example!

```
SELECT *
FROM   payment
WHERE  (payment_date + '4 years'::interval) > now();
```

An Example!

```
SELECT *
FROM   payment
WHERE  payment_date > now() - '4 years'::interval;
```

An Example!

```
SELECT *
FROM   payment
WHERE  payment_date > (now() - '4 years'::interval);
```

On PostgreSQL, the first query has a cost of 467.41 and the second has cost of 457.07.

In this case, we incur most of our cost streaming results from multiple sub tables.

http://www.flickr.com/photos/cuttlefish/4494068038/


```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
FROM CORPDATA.EMPLOYEE X
WHERE EDLEVEL >
   (SELECT AVG(EDLEVEL)
      FROM CORPDATA.EMPLOYEE
      WHERE WORKDEPT = X.WORKDEPT)
```

This can cause cursor behavior - one execution of the subquery per row.
Replace with a join in the from clause.

Another Example!

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
FROM   CORPDATA.EMPLOYEE X
WHERE  EDLEVEL >
       ( SELECT AVG(EDLEVEL)
         FROM   CORPDATA.EMPLOYEE
         WHERE  WORKDEPT = X.WORKDEPT );
```

Another Example!

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
FROM   CORPDATA.EMPLOYEE X
WHERE  EDLEVEL >
       ( SELECT AVG(EDLEVEL)
         FROM   CORPDATA.EMPLOYEE
         WHERE  WORKDEPT = X.WORKDEPT );
```

Another Example!

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
FROM   CORPDATA.EMPLOYEE X
       JOIN ( SELECT WORKDEPT,
                     AVG(EDLEVEL) AVG_EDLEVEL
              FROM   CORPDATA.EMPLOYEE
            ) e ON e.WORKDEPT = X.WORKDEPT
WHERE  X.EDLEVEL > e.AVG_EDLEVEL ;
```

Another Example!

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
FROM   CORPDATA.EMPLOYEE X
       JOIN ( SELECT WORKDEPT,
                     AVG(EDLEVEL) AVG_EDLEVEL
              FROM   CORPDATA.EMPLOYEE
              GROUP BY WORKDEPT
            ) e ON e.WORKDEPT = X.WORKDEPT
WHERE  X.EDLEVEL > e.AVG_EDLEVEL ;
```

Eliminate Repeated Patterns

http://www.flickr.com/photos/seeminglee/3990573818/

Repeated passes over the data mean repeated reads

```
SELECT SUM(a) FROM table GROUP BY x
UNION ALL
SELECT SUM(b) FROM table GROUP BY x
```

Replace these with CASE statements and CTEs

A Third Example!

```
SELECT SUM(weekday_rentals) weekday_rentals,
       SUM(weekend_rentals) weekend_rentals
FROM   ( SELECT sum(amount) weekday_rentals,
                0 weekend_rentals
           FROM   payment
           WHERE  extract(dow FROM payment_date)
                  NOT IN (6, 0)
         UNION ALL
         SELECT 0 weekday_rentals,
                SUM(amount) weekend_rentals
         FROM   payment
         WHERE  extract(dow FROM payment_date)
                IN (6, 0)
       ) x ;
```

A Third Example!

```
SELECT SUM(weekday_rentals) weekday_rentals,
       SUM(weekend_rentals) weekend_rentals
FROM   ( SELECT sum(amount) weekday_rentals,
                0 weekend_rentals
         FROM   payment
         WHERE  extract(dow FROM payment_date)
                NOT IN (6, 0)
         UNION ALL
         SELECT 0 weekday_rentals,
                SUM(amount) weekend_rentals
         FROM   payment
         WHERE  extract(dow FROM payment_date)
                IN (6, 0)
       ) x ;
```

A Third Example!

```
SELECT SUM(CASE WHEN EXTRACT(dow FROM payment_date)
                     NOT IN (6, 0) THEN amount
                ELSE 0 END) weekday_rentals,
       SUM(CASE WHEN EXTRACT(dow FROM payment_date)
                     IN (6, 0) THEN amount
                ELSE 0 END) weekend_rentals
FROM   payment ;
```

A Third Example!

```
SELECT SUM(CASE WHEN EXTRACT(dow FROM payment_date)
                     NOT IN (6, 0) THEN amount
                ELSE 0 END) weekday_rentals,
       SUM(CASE WHEN EXTRACT(dow FROM payment_date)
                     IN (6, 0) THEN amount
                ELSE 0 END) weekend_rentals
FROM   payment ;
```

On my PostgreSQL instance (9.0.3 on OS X 10.6.7) the original query had a "cost" of 888.13 and performed two sequential scans of the purchases table.

The updated query has a cost of 420.66 and only performs one sequential scan of the table. Clearly this is better.

http://www.flickr.com/photos/generated/2084287794/

Not inherently bad, but they are inherently procedural

High number of reads
Difficult to understand the query logic

Look for:
- nested loops
- cursors
- loops in application code
- high execution count

Remove cursors by
- rewriting as set-based operations
- using batch operations
- removing correlated subqueries
- modifying queries so that nested loop joins aren't used

http://www.flickr.com/photos/milanocookiez/2576674769/

http://www.flickr.com/photos/valeriebb/290711738/

Questions?

Jeremiah Peschka

@peschkaj

http://brentozar.com

jeremiah@brentozar.com

To contact me after my presentation – text 86Y to INTRO (46876)

http://www.flickr.com/photos/andreanna/2812118063/

Photo credits:

Agenda http://www.flickr.com/photos/24293932@N00/2752221871/
Beginning of the road: http://www.flickr.com/photos/noahbulgaria/305783109/
notebook: http://www.flickr.com/photos/dawgbyte77/3058183665/
all about me: http://www.flickr.com/photos/bixentro/2091438688/
Zoe's Color Histogram: http://www.flickr.com/photos/acrider/4077303411/
Indexes: http://www.flickr.com/photos/paulk/2687161030/
Standards fishlike: http://www.flickr.com/photos/oskay/265899766/
Alias/disguise: http://www.flickr.com/photos/meanestindian/536475568/
Be Consistent: http://www.flickr.com/photos/fdecomite/449818981/
Parameter and variable names: http://www.flickr.com/photos/wwarby/3297208230/
Join Order: http://www.flickr.com/photos/brighton/2278072114/
Test Your SQL: http://www.flickr.com/photos/x-ray_delta_one/3812795111/
Testing Warnings: http://www.flickr.com/photos/tm-tm/3107095489/
Branching Logic: http://www.flickr.com/photos/oedipusphinx/4425295409/
Configuration Values: http://www.flickr.com/photos/avlxyz/4694694530/
Embedded Constants: http://www.flickr.com/photos/21804434@N02/3984281276/
Horrors of Lookup Code: http://www.flickr.com/photos/pinksherbet/3561662932/
(dys)Functional: http://www.flickr.com/photos/lifeontheedge/374961356/
Correlated Subqueries: http://www.flickr.com/photos/cuttlefish/4494068038/
Eliminate Repeated Patterns: http://www.flickr.com/photos/seeminglee/3990573818/
Cursors: http://www.flickr.com/photos/generated/2084287794/
YMMV: http://www.flickr.com/photos/milanocookiez/2576674769/
This is important: http://www.flickr.com/photos/valeriebb/290711738/
Questions: http://www.flickr.com/photos/andreanna/2812118063/