# Basically, Don't Trust Anything other than the Database

## Overview

- Go over some findings that Joey came across
- Ideas on how to mitigate some of these issues
- Inspect the ORM
- Practical example, a migration

# Indexes! Why tho?

```
analytics_mixpanelevent_name_like
api_apiclient_last_ip_like
api_connectclient_apikey_like
api_ctsclient_apikey_like
api_merchanttablet_apikey_like
api_partnerclient_apikey_like
api_poslinkclient_apikey_like
api_serverapiclient_apikey_like
api_serverapiclient_name_like
auth_group_name_like
core_business_text_code_like
core_businesscategory_name_like
core_businesstag_key_like
core_connectposversion_version_string_like
core_connecttabletversion_version_string_like
core_keyword_name_like
core_network_name_like
core_permission_comment_like
core_permission_name_like
core_posvendor_name_like
core_role_comment_like
core_role_name_like
core_salesforcecontact_sf_contact_id_like
core_usertype_name_like
directory_department_name_like
```

- We've got quite a few indexes around that are “like” indexes
- Indexes for doing partial string matching with wildcards
- Besides the Django admin… we generally don't query like this
- So why do these exist?

# Django Open Source Code

```
# Fields with database column types of `varchar` and `text` need
# a second index that specifies their operator class, which is
# needed when performing correct LIKE queries outside the
# C locale. See #12234.

db_type = f.db_type(connection=self.connection)
if db_type.startswith('varchar'):
    output.append(get_index_sql('%s_%s_like' % (db_table, f.column),
                                ' varchar_pattern_ops'))
elif db_type.startswith('text'):
    output.append(get_index_sql('%s_%s_like' % (db_table, f.column),
                                ' text_pattern_ops'))
```

# It gets weirder

## South Add Index

- Generate SQL for index based on column type
- Varchar gets a regular BTree index
- Does not create a “like” index

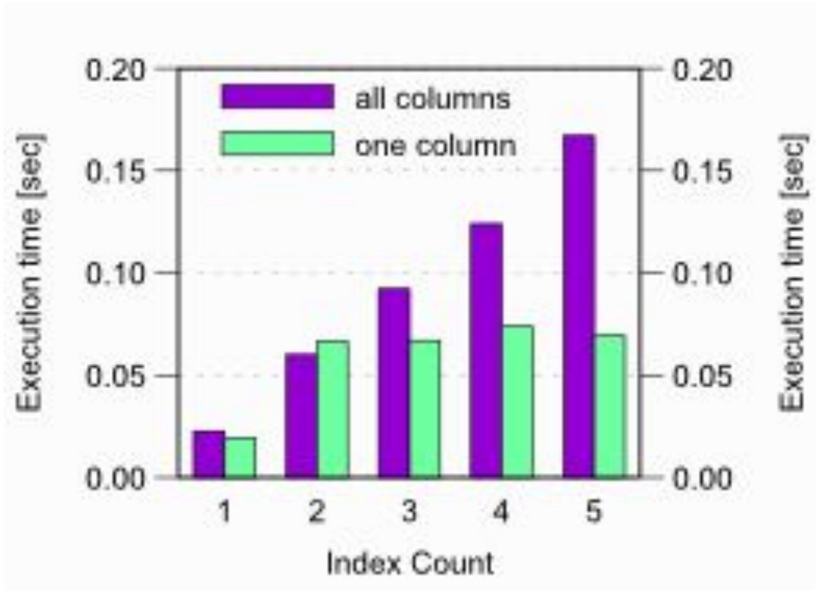
## South Add Column with Index

- Ask Django how to make the column
- Django implementation generates SQL for Indexes
- Creates the same index south would have made
- Special case for Varchar, adds additional “like” index
- ???????????

## manage.py migrate --db-dry-run --verbosity=2

```
Running migrations for core:
 - Migrating forwards to 0223_auto__add_index_business_city__add_index_business_state.
 > core:0223_auto__add_index_business_city__add_index_business_state
   = CREATE INDEX core_business_city ON core_business(city) []
   = CREATE INDEX core_business_state ON core_business(state) []
```

## Why does this matter?



By default, Django will update every column in a table when you save a model.

Every time a column is updated, indexes get updated too

(Postgres does not optimize when the data is the same)

http://use-the-index-luke.com/sql/dml/update (this site is an excellent resource)

## So... How do we get around this?

- Need more knowledge of what is happening on the SQL level for migrations
- Sometimes, Django and South have different goals than us. We should understand that and be able to catch when it does something unexpected
- If adding a migration, part of the code review process should reflect what the migration is actually doing under the hood

# Don’t Trust the ORM in Application Code either

- ORMs do not generally map 1:1 with SQL
    - Except maybe SQLAlchemy -- you’re off the hook
- ORMs usually have the N+1 problem
    - Query for object
    - Loop over N related objects; M2M for example
    - End up making N additional database hits
- ORMs generate shitty SQL
    - Follows questionable patterns
        - Ex. Only using one type of join, regardless of the query
    - models.py encouraged to follow Object oriented design, not schema design
    - Can generate good SQL, but it takes effort

## Log SQL

Settings.py

```
Logging = {

'loggers': {
        'django.db.backends': {
            'level': 'DEBUG',
            'handlers': ['console'],
        }
    }

}
```

## Know your queries

- Querysets have a query property
  - print UserProfile.objects.all().query
- Very useful from a breakpoint
  - Import pdb;pdb.set_trace()

Printing All Queries Made to Database

- from django.db import connection, connections
- print connection.queries
- print connections[“message”].queries
- reset_queries()
- If running through a unit test, may need to make sure settings.DEBUG = True

## No Downtime Migration

Adding a column with a default value

## Original Migration

```
db.add_column(u'merchant_feedactivity', 'active',
self.gf('django.db.models.fields.BooleanField)(default=True,
null=True, blank=True), keep_default=False)
```

```
ALTER TABLE "merchant_feedactivity" ADD COLUMN "active"
boolean NOT NULL DEFAULT True
```

```
ALTER TABLE "merchant_feedactivity" ALTER COLUMN "active"
TYPE boolean, ALTER COLUMN "active" SET NOT NULL, ALTER
COLUMN "active" DROP DEFAULT
```

Full table rewrite to add the default value

This kills the database

## New Migration 1

```
db.execute('ALTER TABLE "merchant_feedactivity" ADD COLUMN
"active" boolean NULL DEFAULT True')
```

- Only execute the first statement from the original migration
  - Modified to be nullable
  - By allowing the row to be null, we avoid the table rewrite
- Keep the default value on the database level
  - Writes to this table will start getting the default value

# New Migration 2

```
FeedActivity = orm["merchant.FeedActivity"]
FeedActivity.objects.filter(active__isnull=True).update(active=True)
```

- Data Migration to fill in all the existing rows that have a null in the new column
- If this was a very large table, we can update in batches
    - Manually trigger database transactions in the migration for each batch

# New Migration 3

Note:

This migration was run AFTER the deploy was finished. The previous two migrations were run BEFORE the deploy

In order for Django  to be able to handle the defaults, the new application code needs to be up on every server.

```
db.alter_column(u'merchant_feedactivity', 'active',
self.gf('django.db.models.fields.BooleanField')())
```

- Remove nullability on the column
  - Add non null
  - Has to scan the table to ensure the constraint is held
  - Pretty quick to scan the table
- Because this is using South again
  - Automatically removes the Default value from the database layer
  - Django likes to handle defaults in Application code
  - I disagree on this, but sometimes it’s good to play along with Django

## Questions?