Searching Inventory

Jacques Uber

April 19, 2013

What is Inventory?

- Infrastructure information
  - Servers
  - DNS Records
  - Datacenters
  - Racks
  - VLANS
  - Networks, etc...
  - etc...
- A good search helps a sysadmin troubleshoot

Example Questions

- "Where is server X?"
- "What is in the 10.2.4.0/24 network?"
- "Show all the servers with 'hp' and 'mozilla.com' in their name."
- "Show all the servers and DNS records that match 'foo05-10.mozilla.com'. Only visible from the Internet?"
- "Show all 'AAAA' records in the zone 'foo.mozilla.com' that also have 'firewall' in their name."

...many more

Possible Search Solutions

1. Giant case statment
   - Checkboxes selecting which types to search
   - Multiple search fields
   - Joins?

Possible Search Solutions

1. Giant case statment
   - Checkboxes selecting which types to search
   - Multiple search fields
   - Joins?
2. A search DSL (Domain Specific Language)
   - keywords (Think Google search)
   - Compile query string to SQL
   - Let the database do the hard work
   - What this talk is about.

My Goal

1. Search everything from one place

My Goal

1. Search everything from one place
   - Every *searchable* object should expose properties to be searched

My Goal

1. Search everything from one place
   - Every *searchable* object should expose properties to be searched
2. Custom searches defined in the application

My Goal

1. Search everything from one place
   - Every *searchable* object should expose properties to be searched
2. Custom searches defined in the application
3. "Fast"

My Goal

1. Search everything from one place
   - Every *searchable* object should expose properties to be searched
2. Custom searches defined in the application
3. "Fast"
   - Number of Inventory users usually very low. Doesn't need to scale

My Goal

1. Search everything from one place
   - Every *searchable* object should expose properties to be searched
2. Custom searches defined in the application
3. "Fast"
   - Number of Inventory users usually very low. Doesn't need to scale
4. Reusable

Things to consider

1. Everything is stored in a relational database

Things to consider

1. Everything is stored in a relational database
    - Fixed schema

Things to consider

1. Everything is stored in a relational database
    - Fixed schema
    - Strongly typed data and not everything is text (I.E. Ip addresses)

Things to consider

1. Everything is stored in a relational database
    - Fixed schema
    - Strongly typed data and not everything is text (I.E. Ip addresses)
    - In the case of DNS an explicit tree structure

Things to consider

1. Everything is stored in a relational database
   - Fixed schema
   - Strongly typed data and not everything is text (I.E. Ip addresses)
   - In the case of DNS an explicit tree structure
2. Use django's ORM to talk to the database

Things to consider

1. Everything is stored in a relational database
   - Fixed schema
   - Strongly typed data and not everything is text (I.E. Ip addresses)
   - In the case of DNS an explicit tree structure
2. Use django's ORM to talk to the database
3. Maybe target Django Q objects?

An Example Search Query

- "firewall db"

An Example Search Query

- "firewall db"
- The user is asking 'Show me everything that matches "firewall" and "db"'

An Example Search Query

- "firewall db"
- The user is asking 'Show me everything that matches "firewall" and "db"'
    - Note the implicit AND

An Example Search Query Cont.

- "firewall OR db"

An Example Search Query Cont.

- "firewall OR db"
- Show me everything that matches "firewall" or "db"

An Example Search Query Cont.

- "firewall OR db"
- Show me everything that matches "firewall" or "db"
- Explicit OR

An Example Search Query Cont.

- "firewall db OR host2 console"

An Example Search Query Cont.

- "firewall db OR host2 console"
- Show me everything that matches "firewall" and "db" or "host2" and "console"

An Example Search Query Cont.

- "firewall db OR host2 console"
- Show me everything that matches "firewall" and "db" or "host2" and "console"
  - ('firewall AND 'db') OR ('host2' AND 'console')

An Example Search Query Cont.

- "(firewall db OR host2 !console) AND zone=:dev.mozilla.com view=:public"

An Example Search Query Cont.

- "(firewall db OR host2 !console) AND zone=:dev.mozilla.com view=:public"
- Show me everything that matches "firewall" and "db" or "host2" and doesn't contain "console", and it must also be a DNS record in the dev.mozilla.com DNS zone, and in the public view.

An Example Search Query Cont.

- "(firewall db OR host2 !console) AND zone=:dev.mozilla.com view=:public"
- Show me everything that matches "firewall" and "db" or "host2" and doesn't contain "console", and it must also be a DNS record in the dev.mozilla.com DNS zone, and in the public view.
  - Things are getting complex.

An Example Search Query Cont.

- "(firewall db OR host2 !console) AND zone=:dev.mozilla.com view=:public"
- Show me everything that matches "firewall" and "db" or "host2" and doesn't contain "console", and it must also be a DNS record in the dev.mozilla.com DNS zone, and in the public view.
  - Things are getting complex.
  - This looks like SQL!

An Example Search Query Cont.

- "(firewall db OR host2 !console) AND zone=:dev.mozilla.com view=:public"
- Show me everything that matches "firewall" and "db" or "host2" and doesn't contain "console", and it must also be a DNS record in the dev.mozilla.com DNS zone, and in the public view.
  - Things are getting complex.
  - This looks like SQL!
  - Note that this search only returns DNS records

An Example Search Query Cont.

- "(firewall db OR host2 !console) AND zone=:dev.mozilla.com view=:public"
- Show me everything that matches "firewall" and "db" or "host2" and doesn't contain "console", and it must also be a DNS record in the dev.mozilla.com DNS zone, and in the public view.
  - Things are getting complex.
  - This looks like SQL!
  - Note that this search only returns DNS records
    - Some types of objects are not appropriate for certain searches

Target Django Querysets

- (Deap Breath) Take a step back, start with a query strings with only one term

Target Django Querysets

- (Deap Breath) Take a step back, start with a query strings with only one term
- Consider the query string "firewall"

Target Django Querysets

- (Deap Breath) Take a step back, start with a query strings with only one term
- Consider the query string "firewall"
- Simply return anything that matches "firewall"

Target Django Querysets

- (Deap Breath) Take a step back, start with a query strings with only one term
- Consider the query string "firewall"
- Simply return anything that matches "firewall"
- How would we use a django Queryset to do this?

Target Django Querysets

- (Deap Breath) Take a step back, start with a query strings with only one term
- Consider the query string "firewall"
- Simply return anything that matches "firewall"
- How would we use a django Queryset to do this?

```
>>> q = Q(hostname__icontains='firewall')
>>> type(q)
<class 'django.db.models.query_utils.Q'>
>>> System.objects.filter(q)
[<System: firewall.db.mozilla.org>]
>>>
```

Target Django Querysets Cont.

- We want to search more than one column (or model attribute)

Target Django Querysets Cont.

- We want to search more than one column (or model attribute)
- For example, `class System` has many attributes that might match a given search term

Target Django Querysets Cont.

- We want to search more than one column (or model attribute)
- For example, `class System` has many attributes that might match a given search term
- Search all attributes of `class System` and filter on the text "firewall":

Target Django Querysets Cont.

- We want to search more than one column (or model attribute)
- For example, `class System` has many attributes that might match a given search term
- Search all attributes of `class System` and filter on the text "firewall":

```
def build_filter(filter_value, fields):
    final_filter = Q()
    for field in fields:
        final_filter = final_filter | Q(
            **{"{0}__icontains".format(field): filter_value}
        )
    return final_filter

System.search_fields = ['hostname', 'oob_ip', ... ]
q = build_filter('firewall', System.search_fields)

System.objects.filter(q)
```

Target Django Querysets Cont.

- A goal is to search everything

Target Django Querysets Cont.

- A goal is to search everything
- Map `build_filter` over every *searchable* type

Target Django Querysets Cont.

- A goal is to search everything
- Map `build_filter` over every *searchable* type
  - Use each model's `search_fields` attribute to do this dynamically.

Target Django Querysets Cont.

- A goal is to search everything
- Map `build_filter` over every *searchable* type
  - Use each model's `search_fields` attribute to do this dynamically.

```
searchables = (
    ('A', AddressRecord),
    ('CNAME', CNAME),
    ('MX', MX),
    ('SYSTEM', System),
    ...
    ...
)

def compile_Q(filter_value):
    result = []
    for name, Klass in searchables:
        result.append(build_filter(filter_value, Klass.search_fields))
    return result
```

Target Django Querysets: Operators

- `compile_Q` returns a list of querysets that can be used to filter Django models (the classes contained in `searchables`)

Target Django Querysets: Operators

- `compile_Q` returns a list of querysets that can be used to filter Django models (the classes contained in `searchables`)
- We want to logically combine these lists of querysets

Target Django Querysets: Operators

- `compile_Q` returns a list of querysets that can be used to filter Django models (the classes contained in `searchables`)
- We want to logically combine these lists of querysets
- Think vector operations:

Target Django Querysets: Operators

- `compile_Q` returns a list of querysets that can be used to filter Django models (the classes contained in `searchables`)
- We want to logically combine these lists of querysets
- Think vector operations:

$$\begin{bmatrix} result_1 \\ result_2 \\ \dots \\ result_n \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \\ \dots \\ q_n \end{bmatrix} \& \begin{bmatrix} p_1 \\ p_2 \\ \dots \\ p_n \end{bmatrix}$$

Supported Q operators:

- AND (&)
- OR (|)
- NOT (~)

Operators: An Example

- "host1 OR (firewall AND db)"

Operators: An Example

- "host1 OR (firewall AND db)"

```
def OR(left_q_sets, right_q_sets):
    result = []
    for ql, qr in zip(left_q_sets, right_q_sets):
        result.append(ql | qr)
    return result

def AND(left_q_sets, right_q_sets):
    result = []
    for ql, qr in zip(left_q_sets, right_q_sets):
        result.append(ql & qr)
    return result

result = OR(
    compile_Q("host1"),
    AND(compile_Q("firewall"), compile_Q("db"))
)
```

Front End

- We have the builing blocks (`compile_Q`) and operators to compile queries

Front End

- We have the builing blocks (`compile_Q`) and operators to compile queries
- We need to interpret user search queries dynamically

Front End

- We have the builing blocks (`compile_Q`) and operators to compile queries
- We need to interpret user search queries dynamically
- It is a solved problem! Yay!

Parsley

- Parsley: "Parsley is a pattern matching and parsing tool for Python programmers."

Parsley

- Parsley: "Parsley is a pattern matching and parsing tool for Python programmers."
- http://parsley.readthedocs.org/

DSL Part 1

```
ws = ' '*
wss = ' '+
not_ws = :c ?(c not in (' ', '\t')) -> c
letter = :c ?('a' <= c <= 'z' or 'A' <= c <= 'Z') -> c
special = '_' | '.' | '-' | ':' | ','

# Lexical Operators
NOT = '!'
AND = <letter+>:and_ ?(and_ == 'AND') -> self.AND_op()
OR = <letter+>:or_ ?(or_ == 'OR') -> self.OR_op()

# Directive
EQ = '=:'
d_lhs = letter | '_'
d_rhs = letterOrDigit | special | '/'
DRCT = <d_lhs+>:d EQ <d_rhs+>:v -> self.directive(d, v)

# Regular Expression
RE = '/' <(not_ws)+>:r -> self.regexpr(r)

# Regular text
text = (~OR ~AND ~NOT <(letterOrDigit | special )+>:t) -> t
TEXT = <text+>:t -> self.text(t)
```

DSL Part 2

```
# DSF (Device Specific Filters)
DSF = DRCT | RE | TEXT

# An atmon
atom = DSF | parens

value = NOT ws atom:a -> self.NOT_op()(a)
| atom

# Parens
parens = '(' ws expr:e ws ')' -> e

# Operators Precidence
# 1) i_and
# 2) 2_and
# 3) e_or

# x AND y <-- Explicit AND
e_and = AND:op wss value:v -> (op, v)

# x y <-- Implicit AND
i_and = (' '+ ~OR ~AND) value:v -> (self.AND_op(), v)

# x OR y <-- Explicit OR
e_or = OR:op wss expr_2:v -> (op, v)
```

DSL Part 3

```
# Compile
expr = expr_2:left ws e_or*:right -> self.compile(left, right)
expr_2 = expr_3:left ws e_and*:right -> self.compile(left, right)
expr_3 = value:left i_and*:right -> self.compile(left, right)
```

DSL

Parsley does all the hard stuff:

DSL

Parsley does all the hard stuff:

```
[uberj@box ~]$ python invdsl.py 'firewall (baz OR fiz)'
(firewall AND (baz OR fiz))
```

DSL

Parsley does all the hard stuff:

```
[uberj@box ~]$ python invdsl.py 'firewall (baz OR fiz)'
(firewall AND (baz OR fiz))

[uberj@box ~]$ python invdsl.py 'firewall baz OR fiz'
((firewall AND baz) OR fiz)
```

So Far...

- The search is kind of like the SELECT clause in SQL

So Far...

- The search is kind of like the SELECT clause in SQL
- Good enough for just searching all your django models

So Far...

- The search is kind of like the SELECT clause in SQL
- Good enough for just searching all your django models
- General enough to tack on some more interesting queries

Directives

- A search term corresponds to a list of Qs that filter *searchable* objects based on that term

Directives

- A search term corresponds to a list of Qs that filter *searchable* objects based on that term
  - We have only seen filters like:

Directives

- A search term corresponds to a list of Qs that filter *searchable* objects based on that term
  - We have only seen filters like:

```
filter_list = [
    ...
    Q(field_name__icontains="keywordstring") | ...
    ...
]
```

Directives Cont.

- The list of Q objects can be built to do a more complex filter

Directives Cont.

- The list of Q objects can be built to do a more complex filter

```
searchables = (
    ('A', AddressRecord),
    ('CNAME', CNAME),
    ('MX', MX),
    ('SYSTEM', System),
    ...
    ...
)

def build_rdtype_qsets(rdtype):
    rdtype = rdtype.upper()
    select = Q(pk__gt=-1)
    no_select = Q(pk__lte=-1)
    result = []
    for name, Klass in searchables:
        if name == rdtype:
            result.append(select)
        else:
            result.append(no_select)
    return result
```

Directives Cont.

- The list of Q objects can be built to do a more complex filter

```
searchables = (
    ('A', AddressRecord),
    ('CNAME', CNAME),
    ('MX', MX),
    ('SYSTEM', System),
    ...
    ...
)

def build_rdtype_qsets(rdtype):
    rdtype = rdtype.upper()
    select = Q(pk__gt=-1)
    no_select = Q(pk__lte=-1)
    result = []
    for name, Klass in searchables:
        if name == rdtype:
            result.append(select)
        else:
            result.append(no_select)
    return result
```

We can now use `build_type_qset('MX')` to give us a list of Q objects that when mapped onto *searchables* will only return 'MX' records.

Directives Cont.

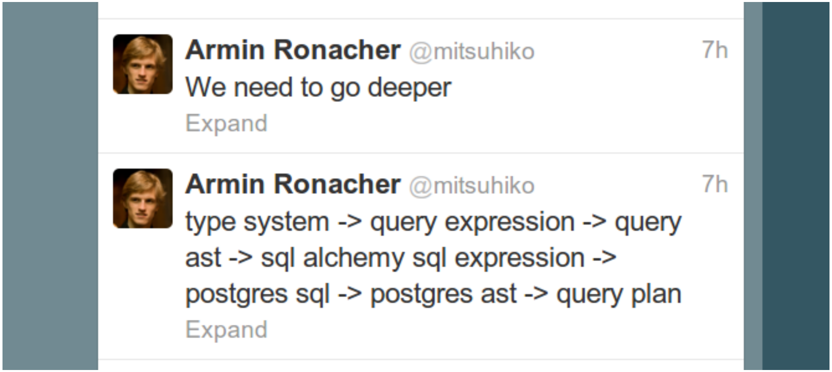
- Filter by DNS view

Directives Cont.

- Filter by DNS view

```
def build_view_qsets(view_name):
    """Filter based on DNS views."""
    view_name = view_name.lower() # Let's get consistent
    try:
        view = View.objects.get(name=view_name)
    except ObjectDoesNotExist:
        raise BadDirective("'{0}' isn't a valid view.".format(view_name))
    view_filter = Q(views=view) # This will slow queries down due to
         joins
    q_sets = []
    select = Q(pk__gt=-1)
    for name, Klass in searchables:
        if name == 'SOA':
            q_sets.append(select) # SOA's are always public and private
        elif hasattr(Klass, 'views'):
            q_sets.append(view_filter)
        else:
            q_sets.append(None)
    return q_sets
```

Making things ~~worse~~ better!



Armin Ronacher @mitsuhiko 7h

We need to go deeper

Expand

Armin Ronacher @mitsuhiko 7h

type system -> query expression -> query ast -> sql alchemy sql expression -> postgres sql -> postgres ast -> query plan

Expand

Demo

- Live Demo!
- Let's hope it works!