Evolution of a query language...
REST API
SQL-ish
Vaguely Familiar

```sql
select percentile(90, value) from cpu
where time > now() - 1d and
  "host" = 'serverA'
group by time(10m)
```
0.8 -> 0.9
Breaking API change, addition of tags
Functional or SQL?
Afraid to switch...
Mathematics across measurements #3552

Open srfraser opened this issue on Aug 4, 2015 · 90 comments
Allow DISTINCT function to operate on tags #3880

TechniclabErdmann opened this issue on Aug 28, 2015 · 80 comments
[feature request] Support month and year as duration unit
#3991

ghost opened this issue on Sep 4, 2015 · 47 comments
Feature Request: DatePart in InfluxQL #6723

mvadu opened this issue on May 25, 2016 · 4 comments
Wire up SORDER #1819

Open pauldix opened this issue on Mar 2, 2015 · 26 comments
[feature request] support for HAVING clause #5266

Open beckettsean opened this issue on Jan 4, 2016 · 21 comments
[[feature collection]] requested Functions and query operators #5930

beckettsean opened this issue on Mar 7, 2016 · 68 comments
Difficult to improve & change
It’s not SQL!
Kapacitor
Fall of 2015
Kapacitor’s TICKscript

stream
   | from()
   |   .database('telegraf')
   |   .measurement('cpu')
   |   .groupBy(*)
| window()
   |   .period(5m)
   |   .every(5m)
   |   .align()
| mean('usage_idle')
   |   .as('usage_idle')
| influxDBOut()
   |   .database('telegraf')
   |   .retentionPolicy('autogen')
   |   .measurement('mean_cpu_idle')
   |   .precision('s')
Hard to debug
Steep learning curve
Not Recomposable
Second Language
Rethinking Everything
Kapacitor is Background Processing
Stream or Batch
InfluxDB is batch interactive
IFQL and unified API

Building towards 2.0
Project Goals

[Image: A goal post on a foggy soccer field]

Photo by Glen Carrie on Unsplash
One Language to Unite!
Feature Velocity
Decouple storage from compute
Iterate & deploy more frequently
Scale independently
Amazon Athena

Start querying data instantly. Get results in seconds. Pay only for the queries you run.

Get Started with Amazon Athena
Decouple language from engine
```json
{
  "operations": [
    {
      "id": "select0",
      "kind": "select",
      "spec": {
        "database": "foo",
        "hosts": null
      }
    },
    {
      "id": "where1",
      "kind": "where",
      "spec": {
        "expression": {
          "root": {
            "type": "binary",
            "operator": "and",
            "left": {
              "type": "binary",
              "operator": "and",
              "left": {
                "type": "binary",
                "operator": "==",
                "left": {
                  "type": "reference",
                  "name": "_measurement",
                  "kind": "tag"
                },
                "right": {
                  "type": "stringLiteral",
                  "value": "cpu"
                }
              },
              "right": {
                "type": "stringLiteral",
                "value": "cpu"
              }
            }
          }
        }
      }
    }
  ]
}
```
A Data Language
Design Philosophy
UI for Many

because no one wants to actually write a query
Readability
over terseness
Flexible
add to language easily
Testable

new functions and user queries
Easy to Contribute

inspiration from Telegraf
Code Sharing & Reuse

no code > code
A few examples
// get the last value written for anything from a given host
from(db: "mydb")
  |> filter(fn: (r) => r["host"] == "server0")
  |> last()
// get the last value written for anything from a given host
from(db: "mydb")
  |> filter(fn: (r) => r["host"] == "server0")
  |> last()

Result: _result

<table>
<thead>
<tr>
<th>_time</th>
<th>_field</th>
<th>_measurement</th>
<th>host</th>
<th>region</th>
<th>_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-02-12T15:53:00.000000000Z</td>
<td>usage_system</td>
<td>cpu</td>
<td>server0</td>
<td>east</td>
<td>60.6284</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>_time</th>
<th>_field</th>
<th>_measurement</th>
<th>host</th>
<th>region</th>
<th>_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-02-12T15:53:00.000000000Z</td>
<td>usage_user</td>
<td>cpu</td>
<td>server0</td>
<td>east</td>
<td>39.3716</td>
</tr>
</tbody>
</table>
from(db: "mydb")
  |> filter(fn: (r) =>
       r["host"] == "server0" and
       r["_measurement"] == "cpu" and
       r["_field"] == "usage_user")
  |> range(start:-1m)
// get the last minute of data from a specific measurement & field & host
from(db: "mydb")
  |> filter(fn: (r) =>
    r["host"] == "server0" and
    r["_measurement"] == "cpu" and
    r["_field"] == "usage_user")
  |> range(start:-1m)

<table>
<thead>
<tr>
<th>_time</th>
<th>_field</th>
<th>_measurement</th>
<th>host</th>
<th>region</th>
<th>_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-02-12T16:01:50.000000000Z</td>
<td>usage_user</td>
<td>cpu</td>
<td>server0</td>
<td>east</td>
<td>50.549</td>
</tr>
<tr>
<td>2018-02-12T16:02:00.000000000Z</td>
<td>usage_user</td>
<td>cpu</td>
<td>server0</td>
<td>east</td>
<td>35.4458</td>
</tr>
<tr>
<td>2018-02-12T16:02:10.000000000Z</td>
<td>usage_user</td>
<td>cpu</td>
<td>server0</td>
<td>east</td>
<td>30.0493</td>
</tr>
<tr>
<td>2018-02-12T16:02:20.000000000Z</td>
<td>usage_user</td>
<td>cpu</td>
<td>server0</td>
<td>east</td>
<td>44.3378</td>
</tr>
<tr>
<td>2018-02-12T16:02:30.000000000Z</td>
<td>usage_user</td>
<td>cpu</td>
<td>server0</td>
<td>east</td>
<td>11.1584</td>
</tr>
<tr>
<td>2018-02-12T16:02:40.000000000Z</td>
<td>usage_user</td>
<td>cpu</td>
<td>server0</td>
<td>east</td>
<td>46.712</td>
</tr>
</tbody>
</table>
// get the mean in 10m intervals of last hour
from(db: "mydb")
|> filter(fn: (r) =>
    r["host"] == "server0" and
    r["_measurement"] == "cpu")
|> range(start:-1h)
|> window(every:15m)
|> mean()

Result: _result
Block: keys: [_field, _measurement, host, region] bounds: [2018-02-12T15:05:06.708945484Z, 2018-02-12T16:05:06.708945484Z)
|_time           |_field     |_measurement |host           |region                  |_value
-----------------|------------|-------------|---------------|------------------------|------------------------
2018-02-12T15:28:41.128654848Z           usage_user |cpu          |server0        |east                     |50.72841444444444
2018-02-12T15:43:41.128654848Z           usage_user |cpu          |server0        |east                     |51.19163333333333
2018-02-12T15:18:41.128654848Z           usage_user |cpu          |server0        |east                     |45.50910882352940
2018-02-12T15:58:41.128654848Z           usage_user |cpu          |server0        |east                     |49.65145555555555
2018-02-12T16:05:06.708945484Z           usage_user |cpu          |server0        |east                     |46.41292368421052

Block: keys: [_field, _measurement, host, region] bounds: [2018-02-12T15:05:06.708945484Z, 2018-02-12T16:05:06.708945484Z)
|_time           |_field     |_measurement |host           |region                  |_value
-----------------|------------|-------------|---------------|------------------------|------------------------
2018-02-12T15:28:41.128654848Z           usage_system |cpu          |server0        |east                     |49.27158555555556
2018-02-12T15:58:41.128654848Z           usage_system |cpu          |server0        |east                     |50.34854444444444
2018-02-12T16:05:06.7089454842           usage_system |cpu          |server0        |east                     |53.58707631578949
2018-02-12T15:13:41.128654848Z           usage_system |cpu          |server0        |east                     |54.49089117647058
2018-02-12T15:43:41.1286548482           usage_system |cpu          |server0        |east                     |48.80836666666666
Elements of IFQL
// get the last 1 hour written for anything from a given host
from(db: "mydb")
    |> filter(fn: (r) => r["host"] == "server0")
    |> range(start:-1m)
// get the last 1 hour written for anything from a given host
from(db: "mydb")
|> filter(fn: (r) => r["host"] == "server0")
|> range(start:-1m)
// get the last 1 hour written for anything from a given host
from(db: "mydb")
|> filter(fn: (r) => r["host"] == "server0")
|> range(start: -1m)
// get the last 1 hour written for anything from a given host
from(db: "mydb")
  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1m)

pipe forward operator
// get the last 1 hour written for anything from a given host
from(db: "mydb")
  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1m)

named parameters only!
Readability
Flexibility
Functions have inputs & outputs
Testability
Builder
// get the last 1 hour written for anything from a given host
from(db: "mydb")
|> filter(fn: (r) => r["host"] == "server0")
|> range(start:-1m)

no input
// get the last 1 hour written for anything from a given host
from(db:"mydb")
|> filter(fn: (r) => r["host"] == "server0")
|> range(start:-1m)

output is entire db
// get the last 1 hour written for anything from a given host
from(db: "mydb")
  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1m)

pipe that output to filter
Filter function input

// get the last 1 hour written for anything from a given host
from(db: "mydb")
  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1m)

anonymous filter function
input is a single record

{"_measurement":"cpu", ",_field":"usage_user", ",host":"server0", ",region":"west", ",_value":23.2}
Filter function input

// get the last 1 hour written for anything from a given host
from(db: "mydb")
  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1m)

A record looks like a flat object or row in a table

{"_measurement":"cpu", "_field":"usage_user", "host":"server0", "region":"west", "_value":23.2}
// get the last 1 hour written for anything from a given host
from(db: "mydb")
  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1m)

{"_measurement":"cpu", "_field":"usage_user", "host":"server0", "region":"west", "_value":23.2}
Record Properties

// get the last 1 hour written for anything from a given host
from(db:"mydb")
  |> filter(fn: (r) => r.host == "server0")
  |> range(start:-1m)

same as before

{"_measurement":"cpu", "_field":"usage_user", "host":"server0", "region":"west", "_value":23.2}
Special Properties

```plaintext
from(db: "mydb")
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> range(start:-1m)
  |> max()
```

{"_measurement":"cpu", ",field":"usage_user", "host":"server0", "region":"west", "value":23.2}
Special Properties

from(db: "mydb")
    |> filter(fn: (r) =>
        r["host"] == "server0" and
        r._measurement == "cpu" and
        r._field == "usage_user")
    |> range(start:-1m)
    |> max()

{"_measurement":"cpu", "_field":"usage_user", "host":"server0", "region":"west", "_value":23.2}
from(db: "mydb")
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> range(start:-1m)
  |> max()

{"_measurement":"cpu", ",_field":"usage_user", "host":"server0", "region":"west", ",_value":23.2}
Special Properties

```plaintext
from(db:"mydb")
  | > filter(fn: (r) =>
     r["host"] == "server0" and
     r["_measurement"] == "cpu" and
     r["_field"] == "usage_user" and
     r["_value"] > 50.0)
  | > range(start:-1m)
  | > max()
```

_value exists in all series

```
{"_measurement":"cpu", "_field":"usage_user", "host":"server0", "region":"west", "_value":23.2}
```
// get the last 1 hour written for anything from a given host
from(db: "mydb")

  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1m)

filter function output is a boolean to determine if record is in set
// get the last 1 hour written for anything from a given host
from(db: "mydb")
|> filter(fn: (r) => r["host"] == "server0")
|> range(start:-1m)
Filter Boolean Logic

// get the last 1 hour written for anything from a given host
from(db: "mydb")
|> filter(fn: (r) => (r["host"] == "server0" or
  r["host"] == "server1") and
  r["_measurement"] == "cpu")
|> range(start:-1m)

parens for precedence
```javascript
// get the last 1 hour written for anything from a given host
from(db: "mydb")
    |> filter(fn: (r) => {return r["host"] == "server0"})
    |> range(start:-1m)
```

long hand function definition
// get the last 1 hour written for anything from a given host
from(db: "mydb")
  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1m)

filter output is set of data matching filter function
// get the last 1 hour written for anything from a given host
from(db: "mydb")
  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1m)

piped to range
which further filters by a time range
// get the last 1 hour written for anything from a given host
from(db: "mydb")
  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1m)

range output is the final query result
Function Isolation

(but the planner may do otherwise)
Does order matter?

```
from(db: "mydb")
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> range(start:-1m)
  |> max()
```

```
from(db: "mydb")
  |> range(start:-1m)
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> max()
```
Does order matter?

```
from(db: "mydb")
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> range(start:-1m)
  |> max()
```
Does order matter?

```
from(db: "mydb")
| > filter(fn: (r) =>
  r["host"] == "server0" and
  r["_measurement"] == "cpu" and
  r["_field"] == "usage_user")
| > range(start:-1m)
| > max()
```

```
from(db: "mydb")
| > range(start:-1m)
| > filter(fn: (r) =>
  r["host"] == "server0" and
  r["_measurement"] == "cpu" and
  r["_field"] == "usage_user")
| > range(start:-1m)
| > max()
```

results the same

Result: _result
Block: keys: ["_field", "_measurement", "host", "region"] bounds: [2018-02-12T17:52:02.322301856Z, 2018-02-12T17:53:02.322301856Z]

<table>
<thead>
<tr>
<th>_time</th>
<th>_field</th>
<th>_measurement</th>
<th>host</th>
<th>region</th>
<th>_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-02-12T17:53:02.322301856Z</td>
<td>usage_user</td>
<td>cpu</td>
<td>server0</td>
<td>east</td>
<td>97.3174</td>
</tr>
</tbody>
</table>
Does order matter?

from(db: "mydb")
|> filter(fn: (r) =>
| r["host"] == "server0" and
| r["_measurement"] == "cpu" and
| r["_field"] == "usage_user")
|> range(start:-1m)
|> max()

is this the same as the top two?

from(db: "mydb")
|> filter(fn: (r) =>
| r["host"] == "server0" and
| r["_measurement"] == "cpu" and
| r["_field"] == "usage_user")
|> range(start:-1m)
|> max()
Does order matter?

```
from(db: "mydb")
  | > filter(fn: (r) =>
    r["host"] == "server0" and
    r["_measurement"] == "cpu" and
    r["_field"] == "usage_user")
  | > range(start:-1m)
  | > max()
```

```
from(db: "mydb")
  | > range(start:-1m)
  | > filter(fn: (r) =>
    r["host"] == "server0" and
    r["_measurement"] == "cpu" and
    r["_field"] == "usage_user")
  | > max()
```

Moving `max` to here changes semantics.

```
from(db: "mydb")
  | > filter(fn: (r) =>
    r["host"] == "server0" and
    r["_measurement"] == "cpu" and
    r["_field"] == "usage_user")
  | > max()
  | > range(start:-1m)
```
Does order matter?

from(db: "mydb")
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> range(start:-1m)
  |> max()

here it operates on only the last minute of data

from(db: "mydb")
  |> range(start:-1m)
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> max()

from(db: "mydb")
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> max()
  |> range(start:-1m)
Does order matter?

from(db: "mydb")
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> range(start:-1m)
  |> max()

from(db: "mydb")
  |> range(start:-1m)
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> max()
Does order matter?

```
from(db: "mydb")
| > filter(fn: (r) =>
|   r["host"] == "server0" and
|   r["_measurement"] == "cpu" and
|   r["_field"] == "usage_user")
| > range(start:-1m)
| > max()

then that result
is filtered down to
the last minute
(which will likely be empty)

from(db: "mydb")
| > range(start:-1m)
| > filter(fn: (r) =>
|   r["host"] == "server0" and
|   r["_measurement"] == "cpu" and
|   r["_field"] == "usage_user")
| > max()
```

```
from(db: "mydb")
| > filter(fn: (r) =>
|   r["host"] == "server0" and
|   r["_measurement"] == "cpu" and
|   r["_field"] == "usage_user")
| > max()
| > range(start:-1m)
```
Planner Optimizes
maintains query semantics
from(db: "mydb")
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> range(start: -1m)
  |> max()

from(db: "mydb")
  |> range(start: -1m)
  |> filter(fn: (r) =>
      r["host"] == "server0" and
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user")
  |> max()
Optimization

```plaintext
from(db: "mydb")
    |> filter(fn: (r) =>
            r["host"] == "server0" and
            r["_measurement"] == "cpu" and
            r["_field"] == "usage_user")
    |> range(start:-1m)
    |> max()
```

```plaintext
from(db: "mydb")
    |> range(start:-1m)
    |> filter(fn: (r) =>
            r["host"] == "server0" and
            r["_measurement"] == "cpu" and
            r["_field"] == "usage_user")
    |> max()
```

this is more efficient
Optimization

```sql
from(db: "mydb")
    |> filter(fn: (r) =>
        r["host"] == "server0" and
        r["_measurement"] == "cpu" and
        r["_field"] == "usage_user"
    )
    |> range(start:-1m)
    |> max()
```
from(db: "mydb")
| > filter(fn: (r) =>
  r["host"] == "server0" and
  r["_measurement"] == "cpu" and
  r["_field"] == "usage_user"
  r["_value"] > 22.0)
| > range(start: -1m)
| > max()

this does a full table scan
Variables & Closures

db = "mydb"
measurement = "cpu"

from(db:db)
  |> filter(fn: (r) => r._measurement == measurement and r.host == "server0")
  |> last()
Variables & Closures

db = "mydb"
measurement = "cpu"

from(db: db)
|> filter(fn: (r) => r._measurement == measurement and r.host == "server0")
|> last()
User Defined Functions

db = "mydb"
measurement = "cpu"
fn = (r) => r._measurement == measurement and r.host == "server0"

from(db:db)
|> filter(fn: fn)
|> last()

assign function to variable fn
from(db: "mydb")
  |> filter(fn: (r) =>
      r["_measurement"] == "cpu" and
      r["_field"] == "usage_user" and
      r["host"] == "server0")
  |> range(start:-1h)
User Defined Functions

gerid of some common boilerplate?
User Defined Functions

```javascript
select = (db, m, f) => {
  return from(db)
    |> filter(fn: (r) => r._measurement == m and r._field == f)
}
```
User Defined Functions

```javascript
select = (db, m, f) => {
    return from(db:db)
        |> filter(fn: (r) => r._measurement == m and r._field == f)
}

select(db: "mydb", m: "cpu", f: "usage_user")
    |> filter(fn: (r) => r["host"] == "server0")
    |> range(start:-1h)
```
User Defined Functions

```plaintext
select = (db, m, f) => {
  return from(db:db)
    |> filter(fn: (r) => r._measurement == m and r._field == f)
}

select(m: "cpu", f: "usage_user")
  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1h)
```

Error calling function "select": missing required keyword argument "db"
Default Arguments

```javascript
select = (db="mydb", m, f) => {
  return from(db:db)
    |> filter(fn: (r) => r._measurement == m and r._field == f)
}

select(m: "cpu", f: "usage_user")
  |> filter(fn: (r) => r["host"] == "server0")
  |> range(start:-1h)
```
Default Arguments

```javascript
select = (db="mydb", m, f) => {
    return from(db:db)
        |> filter(fn: (r) => r._measurement == m and r._field == f)
}

select(m: "cpu", f: "usage_user")
    |> filter(fn: (r) => r["host"] == "server0")
    |> range(start:-1h)
```
Multiple Results to Client

data = from(db: "mydb")
    |> filter(fn: (r) r._measurement == "cpu" and r._field == "usage_user")
    |> range(start: -4h)
    |> window(every: 5m)

    data  |> min()  |> yield(name: "min")
    data  |> max()  |> yield(name: "max")
    data  |> mean()  |> yield(name: "mean")
Multiple Results to Client

data = from(db: "mydb")
| filter(fn: (r) r._measurement == "cpu" and r._field == "usage_user")
| range(start: -4h)
| window(every: 5m)

data | > min() | > yield(name: "min")
data | > max() | > yield(name: "max")
data | > mean() | > yield(name: "mean")

Result: min
ame


<table>
<thead>
<tr>
<th>time</th>
<th>field</th>
<th>measurement</th>
<th>host</th>
<th>region</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
User Defined Pipe Forwardable Functions

mf = (m, f, table=<-) => {
  return table
      |> filter(fn: (r) => r._measurement == m and
          r._field == f)
}

from(db: "mydb")
  |> mf(m: "cpu", f: "usage_user")
  |> filter(fn: (r) => r.host == "server0")
  |> last()
User Defined Pipe Forwardable Functions

```javascript
mf = (m, f, table=<-)) => {
    return table
        |> filter(fn: (r) => r._measurement == m and
                         r._field == f)
}
```

takes a table from a pipe forward by default

```javascript
from(db: "mydb")
    |> mf(m: "cpu", f: "usage_user")
    |> filter(fn: (r) => r.host == "server0")
    |> last()
```
User Defined Pipe Forwardable Functions

mf = (m, f, table=<=-) => {
  return table
      |> filter(fn: (r) => r._measurement == m and r._field == f)
}

from(db: "mydb")
  |> mf(m: "cpu", f: "usage_user")
  |> filter(fn: (r) => r.host == "server0")
  |> last()

calling it, then chaining
Passing as Argument

```javascript
mf = (m, f, table=<-) => {
    return table
        |> filter(fn: (r) => r._measurement == m and r._field == f)
}

sending the from as argument

mf(m: "cpu", f: "usage_user", table: from(db:"mydb"))
    |> filter(fn: (r) => r.host == "server0")
    |> last()
```
Passing as Argument

```haskell
mf = (m, f, table=<>-) =>
    filter(fn: (r) => r._measurement == m and r._field == f,
            table: table)
```

**rewrite the function to use argument**

```haskell
mf(m: "cpu", f: "usage_user", table: from(db:"mydb"))
|> filter(fn: (r) => r.host == "server0")
|> last()
```
Any pipe forward function can use arguments

min(table:
    range(start: -1h, table:
        filter(fn: (r) => r.host == "server0", table:
            from(db: "mydb"))))
Make you a Lisp
Easy to add Functions

like plugins in Telegraf
package functions

import {
  "fmt"
  "github.com/influxdata/ifql/ifql"
  "github.com/influxdata/ifql/query"nquery/execute"
  "github.com/influxdata/ifql/query/plan"
}

const CountKind = "count"

type CountOpSpec struct {

}

func init() {
  ifql.RegisterFunction(CountKind, createCountOpSpec)
  query.RegisterOpSpec(CountKind, newCountOp)
  plan.RegisterProcedureSpec(CountKind, newCountProcedure, CountKind)
  execute.RegisterTransformation(CountKind, createCountTransformation)
}

  if len(args) != 0 {
    return nil, fmt.Errorf(`count function requires no arguments`)
  }

  return new(CountOpSpec), nil
}

func newCountOp() query.OperationSpec {
  return new(CountOpSpec)
}

func (s *CountOpSpec) Kind() query.OperationKind {
  return CountKind
}
type CountProcedureSpec struct {
}

func newCountProcedure(query.OperationSpec) (plan.ProcedureSpec, error) {
    return new(CountProcedureSpec), nil
}

func (s *CountProcedureSpec) Kind() plan.ProcedureKind {
    return CountKind
}

func (s *CountProcedureSpec) Copy() plan.ProcedureSpec {
    return new(CountProcedureSpec)
}

func (s *CountProcedureSpec) PushDownRule() plan.PushDownRule {
    return plan.PushDownRule{
        Root: SelectKind,
        Through: nil,
    }
}

func (s *CountProcedureSpec) PushDown(root *plan.Procedure, dup func() *plan.Procedure) {
    selectSpec := root.Spec.(*SelectProcedureSpec)
    if selectSpec.AggregateSet {
        root = dup()
        selectSpec = root.Spec.(*SelectProcedureSpec)
        selectSpec.AggregateSet = false
        selectSpec.AggregateType = ""
        return
    }
    selectSpec.AggregateSet = true
    selectSpec.AggregateType = CountKind
}
type CountAgg struct {
    count int64
}

    t, d := execute.NewAggregateTransformationAndDataset(id, mode, ctx.Bounds(), new(CountAgg))
    return t, d, nil
}

func (a *CountAgg) DoBool(vs []bool) {
    a.count += int64(len(vs))
}

func (a *CountAgg) DoUInt(vs []uint64) {
    a.count += int64(len(vs))
}

func (a *CountAgg) DoInt(vs []int64) {
    a.count += int64(len(vs))
}

func (a *CountAgg) DoFloat(vs []float64) {
    a.count += int64(len(vs))
}

func (a *CountAgg) DoString(vs []string) {
    a.count += int64(len(vs))
}

func (a *CountAgg) Type() execute.DataType {
    return execute.TInt
}

func (a *CountAgg) ValueInt() int64 {
    return a.count
}
Defines parser, validation, execution
Imports and Namespaces

from(db: "mydb")
  |> filter(fn: (r) => r.host == "server0")
  |> range(start: -1h)
  // square the value
  |> map(fn: (r) => r._value * r._value)

shortcut for this?
Imports and Namespaces

from(db: "mydb")
  |> filter(fn: (r) => r.host == "server0")
  |> range(start: -1h)
  // square the value
  |> map(fn: (r) => r._value * r._value)

square = (table=<<) {
  table  |> map(fn: (r) => r._value * r._value)
}

Imports and Namespaces

```plaintext
import "github.com/pauldix/ifqlmath"

from(db:"mydb")
  |> filter(fn: (r) => r.host == "server0")
  |> range(start: -1h)
  |> ifqlmath.square()
```
Imports and Namespaces

```go
import "github.com/pauldix/ifqlmath"

from(db:"mydb"
    |> filter(fn: (r) => r.host == "server0")
    |> range(start: -1h)
    |> ifqlmath.square())
```
MOAR EXAMPLES!
Math across measurements

```plaintext
foo = from(db: "mydb")
    |> filter(fn: (r) => r._measurement == "foo")
    |> range(start: -1h)
bar = from(db: "mydb")
    |> filter(fn: (r) => r._measurement == "bar")
    |> range(start: -1h)
join(
    tables: {foo:foo, bar:bar},
    fn: (t) => t.foo._value + t.bar._value)
    |> yield(name: "foobar")
```
Having Query

```plaintext
from(db: "mydb")
| > filter(fn: (r) => r._measurement == "cpu")
| > range(start:-1h)
| > window(every:10m)
| > mean()
// this is the having part
| > filter(fn: (r) => r._value > 90)
```
Grouping

// group - average utilization across regions
from(db: "mydb")
  |> filter(fn: (r) => r._measurement == "cpu" and r._field == "usage_system")
  |> range(start: -1h)
  |> group(by: ["region"])  
  |> window(every: 10m)
  |> mean()
Get Metadata

```plaintext
from(db: "mydb")
| > filter(fn: (r) => r._measurement == "cpu")
| > range(start: -48h, stop: -47h)
| > tagValues(key: "host")
```
Get Metadata

from(db:"mydb")
| > filter(fn: (r) => r._measurement == "cpu")
| > range(start: -48h, stop: -47h)
| > group(by: ["measurement"], keep: ["host"])
| > distinct(column: "host")
tagValues = (table=<-) =>
  table
  |> group(by: ["measurement"], keep: ["host"])
  |> distinct(column: "host")
Get Metadata

from(db: "mydb")
| > filter(fn: (r) => r._measurement == "cpu")
| > range(start: -48h, stop: -47h)
| > tagValues(key: "host")
| > count()
Functions Implemented as IFQL

// _sortLimit is a helper function, which sorts and limits a table.
_sortLimit = (n, desc, cols=["_value"], table=<-) =>
    table
    |> sort(cols:cols, desc:desc)
    |> limit(n:n)

// top sorts a table by cols and keeps only the top n records.
top = (n, cols=["_value"], table=<-) =>
    _sortLimit(table:table, n:n, cols:cols, desc:true)
Project Status and Timeline
API 2.0 Work

Lock down query request/response format
Apache Arrow
We’re contributing the Go implementation!

https://github.com/influxdata/arrow
Finalize Language
(a few months or so)
Ship with Enterprise 1.6

(summertime)
Thank you!

Paul Dix
paul@influxdata.com
@pauldix