

IFQL

Paul Dix

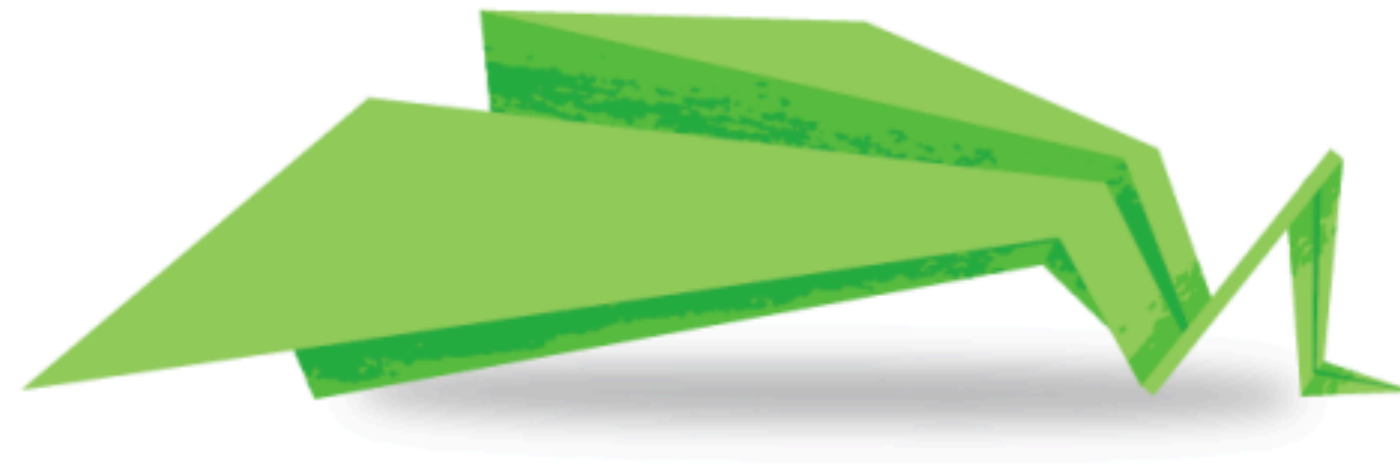
Founder & CTO

@pauldix

paul@influxdata.com

Evolution of a query language...

REST API



errplane

SQL-ish



Vaguely Familiar

```
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

 Open

TechniclaborErdmann opened this issue on Aug 28, 2015 · 80 comments

[feature request] Support month and year as duration unit

#3991

 **Open** ghost opened this issue on Sep 4, 2015 · 47 comments

Feature Request: DatePart in InfluxQL #6723

 **Open**

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

 **Open**

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

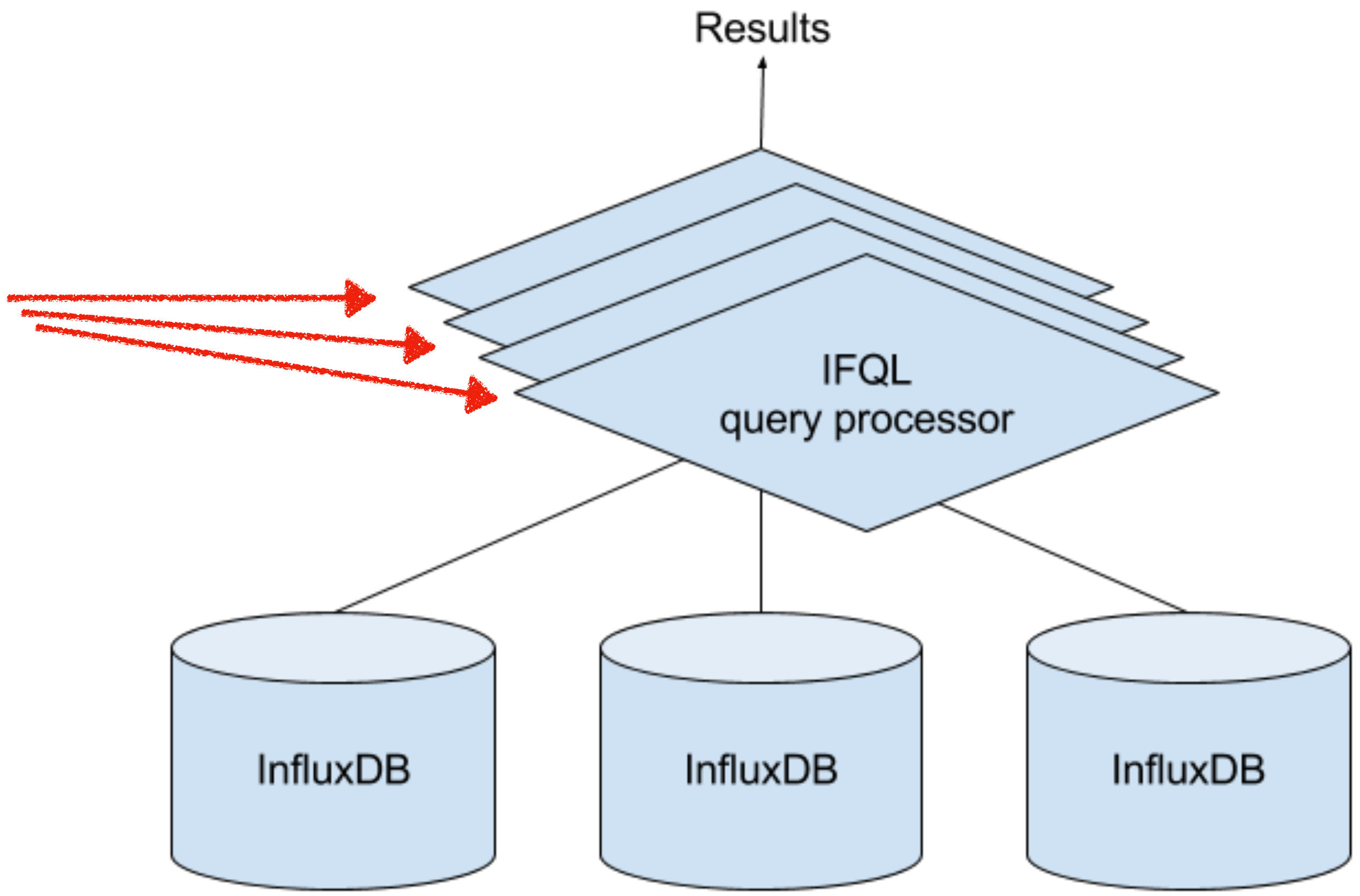


One Language to Unite!

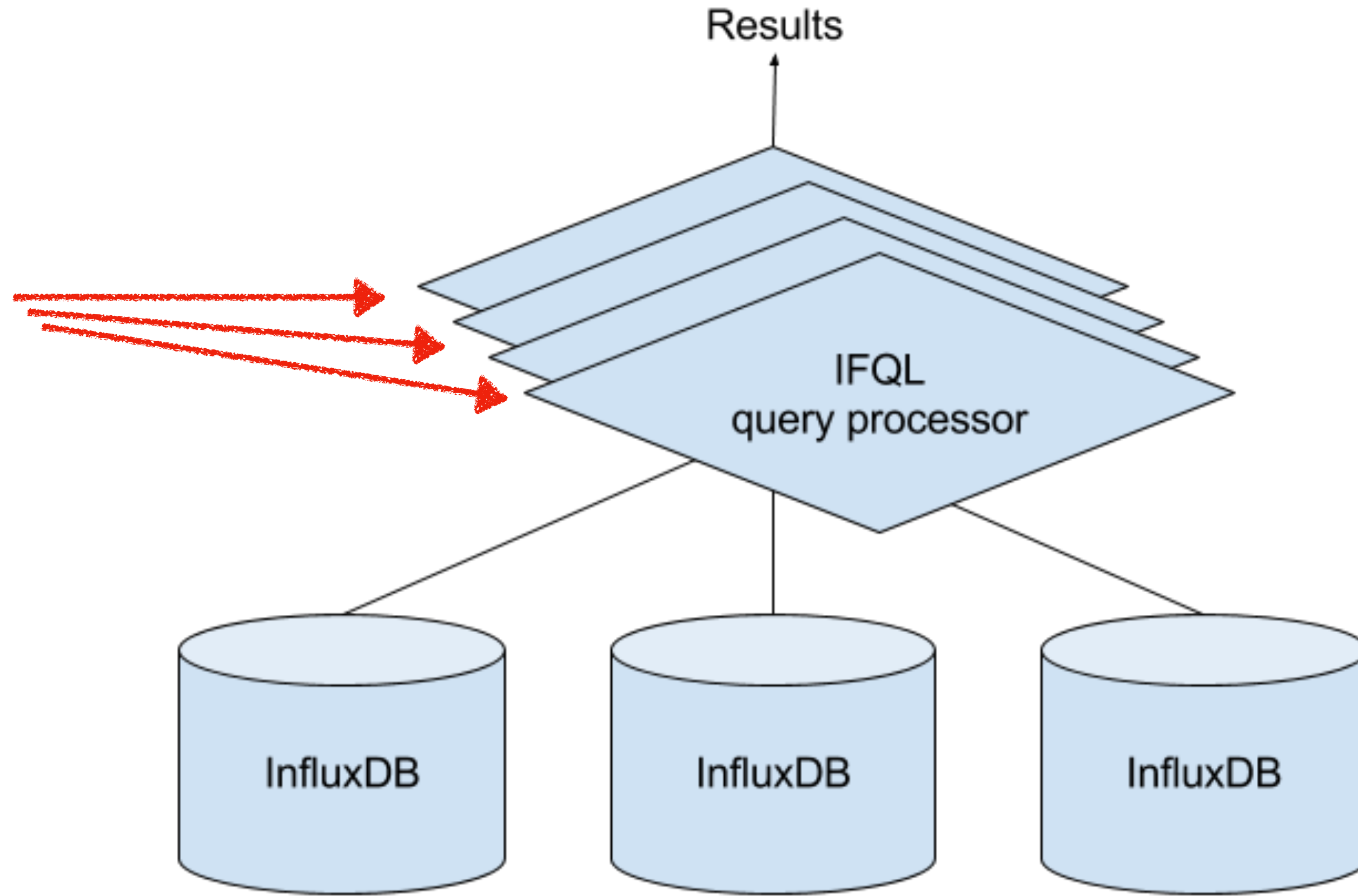
Feature Velocity

**Decouple storage from
compute**

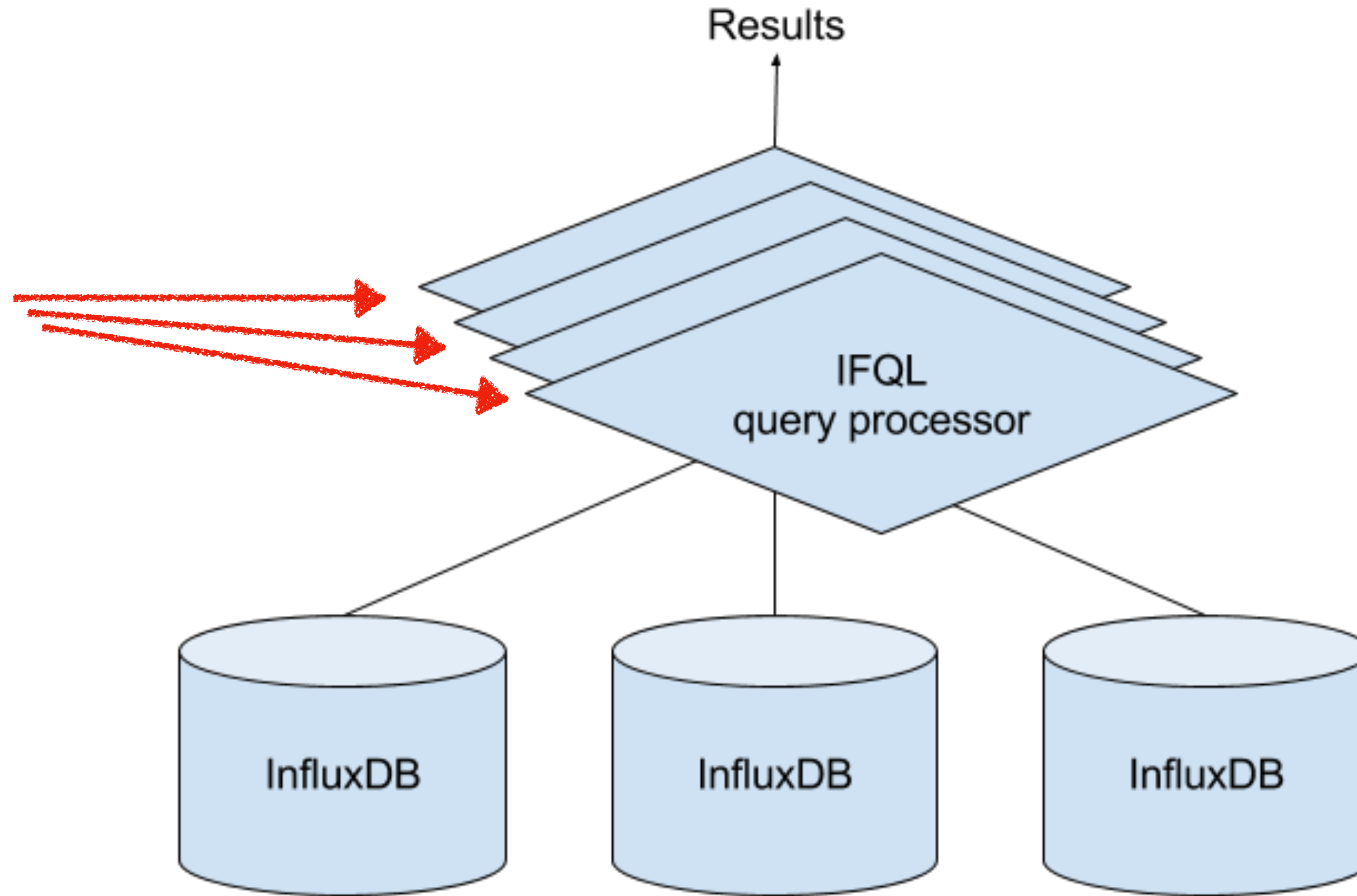
**Iterate & deploy
more frequently**



**Scale
independently**



**Workload
Isolation**





[Contact Sales](#)

[Products](#) ▾

[Solutions](#)

[Pricing](#)

[Getting Started](#)

[Documentation](#)

[Software](#)

[Support](#)

[More](#) ▾

[English](#) ▾

[My Account](#) ▾

[Sign In to the Console](#)



Amazon Athena

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

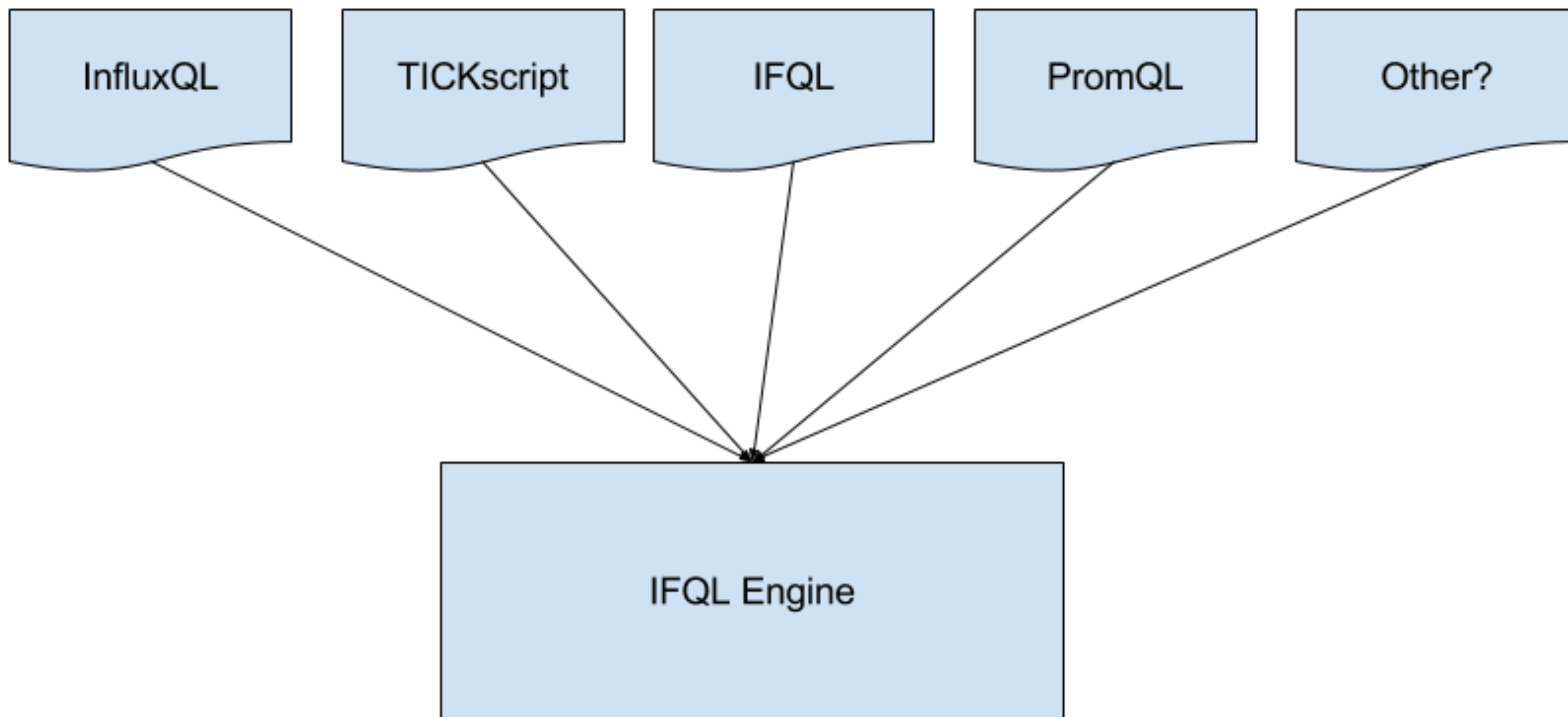
[Get Started with Amazon Athena](#)

**Decouple language from
engine**

```
{
  "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"
              }
            },
            "right": {
              "type": "stringLiteral",
              "value": "cpu"
            }
          }
        }
      }
    }
  ]
}
```

Query represented as DAG in JSON





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`

Block: keys: `[_field, _measurement, host, region]` bounds: `[1677-09-21T00:12:43.145224192Z, 2018-02-12T15:53:04.361902250Z)`

<code>_time</code>	<code>_field</code>	<code>_measurement</code>	<code>host</code>	<code>region</code>	<code>_value</code>
--------------------	---------------------	---------------------------	-------------------	---------------------	---------------------

<code>2018-02-12T15:53:00.000000000Z</code>	<code>usage_system</code>	<code>cpu</code>	<code>server0</code>	<code>east</code>	<code>60.6284</code>
---	---------------------------	------------------	----------------------	-------------------	----------------------

Block: keys: `[_field, _measurement, host, region]` bounds: `[1677-09-21T00:12:43.145224192Z, 2018-02-12T15:53:04.361902250Z)`

<code>_time</code>	<code>_field</code>	<code>_measurement</code>	<code>host</code>	<code>region</code>	<code>_value</code>
--------------------	---------------------	---------------------------	-------------------	---------------------	---------------------

<code>2018-02-12T15:53:00.000000000Z</code>	<code>usage_user</code>	<code>cpu</code>	<code>server0</code>	<code>east</code>	<code>39.3716</code>
---	-------------------------	------------------	----------------------	-------------------	----------------------

```
// 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)
```

```

// 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)

```

Result: _result

Block: keys: [_field, _measurement, host, region] bounds: [2018-02-12T16:01:45.677502014Z, 2018-02-12T16:02:45.677502014Z)

_time	_field	_measurement	host	region	_value
2018-02-12T16:01:50.000000000Z	usage_user	cpu	server0	east	50.549
2018-02-12T16:02:00.000000000Z	usage_user	cpu	server0	east	35.4458
2018-02-12T16:02:10.000000000Z	usage_user	cpu	server0	east	30.0493
2018-02-12T16:02:20.000000000Z	usage_user	cpu	server0	east	44.3378
2018-02-12T16:02:30.000000000Z	usage_user	cpu	server0	east	11.1584
2018-02-12T16:02:40.000000000Z	usage_user	cpu	server0	east	46.712

```
// 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
2018-02-12T15:13:41.128654848Z	usage_user	cpu	server0	east	45.5091088235294
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
2018-02-12T16:05:06.708945484Z	usage_system	cpu	server0	east	53.58707631578949
2018-02-12T15:13:41.128654848Z	usage_system	cpu	server0	east	54.49089117647058
2018-02-12T15:43:41.128654848Z	usage_system	cpu	server0	east	48.80836666666666

Elements of IFQL

Functional

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

Functional

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

built in functions



Functional

```
// 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 functions



Functional

```
// 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



Named Parameters

```
// 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

Inputs

```
// 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



Outputs

```
// 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



Outputs

```
// 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 }



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)
```

↑
tag key

```
{ "_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)
```



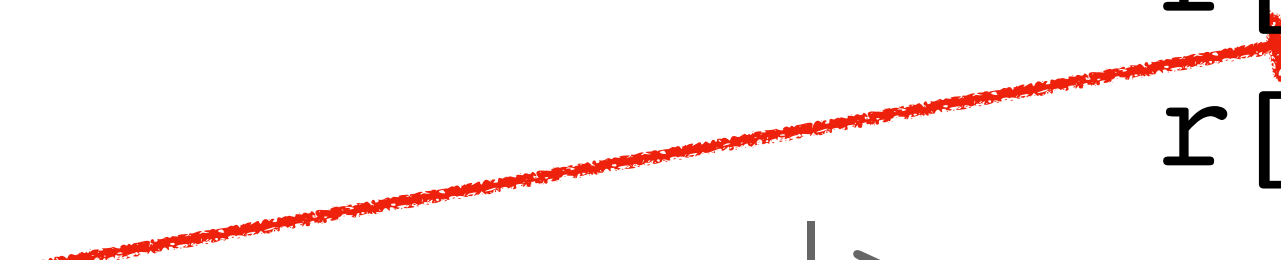
same as before

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

Special Properties

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

**starts with _
reserved for system
attributes**



```
{ "_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( )
```

works other way



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


Special Properties

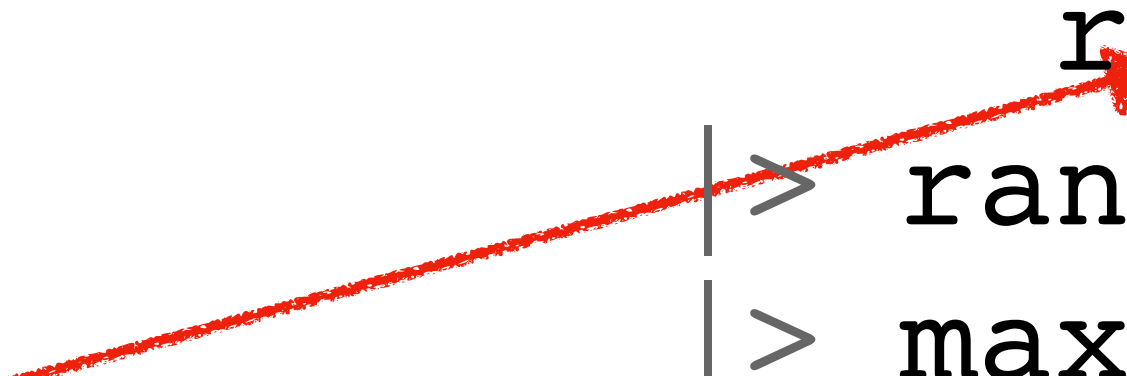
```
from(db: "mydb" )
  |> filter(fn: (r) =>
    r["host"] == "server0" and
    r["_measurement"] == "cpu" and
    r["_field"] == "usage_user" )
  |> range(start:-1m)
  |> max( )
```

**_measurement and _field
present for all InfluxDB data**

```
{ "_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" 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 }
```

Filter function output

```
// 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**



Filter Operators

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

↑
!=
=~
!~
in

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



Function with explicit return

```
// 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



Outputs

```
// 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

Outputs

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



pipled to range

which further filters by a time range

Outputs

```
// 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()
```

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

range and filter switched

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()
```

results the same

Result: _result

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

_time	_field	_measurement	host	region	_value
2018-02-12T17:53:02.322301856Z	usage_user	cpu	server0	east	97.3174

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()
```

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")
|> 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( )
```

**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()
```

```
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)
```

**here it operates on
only the last minute of data**



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()
```

**here it operates on
data for all time**



```
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()
```

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

```
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

Optimization

```
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

```
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( )
```

this is more efficient



Optimization

```
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()
```



**query DAG different
plan DAG same as one on left**

Optimization

```
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( )
```

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

**anonymous filter function
closure over surrounding context**



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



User Defined Functions

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

User Defined Functions

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

get rid of some common boilerplate?

User Defined Functions

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

User Defined Functions

```
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

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

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

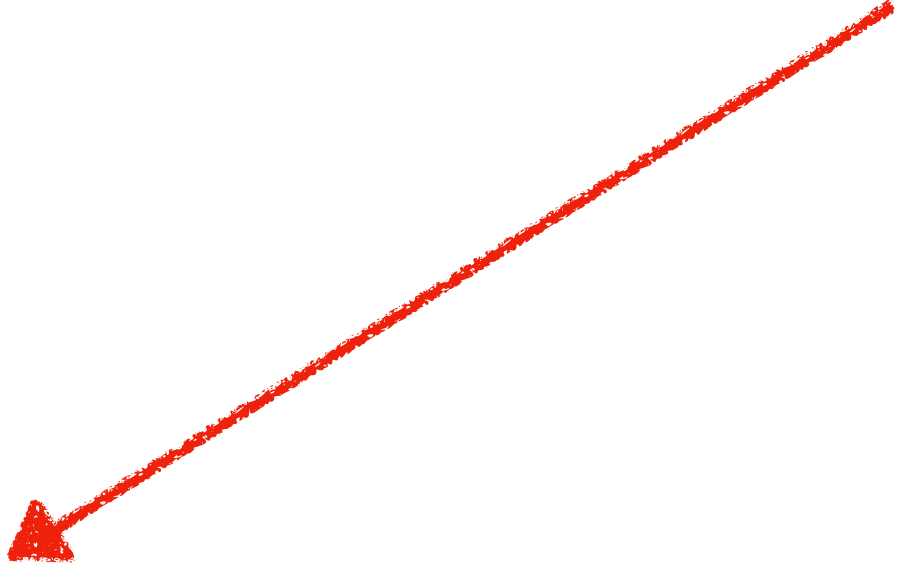
error calling function "select": missing required keyword argument "db"

Default Arguments

```
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



```
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 ← **name**

Block: keys: [_field, _measurement, host, region] bounds: [2018-02-12T16:55:55.487457216Z, 2018-02-12T20:55:55.487457216Z)

_time	_field	_measurement	host	region	_value
-------	--------	--------------	------	--------	--------

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

**takes a table
from a pipe forward
by default**

```
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

```
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

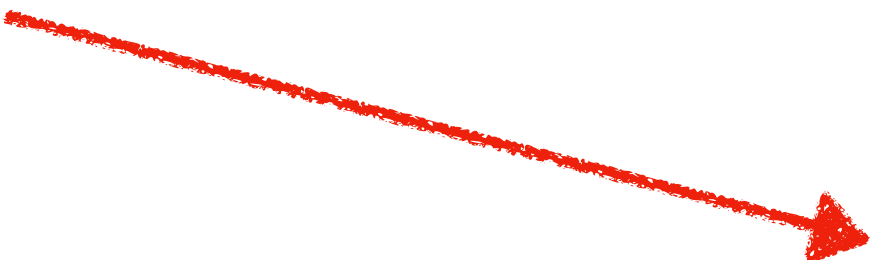


Passing as Argument

```
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

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



rewrite the function to use argument

```
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

code file



▼ functions

/* count.go

/* count_test.go

/* data_test.go

/* first.go

/* first_test.go

/* group.go

/* group_test.go

/* join.go

/* join_test.go

/* last.go

/* last_test.go

/* limit.go

/* limit_test.go

/* max.go

/* max_test.go

test file



▼ functions

/* count.go

/* count_test.go

/* data_test.go

/* first.go

/* first_test.go

/* group.go

/* group_test.go

/* join.go

/* join_test.go

/* last.go

/* last_test.go

/* limit.go

/* limit_test.go

/* max.go

/* max_test.go

```
package functions
```

```
import (  
    "fmt"  
  
    "github.com/influxdata/ifql/ifql"  
    "github.com/influxdata/ifql/query"  
    "github.com/influxdata/ifql/query/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)  
}
```

```
func createCountOpSpec(args map[string]ifql.Value, ctx ifql.Context) (query.OperationSpec, error) {  
    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
}

func createCountTransformation(id execute.DatasetID, mode execute.AccumulationMode, spec plan.ProcedureSpec, ctx execute.Context)
(execute.Transformation, execute.Dataset, error) {
    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

```
import "github.com/pauldix/ifqlmath"  
  
from(db: "mydb" )  
  |> filter(fn: (r) => r.host == "server0" )  
  |> range(start: -1h)  
  |> ifqlmath.square( )
```

Imports and Namespaces

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

```
from(db: "mydb" )
```

```
|> filter(fn: (r) => r.host == "server0" )
```

```
|> range(start: -1h)
```

```
|> ifqlmath.square( )
```

namespace



MOAR EXAMPLES!

Math across measurements

```
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

```
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

```
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" )
```

Get Metadata

```
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

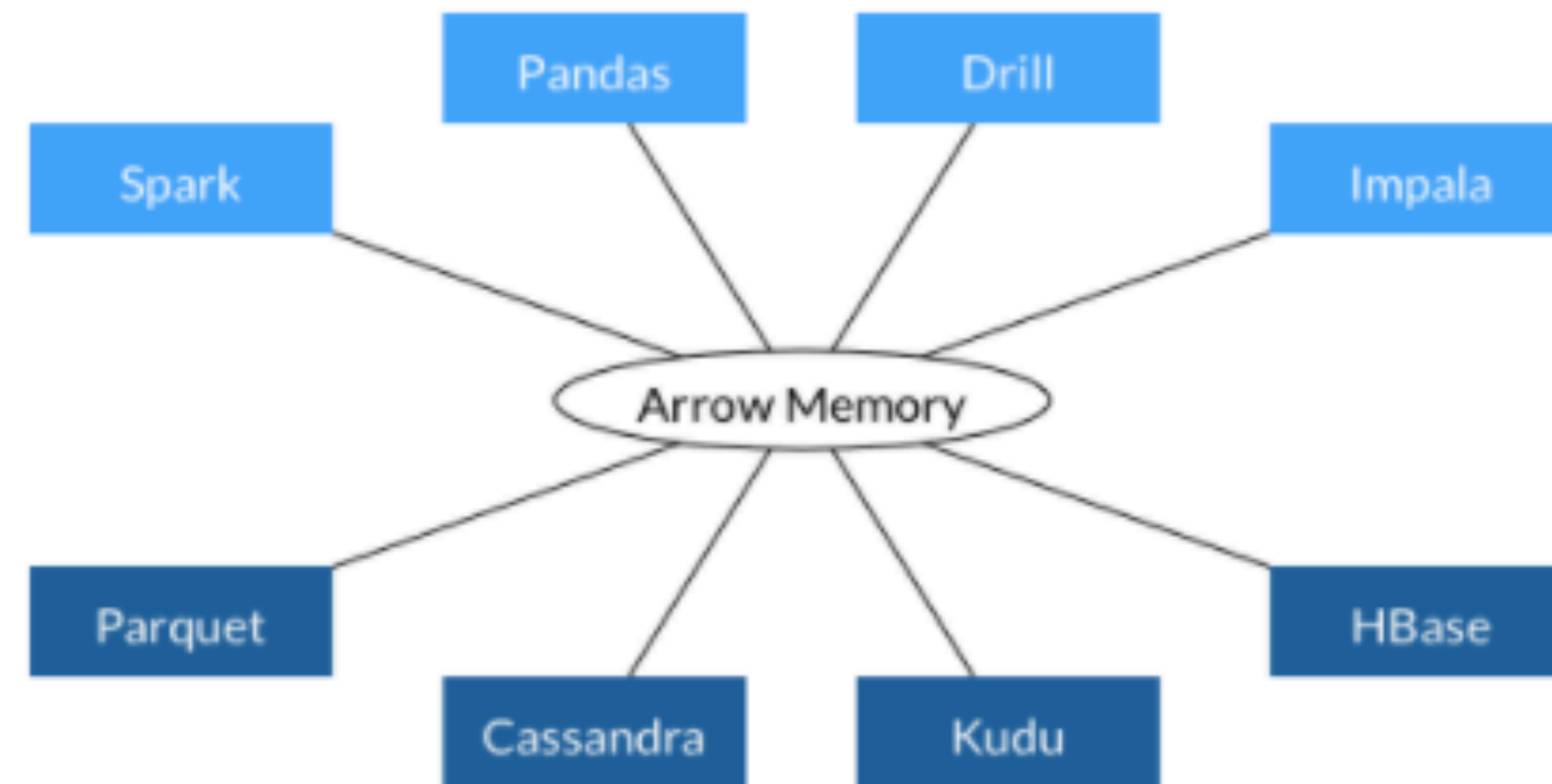
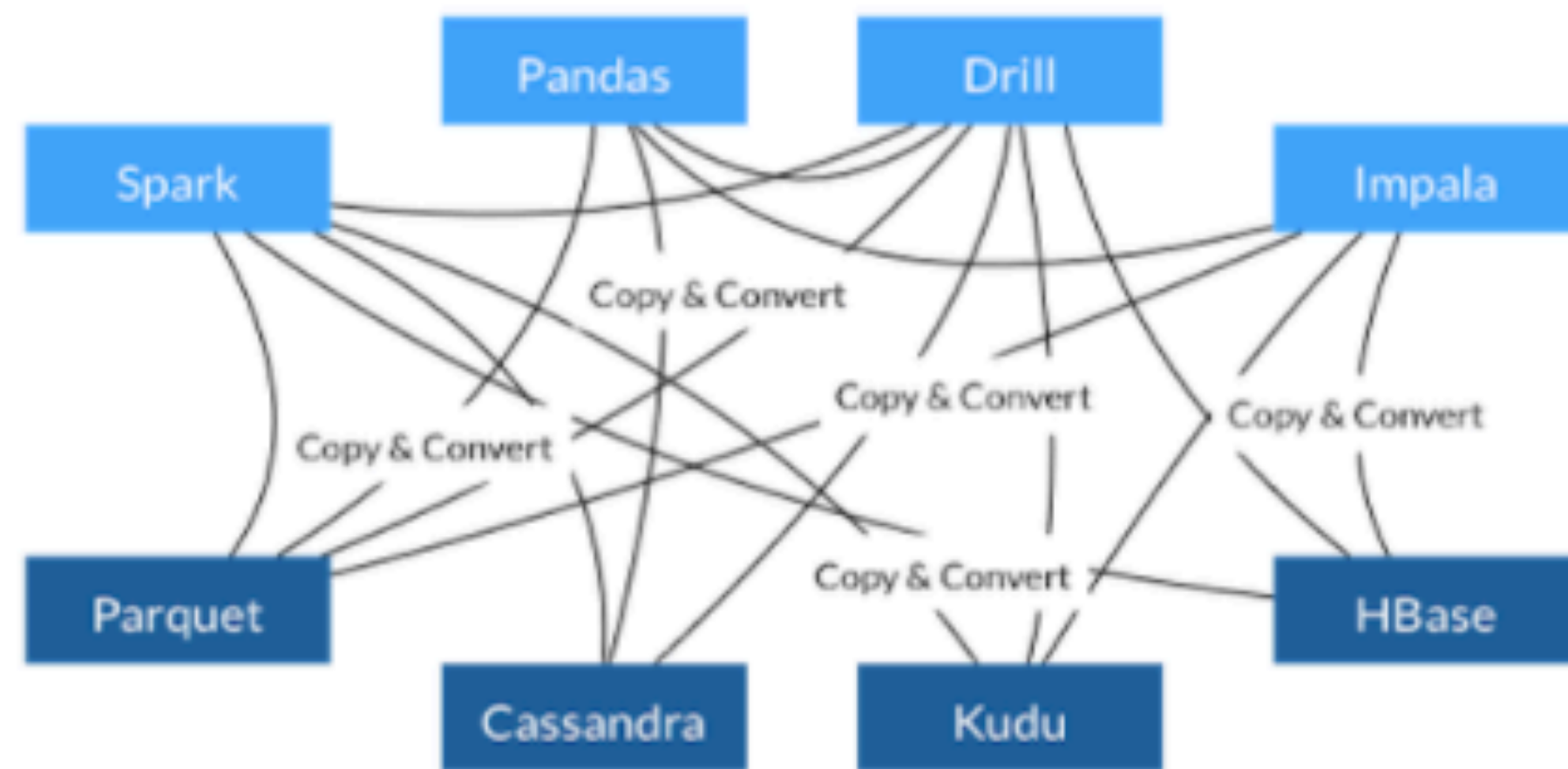
```
// _sortByLimit is a helper function, which sorts  
// and limits a table.  
_sortByLimit = (n, desc, cols=["_value"], table=<-) =>  
  table  
  |> sort(cols:cols, desc:desc)  
  |> limit(n:n)  
  
// top sorts a table by cols and keeps only the top n records.  
top = (n, cols=["_value"], table=<-) =>  
  _sortByLimit(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