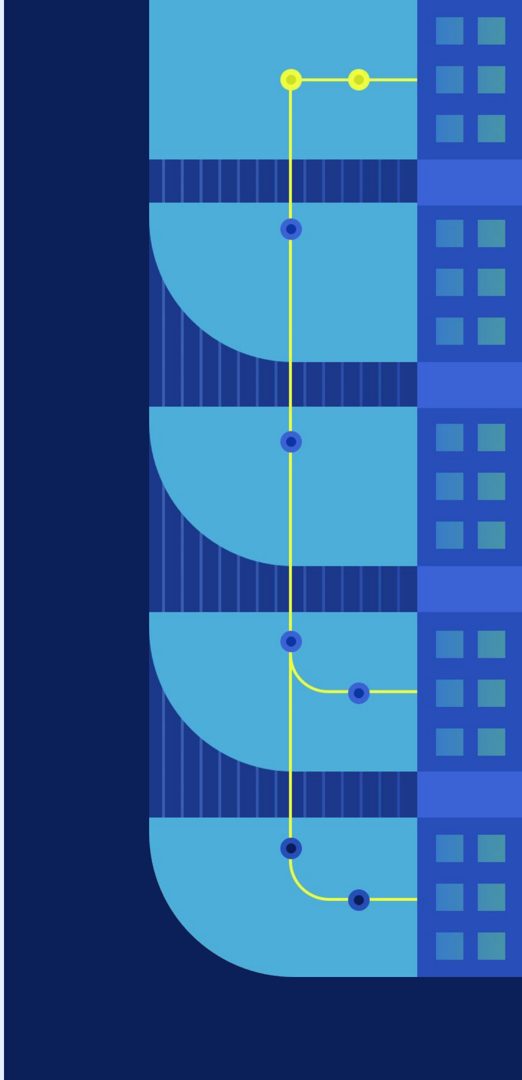
Introducing Switch: a framework for custom data applications

Josh Ferguson
Chief Architect @ Mode



we're going to talk about building tools to make better decisions with data

actually we’re going to talk about building tools to build tools to make better decisions with data

i’ve been obsessed with building these kinds of tools for about 20 years

@besquared almost everywhere

mode(.com)

a collaborative data science platform

our users are data scientists, analysts, and engineers

whose job it is to help everybody make better decisions with data

we make visualization tools

a few years ago we decided to rethink our approach to building these tools

from static to explorable

from presentation to analysis

we had to build a better data framework

we call that framework switch

it’s a collection of typescript libraries and tools that accelerate our data application development we’re in the process of releasing the framework with an open source license we think that everybody should build data applications with it

what’s a data application?

a customer data portal

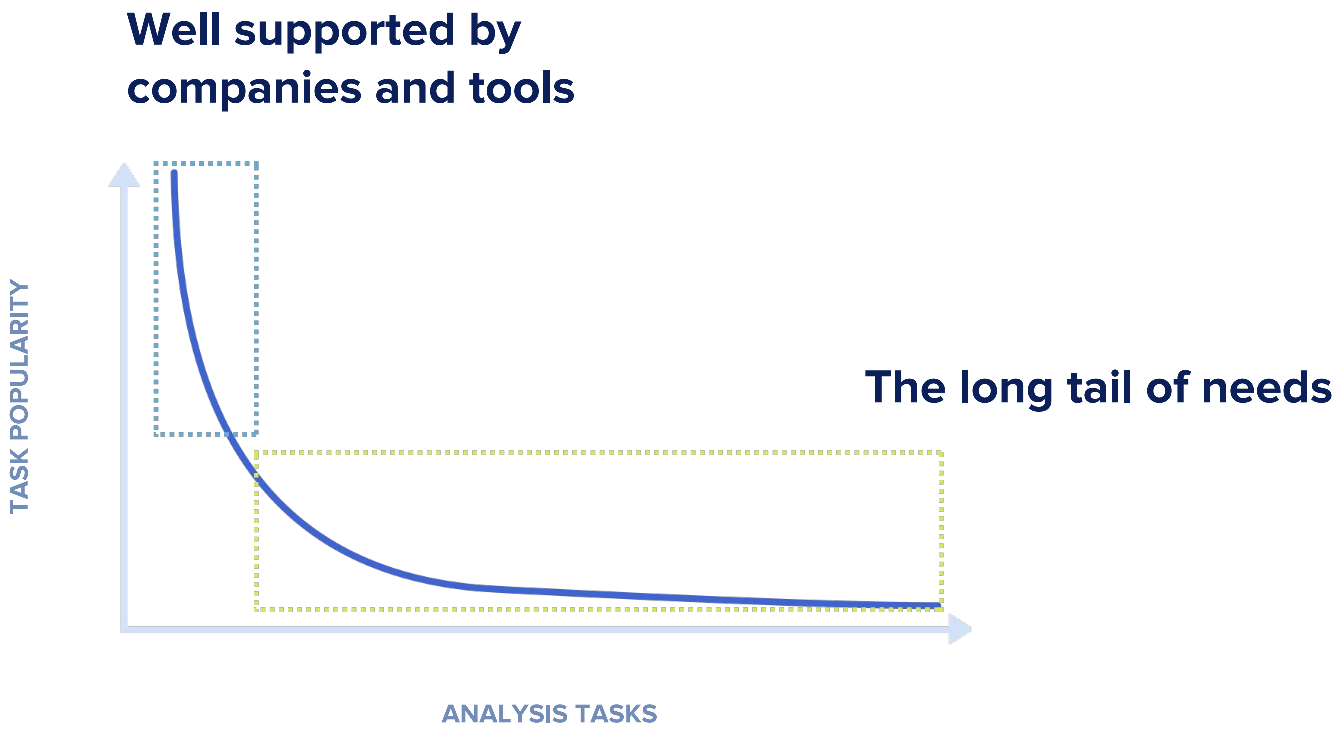
an a/b testing tool for our product team an account lookup app for sales

a customer engagement scoring tool for success

a revenue modeling dashboard for finance

why would we build these ourselves?

can’t we just buy them?


Well supported by companies and tools
The long tail of needs
TASK POPULARITY
ANALYSIS TASKS

there's no collection of off-the-shelf tools that will provide everything we need to make better decisions with data

as our organizations continue to become more analytically mature, we need to build custom data apps to keep up

sounds great, what’s the problem?

today almost all of these apps are one-offs

🤑 🕑 🚑

how can we make our data apps faster to build, easier to maintain, and more reliable?

there’s several challenges we’ve got to address to get there

we’re going to look at three of them

how the framework addresses each one

challenge number one

CHALLENGE #1

our data is incomplete and messy

CHALLENGE #1

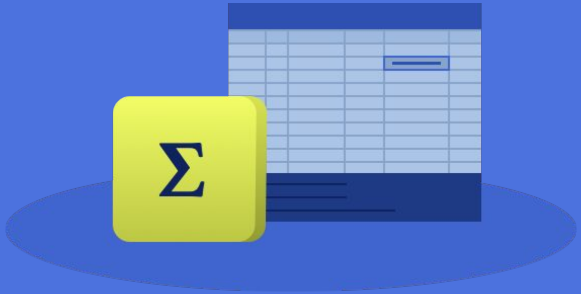
we don’t know what we’ll need ahead of time

CHALLENGE #1

we can't build a new etl pipeline or deploy our app every time we need to answer a slightly different question

CHALLENGE #1

we should let our users quickly and easily express data in new and different ways



Introducing Formulas

an excel-like language for data expression

they lets us work with datasets, even if we’re not database or programming language experts what can we do with them?

unlike excel whose formulas operate on cells, our formulas operate on entire datasets at a time

FORMULAS

sample dataset

ID	Date	Product	Quantity	Price	Filled
1	2019-01-01	A	10	10.00	true
2	2019-01-02	B	5	20.00	false
...	...	...	...	...	

EXAMPLES

calculate ratios!

```
[Price] / [Quantity]
```

EXAMPLES

convert units!

```
Dollar to cents

[Price] * 100
```

EXAMPLES

clean data!

```
CASE [Product]
WHEN “A,”
THEN “A”
ELSE [Product]
 END
```

EXAMPLES

aggregate data!

```
AVG([Price] / [Quantity])
```

EXAMPLES

lookup values!

```
LOOKUP(AVG([Price]), FIRST())
```

what else!?

let’s look at the language to find out

LITERALS

nulls

```
NULL
```

LITERALS

booleans

```
TRUE

FALSE
```

LITERALS

numbers

```
-42

1000

3.1415926

0xBEEF
```

LITERALS

strings

```
'Category'

"Product Name"
```

LITERALS

dates

```
#2019-04-18#

#2019-04-18T10:50:15#
```

LITERALS

regular expressions

```
/[\w\d]+/ig
```

ACCESS

data access

```
[Product]
[Quantity]
```

OPERATORS

mathematic

```
[Quantity] * 500
[Quantity] / 500
[Quantity] + 500
[Quantity] - 500
[Quantity] % 500
```

OPERATORS

relational

```
[Quantity]  = 500
[Quantity] <> 500
[Quantity]  < 500
[Quantity] <= 500
[Quantity]  > 500
[Quantity] >= 500
```

OPERATORS

logical

```
NOT [Filled]
[Filled] AND [Quantity]  > 500
[Filled]  OR [Quantity] <= 500
```

CONDITIONAL

if/elsif/else

```
   IF [Quantity] > 500
 THEN “Large Order”
ELSIF [Quantity] > 250
 THEN “Medium Order”
 ELSE “Small Order”
  END
```

CONDITIONAL

case

```
CASE [Filled]
WHEN TRUE
THEN “Filled”
WHEN FALSE
THEN “Unfilled”
ELSE “Unknown”
 END
```

FUNCTIONS

constant

```
NOW()
```

FUNCTIONS

scalar

```
FLOOR([Price])

TRIM([Product])

DATETRUNC('day', [Date])
```

FUNCTIONS

aggregate

```
SUM([Price])

AVG([Quantity])

COUNTD([Product])
```

FUNCTIONS

analytic

```
RANK(SUM([Quantity]))

RUNNING_SUM(COUNT([Price]))

LOOKUP(AVG([Price]), FIRST())
```

that’s it, simple and powerful

the possibilities are infinite

formulas play a key role in the framework and in our data applications

in the framework we use them to model, query, and process our data

they provide a common language that we can use across all of our data applications we can build interfaces that let users extend our apps with their own business logic and calculations a single formula that takes someone a few minutes to write might take hours or days to implement and deploy otherwise

not having to build ETL pipelines or write app code all the time can amplify our effort 100x

that’s pretty rad

let’s keep going and see how we use formulas to query our data

challenge number two

CHALLENGE #2

getting from data to visualization

CHALLENGE #2

we can't build ad-hoc data transformation code every time we want to build a new visualization

CHALLENGE #2

we should describe the data we need in way that matches the visualizations we're trying to build

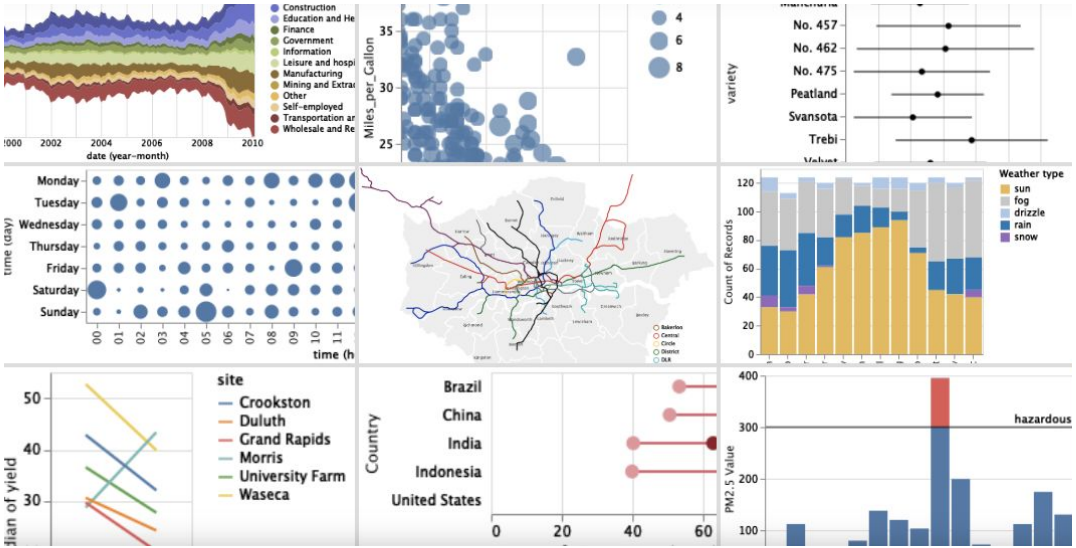
Introducing Queries

our queries speak the language of data visualization

what language is that?

QUERIES

none other than the grammar of graphics



most of the visualizations that we can encode with tools like vega-lite can be translated directly into switch queries

how do they work?

first we define the data we want to see

QUERIES

we start with fields which are defined with a formula

```
Field {
  formula: string;
}
```

QUERIES

they let us describe the data and calculations we want to see in our result

```
Field {
  formula: string;
}
```

QUERIES

there are two special fields worth mentioning which are called names and values

```
Names {
  formula: “$[Names]”;
}

Values {
  formula: “$[Values]”;
}
```

NAMES/VALUES

they let us combine multiple aggregate fields together into a single field

```
Names {
  formula: “$[Names]”;
}

Values {
  formula: “$[Values]”;
}
```

NAMES/VALUES

we’ll go through an example of where that can be useful in a minute

```
Names {
  formula: “$[Names]”;
}

Values {
  formula: “$[Values]”;
}
```

QUERIES

next we’ve got filters

```
Filter {
  field: Field;
  conds: Conditions;
}
```

QUERIES

they let us get rid of data we don't want in our result by adding conditions on our fields

```
Filter {
  field: Field;
  conds: Conditions;
}
```

QUERIES

last but not least we have sorts

```
Sort {
  field: Field;
   type: SortType;
  order: SortOrder;
}
```

QUERIES

they let us re-arrange our result by adding orders to our fields

```
Sort {
  field: Field;
   type: SortType;
  order: SortOrder;
}
```

then we map our data to our visualization

QUERIES

the first way to do that is with marks

```
Mark {
  field: Field;
  color: Field[];
   size: Field[];
  label: Field[];
     ...
}
```

QUERIES

marks are how we define the layers of our visualization

```
Mark {
  field: Field;
  color: Field[];
   size: Field[];
  label: Field[];
    ...
}
```

QUERIES

1 mark = 1 layer

```
Mark {
  field: Field;
  color: Field[];
   size: Field[];
  label: Field[];
    ...
}
```

MARKS

it’s defined by a single field

```
Mark {
  field: Field;
  color: Field[];
   size: Field[];
  label: Field[];
     ...
}
```

MARKS

it's got channels like color, size, and label, that let us associate data with visual properties

```
Mark {
  field: Field;
  color: Field[];
   size: Field[];
  label: Field[];
    ...
}
```

MARKS

we can map as many channels as we want based on the needs of our visualization

```
Mark {
  field: Field;
  color: Field[];
   size: Field[];
  label: Field[];
     ...
}
```

QUERIES

using marks and the other pieces we talked about we can build a complete visual mapping which we call a pivot query

```
PivotQuery {
   column: Field[];
        x: Field[];
      row: Field[];
        y: Field[];

   values: Field[];

    marks: Mark[];
  filters: Filter[];
    sorts: Sort[];
}
```

QUERIES

we’ve got our marks, filters, and sorts

```
PivotQuery {
    column: Field[];
         x: Field[];
       row: Field[];
         y: Field[];

    values: Field[];

     marks: Mark[];
   filters: Filter[];
     sorts: Sort[];
}
```

PIVOT QUERY

we’ve also got some other channels here that we can use to map other visual properties

```
PivotQuery {
    column: Field[];
         x: Field[];
       row: Field[];
         y: Field[];

    values: Field[];

     marks: Mark[];
   filters: Filter[];
     sorts: Sort[];
}
```

PIVOT QUERY

first, column and row which let us facet or group data across or down our visualization respectively

```
PivotQuery {
    column: Field[];
         x: Field[];
       row: Field[];
         y: Field[];

    values: Field[];

     marks: Mark[];
   filters: Filter[];
     sorts: Sort[];
}
```

PIVOT QUERY

next, x and y which let us position data across or down our visualization within those facets

```
PivotQuery {
    column: Field[];
         x: Field[];
       row: Field[];
         y: Field[];

    values: Field[];

     marks: Mark[];
   filters: Filter[];
     sorts: Sort[];
}
```

PIVOT QUERY

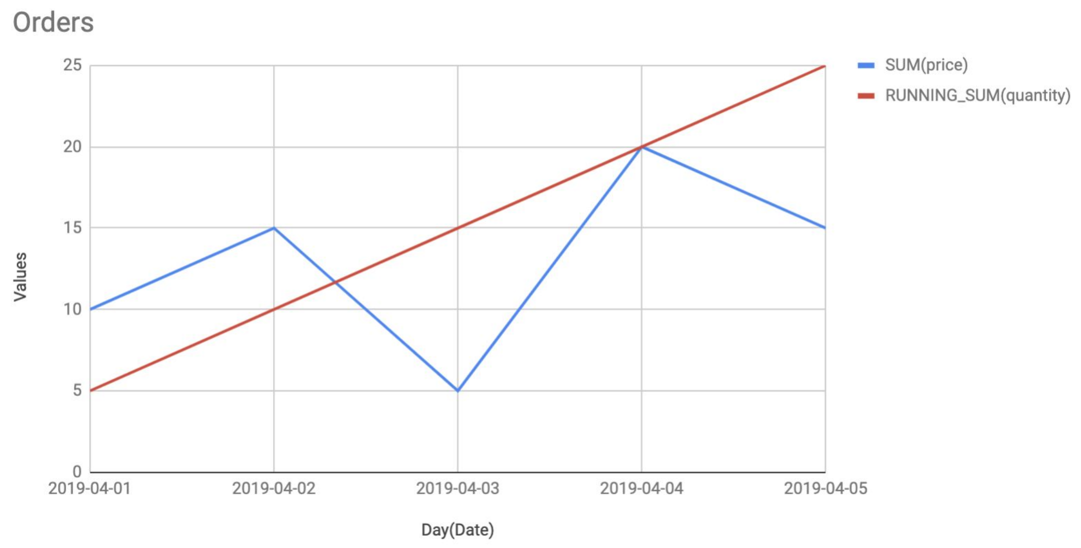
finally, we’ve got values which let’s us combine all of the fields in it into a single field that we can use in the other channels

```
PivotQuery {
   column: Field[];
        x: Field[];
      row: Field[];
        y: Field[];

   values: Field[];

    marks: Mark[];
  filters: Filter[];
    sorts: Sort[];
}
```

Let’s visualize an example

QUERIES

here’s a beautiful chart



QUERIES

a beautiful chart deserves a beautiful query

```
PivotQuery {
  x: [ “DATETRUNC(‘day’, [Date])” ],
  y: [ “$[Values]” ],
  values: [
    “SUM([Price])”,
    “RUNNING_SUM(SUM([Quantity]))”
  ],
  marks: [{
    field: “$[Values]”,
    color: [ “$[Names]” ]
  }]
}
```

EXAMPLE

we’ve got day on the x axis

```
PivotQuery {
  x: [ “DATETRUNC(‘day’, [Date])” ],
  y: [ “$[Values]” ],
  values: [
    “SUM([Price])”,
    “RUNNING_SUM(SUM([Quantity]))”
  ],
  marks: [{
    field: “$[Values]”,
    color: [ “$[Names]” ]
  }]
}
```

EXAMPLE

our values field on the y axis

```
PivotQuery {
  x: [ "DATETRUNC('day', [Date])" ],
  y: [ "$[Values]" ],
  values: [
    "SUM([Price])",
    "RUNNING_SUM(SUM([Quantity]))"
  ],
  marks: [{
    field: "$[Values]",
    color: [ "$[Names]" ]
  }]
}
```

EXAMPLE

there's the sum of price and a running sum of quantity in values

```
PivotQuery {
  x: [ "DATETRUNC('day', [Date])" ],
  y: [ "$[Values]" ],
  values: [
    "SUM([Price])",
    "RUNNING_SUM(SUM([Quantity]))"
  ],
  marks: [{
    field: "$[Values]",
    color: [ "$[Names]" ]
  }]
}
```

EXAMPLE

this chart is going to be a single layer so we’ve got one mark

```
PivotQuery {
  x: [ “DATETRUNC(‘day’, [Date])” ],
  y: [ “$[Values]” ],
  values: [
    “SUM([Price])”,
    “RUNNING_SUM(SUM([Quantity]))”
  ],
  marks: [{
    field: “$[Values]”,
    color: [ “$[Names]” ]
  }]
}
```

EXAMPLE

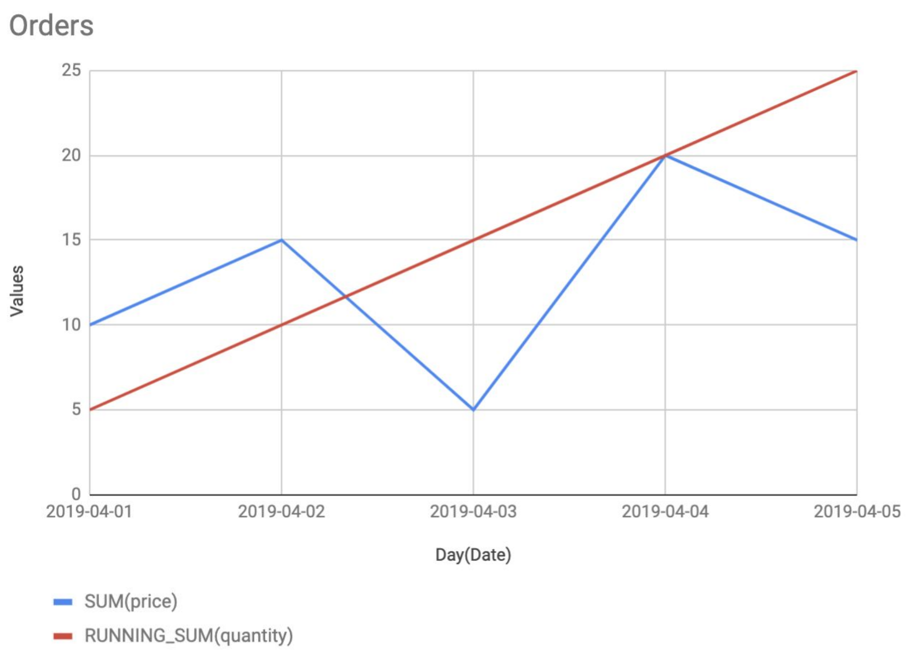
it’s defined by our values field

```
PivotQuery {
  x: [ “DATETRUNC(‘day’, [Date])” ],
  y: [ “$[Values]” ],
  values: [
    “SUM([Price])”,
    “RUNNING_SUM(SUM([Quantity]))”
  ],
  marks: [{
    field: “$[Values]”,
    color: [ “$[Names]” ]
  }]
}
```

EXAMPLE

within that layer we want to see two distinct series each with its own color so we add names to our color channel

```
PivotQuery {
  x: [ “DATETRUNC(‘day’, [Date])” ],
  y: [ “$[Values]” ],
  values: [
    “SUM([Price])”,
    “RUNNING_SUM(SUM([Quantity]))”
  ],
  marks: [{
    field: “$[Values]”,
    color: [ “$[Names]” ]
  }]
}
```




```
PivotQuery {
  x: [ “DATETRUNC(‘day’, [Date])” ],
  y: [ “$[Values]” ],
  values: [
    “SUM([Price])”,
    “RUNNING_SUM(SUM([Quantity]))”
  ],
  marks: [{
    field: “$[Values]”,
    color: [ “$[Names]” ]
  }]
}
```

over time it becomes second nature

once we learn to speak the language our ability to quickly transform and visualize data is increased by 10x challenge number three

CHALLENGE #3

our datasets are millions and billions of rows and growing

CHALLENGE #3

we can't constantly move it around or materialize every view we might need to analyze ahead of time

CHALLENGE #3

we should work with data it as it exists in the places where it already lives

Introducing Processors

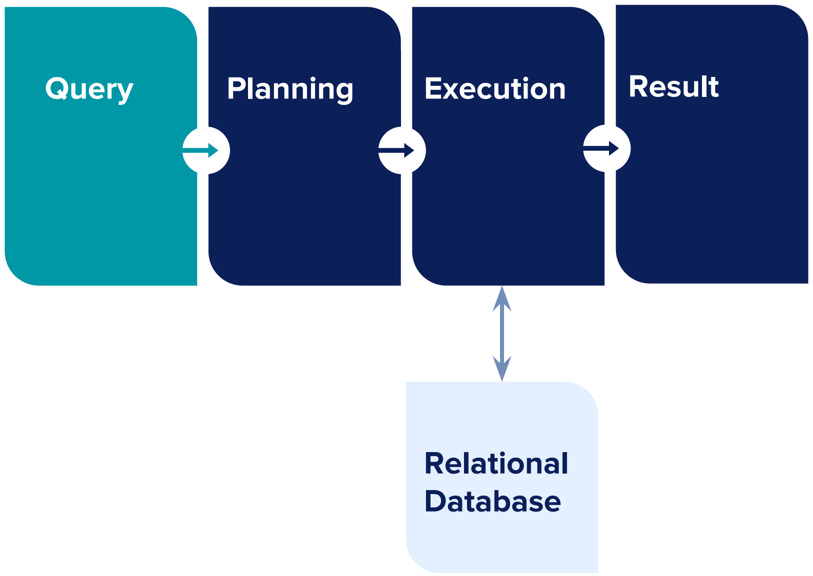
they’re our database’s analytical co-pilots

what do we mean by that?

let’s talk about how they work

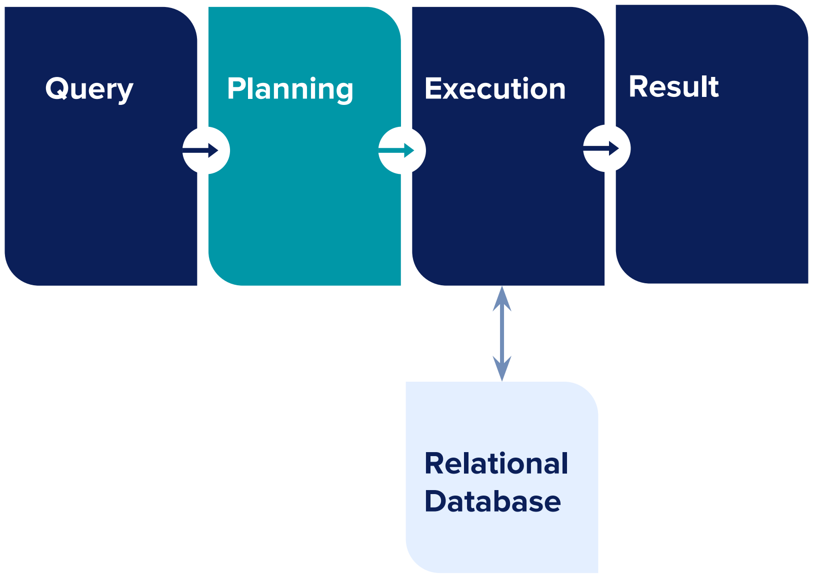
PROCESSORS

we start with a query like the one we saw in the last section



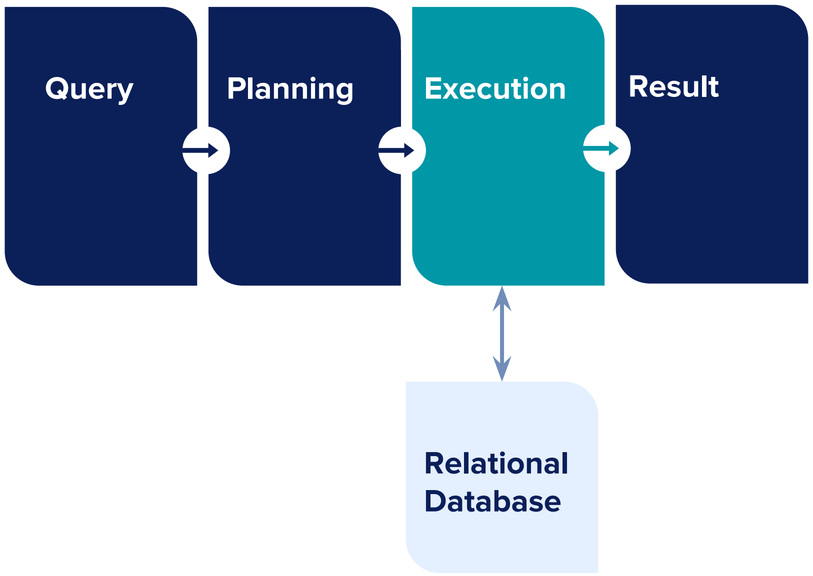
PROCESSORS

we build a plan, which is a set of instructions for processing that query



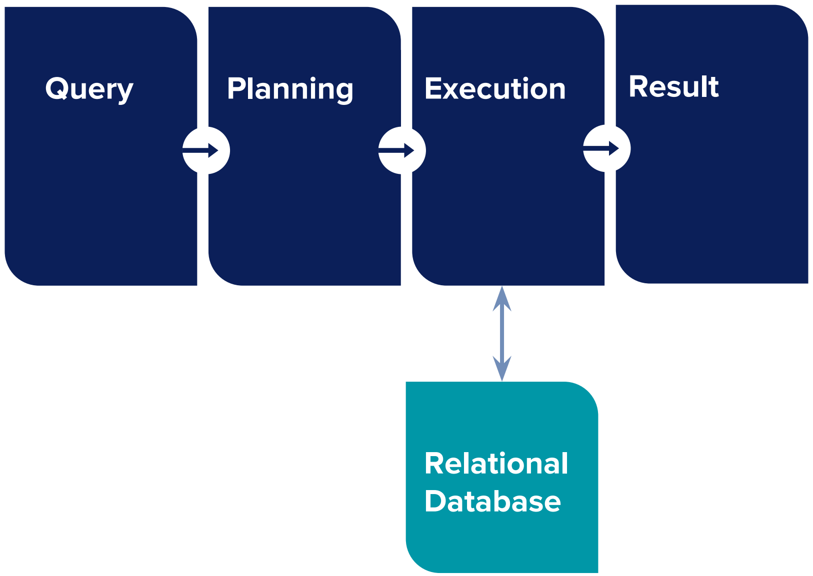
PROCESSORS

that plan gets passed along to the next step where it's executed by the processor



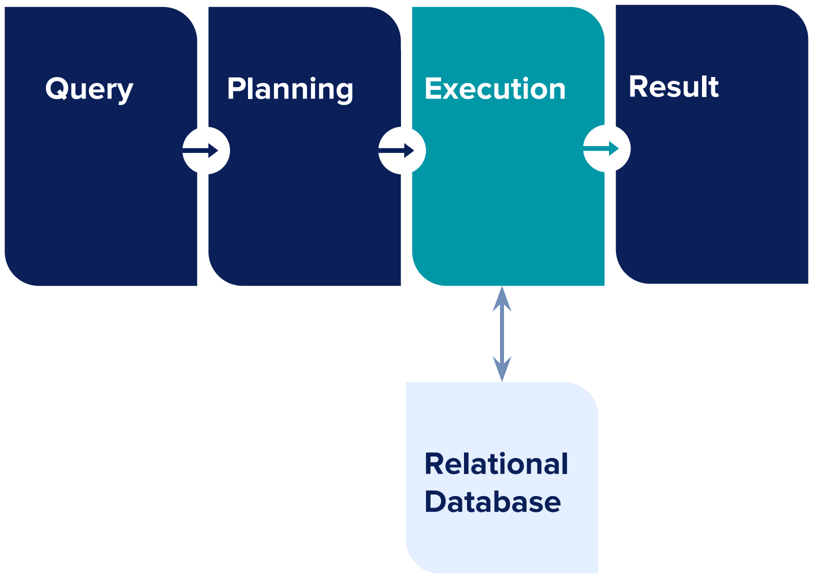
PROCESSORS

during execution, the processor will issue queries against our database



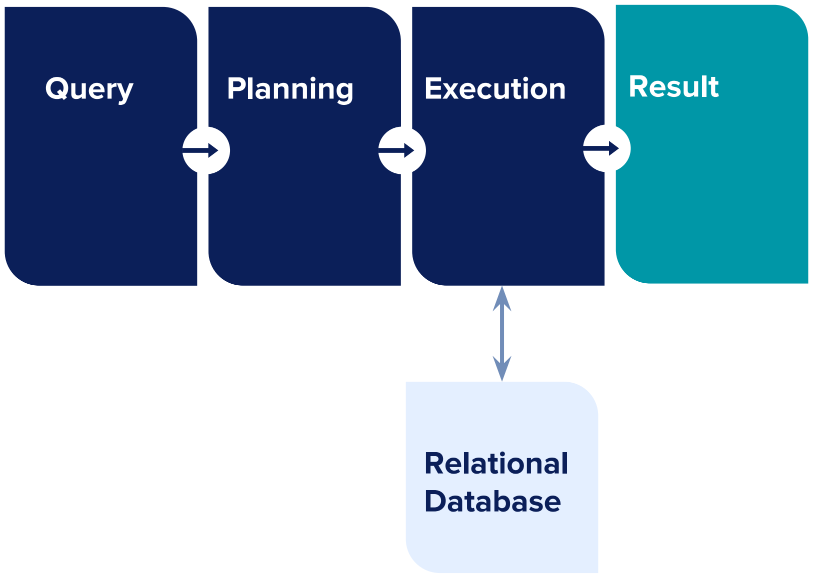
PROCESSORS

it'll take those intermediate query results and process them further to produce a final result



PROCESSORS

the last step is taking the final result and sending it back to our app

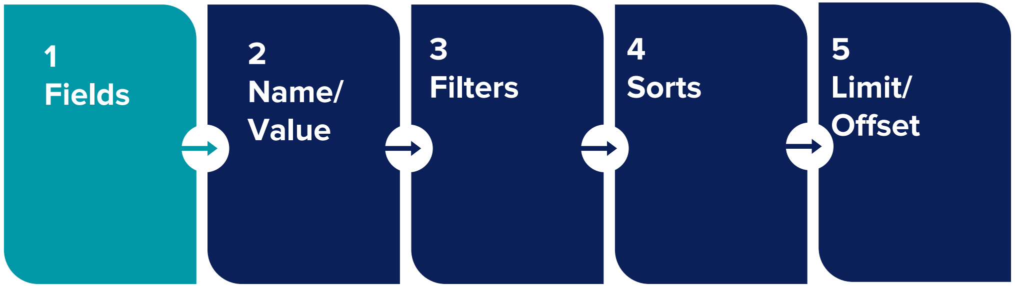


let’s look at how planning works first

the planner looks at our query in a specific order and builds a logical execution plan

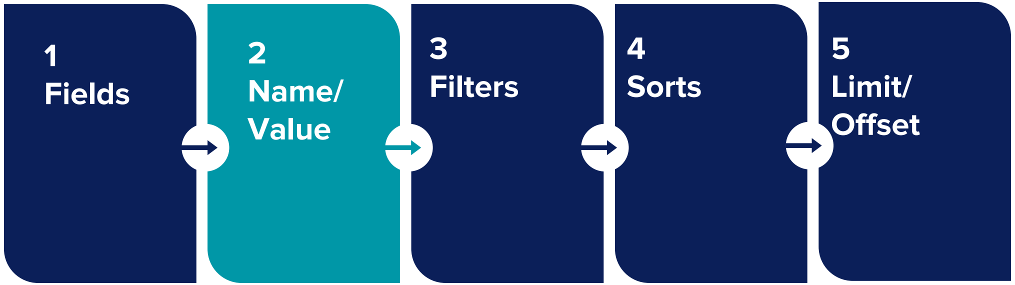
PROCESSORS

first, we go through the fields in each channel



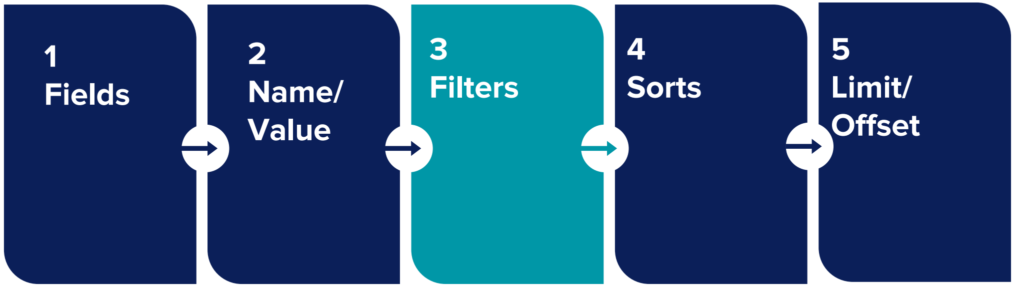
PROCESSORS

next, if we have names or values fields we add instructions for those



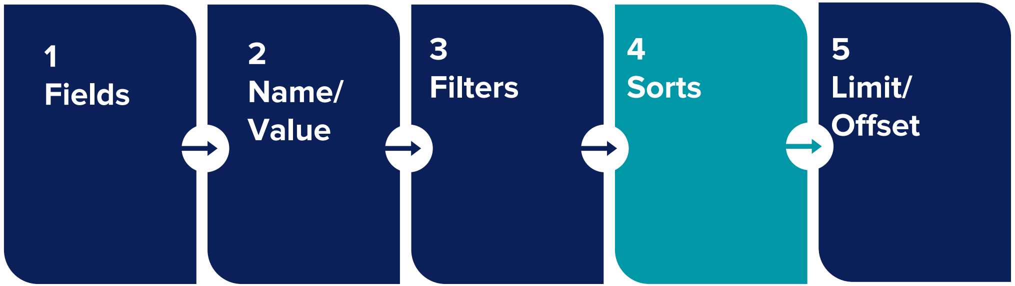
PROCESSORS

after that we plan all of the filters



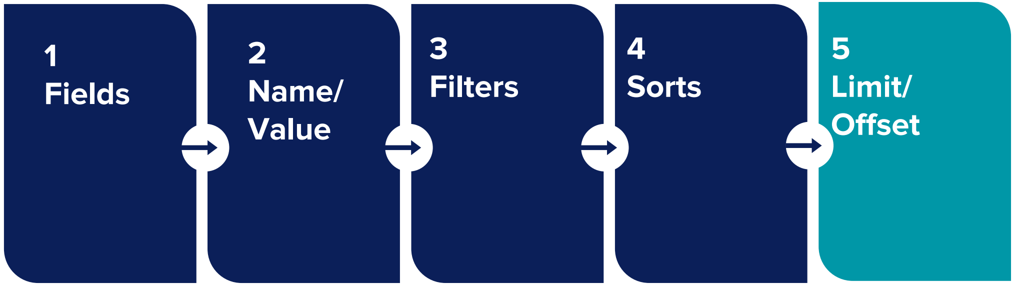
PROCESSORS

followed by sorts



PROCESSORS

finally we add a limit or offset if they're part of the query



as we go through each step the planner decides what we want to do in the database and what we want to process ourselves

how does it decide?

the planner always decides to "push-down" grouping and aggregate expressions and "pull-up" analytical expressions

an example is in our plans

DATAFLOW

let’s say we’ve got this field in our query

```
Field
1 + RUNNING_AVG(SUM([Price]) + 1)
```

DATAFLOW

this fragment is an aggregate expression

```
Field
1 + RUNNING_AVG(SUM([Price]) + 1)
               |________________|
```

DATAFLOW

an aggregate expression is any aggregate function and the operators attached to it

```
Field
1 + RUNNING_AVG(SUM([Price]) + 1)
```

DATAFLOW

this fragment is an analytic expression

```
Field
1 + RUNNING_AVG(SUM([Price]) + 1)
|______________|
```

DATAFLOW

an analytic expression is any analytic function and the operators attached to it

```
Field
1 + RUNNING_AVG(SUM([Price]) + 1)
|______________|
```

DATAFLOW

the planner will split this field into two parts

```
Field
1 + RUNNING_AVG(SUM([Price]) + 1)

Push-Down
?

Pull-Up
?
```

DATAFLOW

the first part is the aggregate expression that gets pushed down to the database

```
Field
1 + RUNNING_AVG(SUM([Price]) + 1)

Push-Down
SUM([Price]) + 1 AS C1

Pull-Up
?
```

DATAFLOW

the second part is the analytic expression that gets pulled up to the processor

```
Field
1 + RUNNING_AVG(SUM([Price]) + 1)

Push-Down
SUM([Price]) + 1 AS C1

Pull-Up
1 + RUNNING_SUM([C1])
```

DATAFLOW

expressions that get pushed down are given a unique alias that we use when we do our analytical processing

```
Field
1 + RUNNING_AVG(SUM([Price]) + 1)

Push-Down
SUM([Price]) + 1 AS C1

Pull-Up
1 + RUNNING_SUM([C1])
```

this is what we mean when we say processors are your database's analytical co-pilot

I know what you’re thinking

seems like a lot of trouble

why don’t we everything in the database?

organizations operate dozens of databases across almost as many vendors they don’t all support the same features in the same way

we want our processors to work the same way everywhere

we want a common data processing model we can rely on across all of the apps in our organization we’ve taken a “lowest common denominator” approach to solving this problem as long as our databases can do the basic stuff like select, group, aggregate, filter, and sort data, we can handle the rest it’s also a great opportunity to extend our processors to do more with our data things like regression, clustering, anova, etc.

this means delivering better applications more quickly to more people

time to execute an example

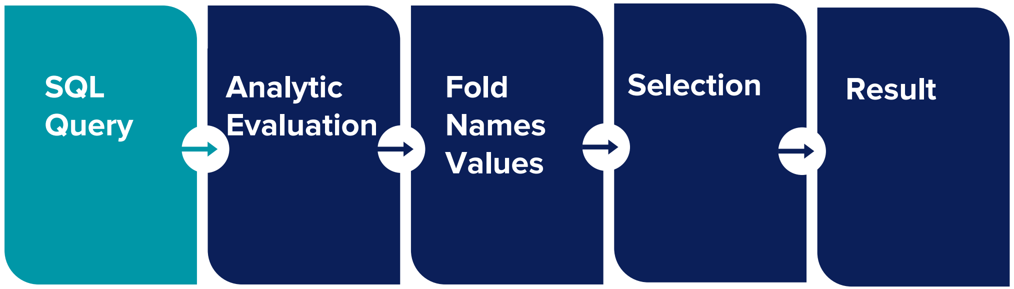
EXECUTION

we're going to walk through how we would execute a plan for our beautiful query

```
PivotQuery {
  x: [ "DATETRUNC('day', [Date])" ],
  y: [ "$[Values]" ],
  values: [
    "SUM([Price])",
    "RUNNING_SUM(SUM([Quantity]))"
  ],
  marks: [{
    field: "$[Values]",
    color: [ "$[Names]" ]
  }]
}
```

PROCESSORS

the first step is to execute our pushed down query against our relational database



EXECUTION

we’ve got three expressions here that get pushed down

```
PivotQuery {
  x: [ “DATETRUNC(‘day’, [Date])” ],
  y: [ “$[Values]” ],
  values: [
    “SUM([Price])”,
    “RUNNING_SUM(SUM([Quantity]))”
  ],
  marks: [{
    field: “$[Values]”,
    color: [ “$[Names]” ]
  }]
}
```

EXECUTION

the first is the truncated date on our x-axis

```
PivotQuery {
  x: [ “DATETRUNC(‘day’, [Date])” ],
  y: [ “$[Values]” ],
  values: [
    “SUM([Price])”,
    “RUNNING_SUM(SUM([Quantity]))”
  ],
  marks: [{
    field: “$[Values]”,
    color: [ “$[Names]” ]
  }]
}
```

EXECUTION

the second is the sum of price in values

```
PivotQuery {
  x: [ “DATETRUNC(‘day’, [Date])” ],
  y: [ “$[Values]” ],
  values: [
    “SUM([Price])”,
    “RUNNING_SUM(SUM([Quantity]))”
  ],
  marks: [{
    field: “$[Values]”,
    color: [ “$[Names]” ]
  }]
}
```

EXECUTION

the third is the sum of quantity used in the running_sum function also in values

```
PivotQuery {
  x: [ “DATETRUNC(‘day’, [Date])” ],
  y: [ “$[Values]” ],
  values: [
    “SUM([Price])”,
    “RUNNING_SUM(SUM([Quantity]))”
  ],
  marks: [{
    field: “$[Values]”,
    color: [ “$[Names]” ]
  }]
}
```

EXECUTION

just like before, these expressions get pushed down to the database resulting in this beautiful sql query

```
SELECT DATETRUNC('day', date) AS C1
       SUM(price) AS C2,
       SUM(quantity) AS C3
  FROM orders
 GROUP BY DATETRUNC('day', date)
```

EXECUTION

the table name comes from our data model which is used during planning to tell the processor about our database schema

```sql
SELECT DATETRUNC('day', date) AS C1
       SUM(price) AS C2,
       SUM(quantity) AS C3
  FROM orders
 GROUP BY DATETRUNC('day', date)
```

EXECUTION

we’ve saved details about that part for next time

```
SELECT DATETRUNC('day', date) AS C1
       SUM(price) AS C2,
       SUM(quantity) AS C3
  FROM orders
 GROUP BY DATETRUNC('day', date)
```

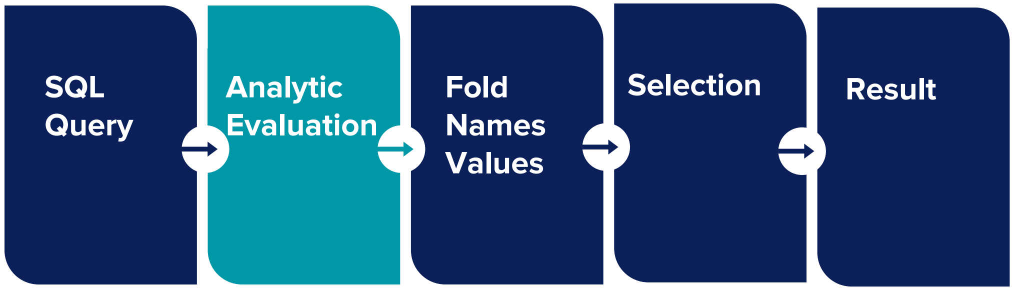
EXECUTION

this is what our database gives us back

DAY(Date)	SUM(price)	SUM(quantity)
2019-01-01	10	15
2019-01-02	5	5
...	...	...

PROCESSORS

we take that and process our analytic expressions



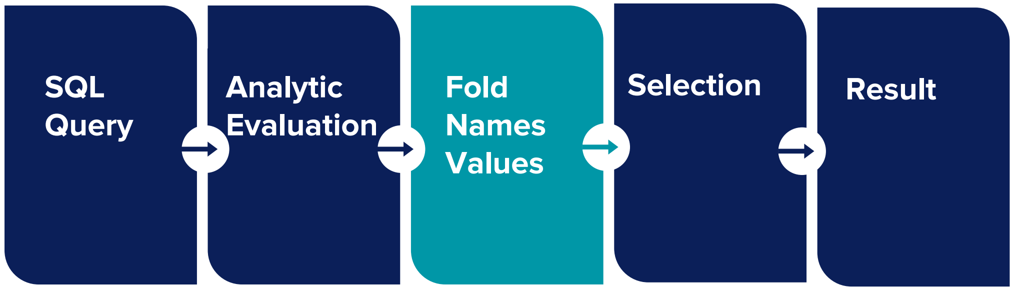
EXECUTION

this gives us a new attribute as we can see here

DAY(Date)	SUM(price)	SUM(quantity)	RUNNING_SUM(quantity)
2019-01-01	10	15	15
2019-01-02	5	5	20
...	...	...	

PROCESSORS

since we have names and values, we use a fold transform to “unpivot” the result



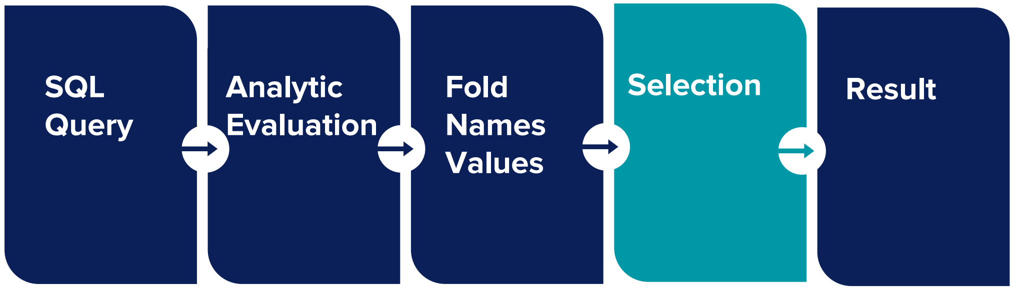
EXECUTION

this adds two new fields, names and values, and also expands the number of rows in the result

Names	Values	DAY (Date)	SUM (price)	SUM (quantity)	RUNNING_SUM (quantity)
SUM(price)	10	2019-01-01	10	15	15
RUNNING_SUM (quantity)	15	2019-01-01	10	5	15
SUM(price)	5	2019-01-02	5	15	20
RUNNING_SUM (quantity)	20	2019-01-02	5	5	20
...	...	...	...		...

PROCESSORS

almost done! next we slice out any intermediate fields that we don't want which we call selection



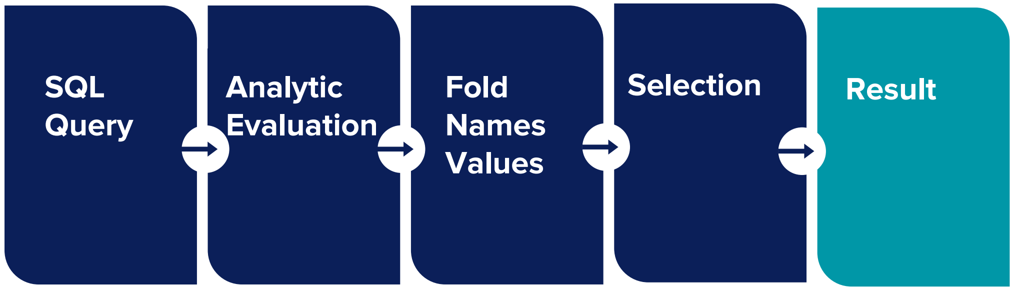
EXECUTION

that gives us this beautiful result

Names	Values	DAY(Date)
SUM(price)	10	2019-01-01
RUNNING_SUM(quantity)	15	2019-01-01
SUM(price)	5	2019-01-02
RUNNING_SUM(quantity)	20	2019-01-02
...	...	...

PROCESSORS

last but not least, we take our result and send it back to the app



```
PivotQuery {
  x: [ “DATETRUNC(‘day’, [Date])” ],
  y: [ “$[Values]” ],
  values: [
    “SUM([Price])”,
    “RUNNING_SUM(SUM([Quantity]))”
  ],
  marks: [{
    field: “$[Values]”,
    color: [ “$[Names]” ]
  }]
}
```

Names	Values	DAY(Date)
SUM(price)	10	2019-01-01
RUNNING_SUM(quantity)	15	2019-01-01
SUM(price)	5	2019-01-02
RUNNING_SUM(quantity)	20	2019-01-02
...	...	...

and that’s how the tables turn

not having to move data around or materialize all our views ahead of time lets us effectively use 1000x more data

where does that leave us?

using formulas is 100x faster than writing etl pipelines or app code

using queries is 10x faster than building ad-hoc transformation code for visualizations

using processors lets us work with 1000x more data

we did the math and that’s 1,000,000x

I rest my case, your honor

in all seriousness we think this can be incredibly valuable for data teams everywhere

where are we going to go from here?

ROADMAP

where are we going from here?

- Release the code under open license

ROADMAP

where are we going from here?

- Release the code under open license
- **Expand the built-in function library**

ROADMAP

where are we going from here?

- Release the code under open license
- Expand the built-in function library
- **Build out more real-world examples**

ROADMAP

where are we going from here?

- Release the code under open license
- Expand the built-in function library
- Built out more real-world examples
- **Expand our database adapter library**

ROADMAP

where are we going from here?

- Release the code under open license
- Expand the built-in function library
- Build out more real-world examples
- Expand our database adapter library
- **Integrate with open tools like DBT**

ROADMAP

where are we going from here?

- Release the code under open license
- Expand the built-in function library
- Build out more real-world examples
- Expand our database adapter library
- Integrate with open tools like DBT
- **Integrate with libraries like vega-lite**

ROADMAP

where are we going from here?

- Release the code under open license
- Expand the built-in function library
- Build out more real-world examples
- Expand our database adapter library
- Integrate with open tools like DBT
- Integrate with libraries like vega-lite
- **Build common components for frameworks like angular, react, native, etc.**

how do we get involved?

COMMUNITY

first of all, go here 👇

github.com/switch-data/community

COMMUNITY

AND HIT THE STAR BUTTON 🌟

github.com/switch-data/community

COMMUNITY

you won’t regret it

github.com/switch-data/community

COMMUNITY

we’re gonna update it with links, resources, issues, slack chat info, etc.

COMMUNITY

how can I get involved?

COMMUNITY

I want to go to meetups

COMMUNITY

I want to visit organizations

COMMUNITY

I want to understand our challenges

thank you again

Q&A

Come see me during office hours!