Introducing Switch: a framework for custom data applications

Josh Ferguson
Chief Architect @ Mode

josh@modeanalytics.com

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

i’ve been obsessed with building data tools for about 20 years

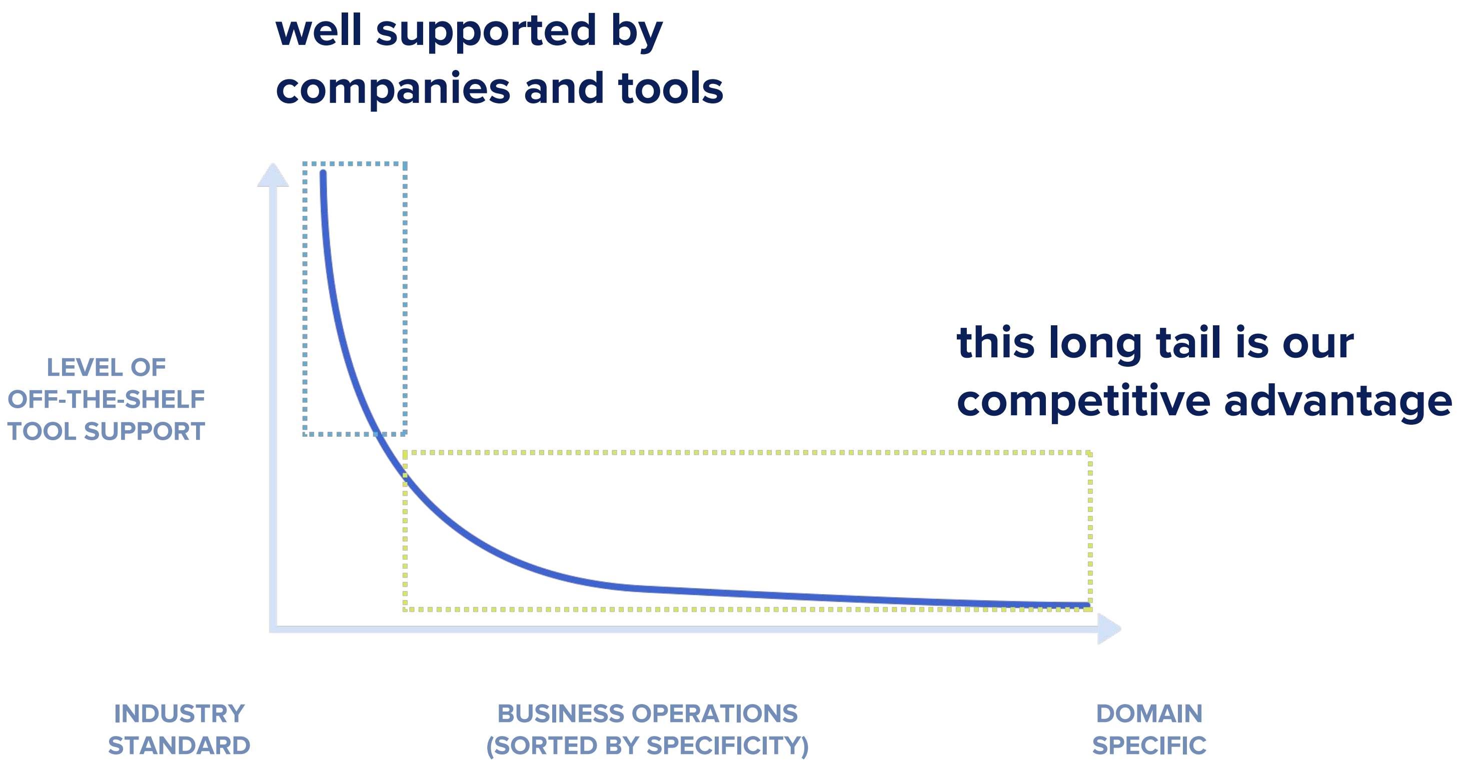
@besquared almost everywhere

mode(.com)

a collaborative data science platform

our users are data scientists, analysts, and engineers help everybody make better decisions with data we’re here to talk about data applications custom data applications

what’s a custom data application?


well supported by
companies and tools
this long tail is our
competitive advantage
LEVEL OF
OFF-THE-SHELF
TOOL SUPPORT
INDUSTRY
STANDARD
BUSINESS OPERATIONS
(SORTED BY SPECIFICITY)
DOMAIN
SPECIFIC

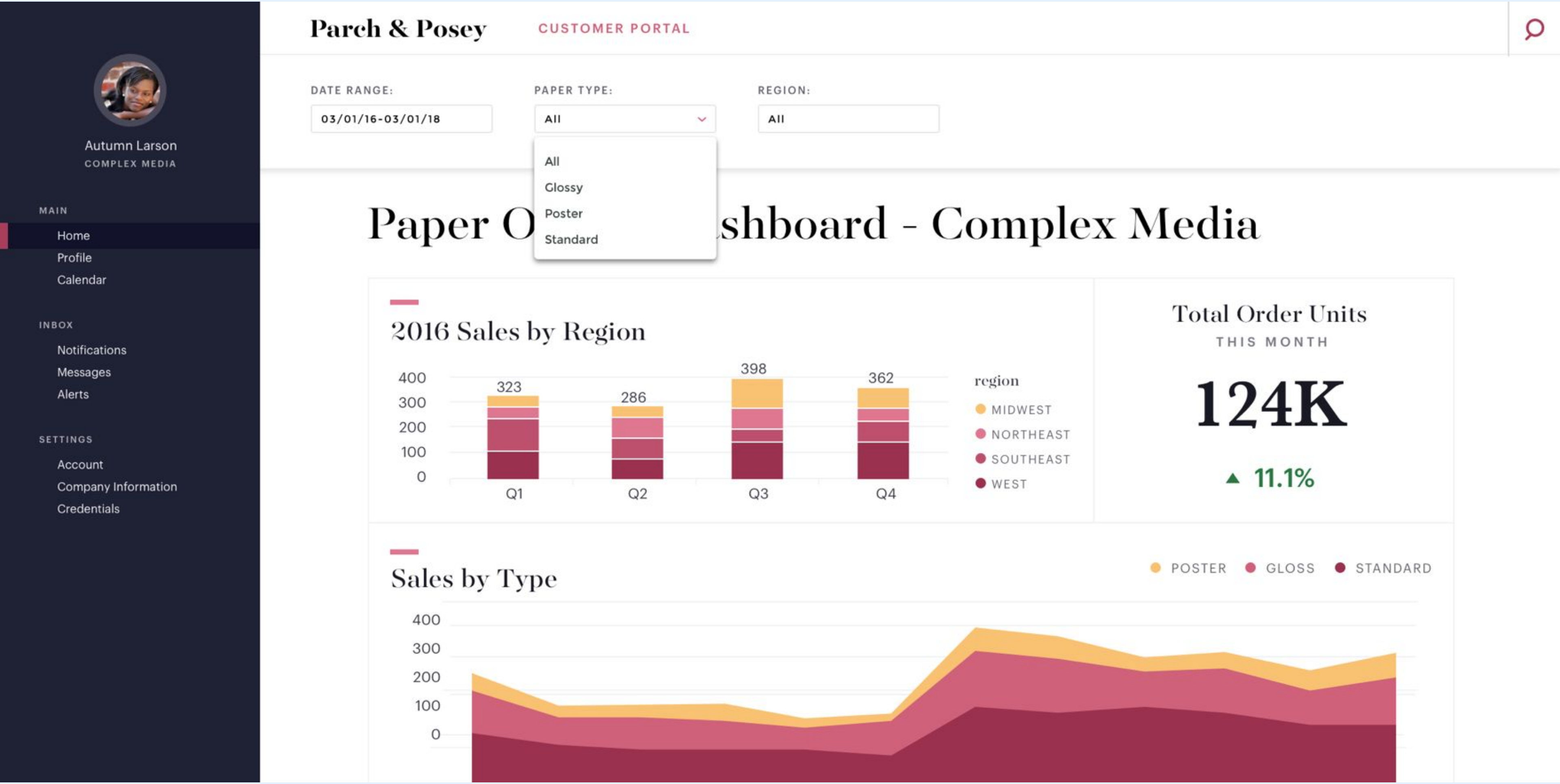
there's no collection of off-the-shelf tools that will provide everything our organization needs to make better decisions with data

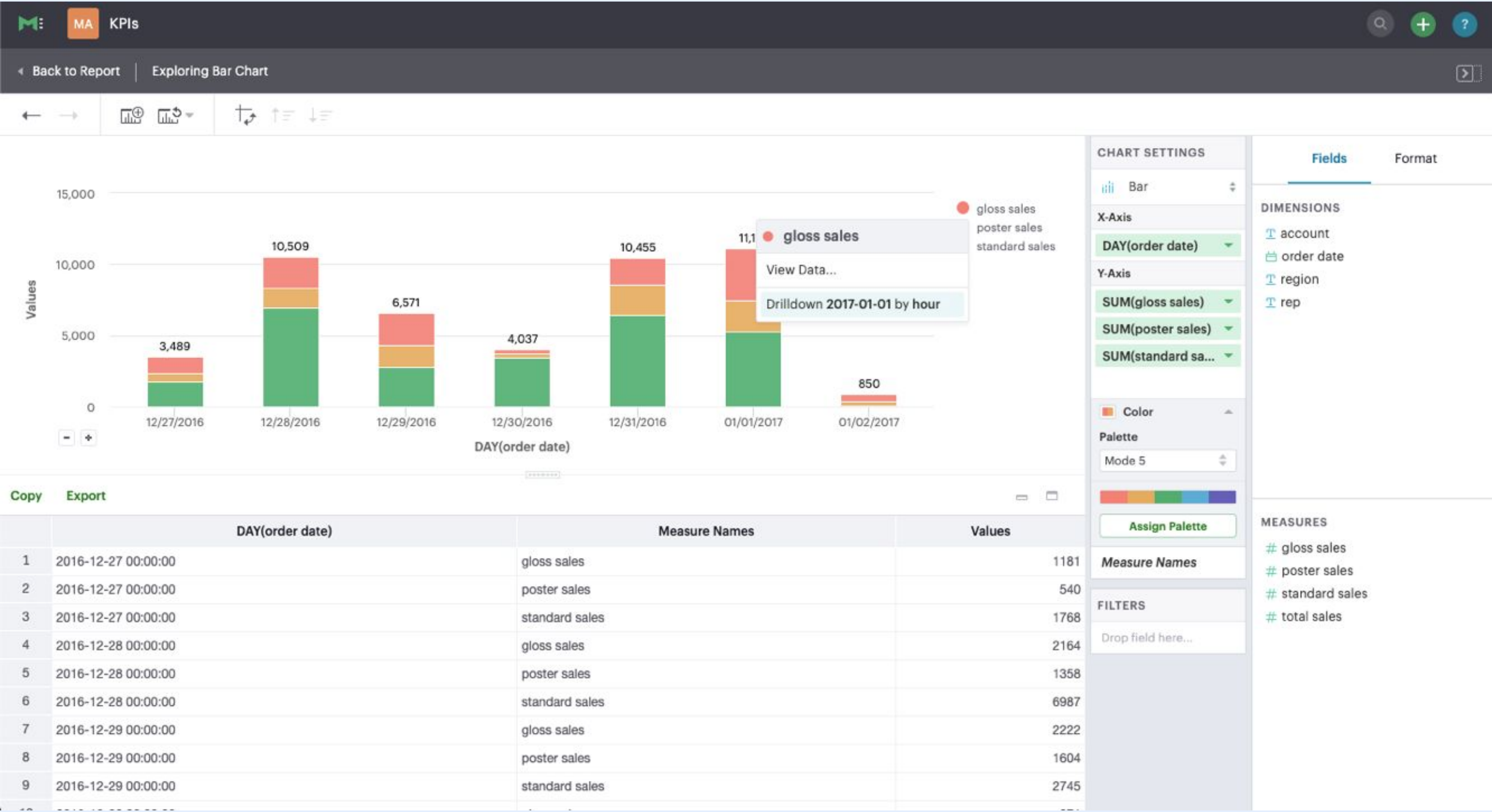
this is where we should focus

logistics tracking and monitoring

customer health monitoring tools for success

a/b testing tools for our product team


Parch & Posey
CUSTOMER PORTAL
Autumn Larson
COMPLEX MEDIA
MAIN
Home
Profile
Calendar
INBOX
Notifications
Messages
Alerts
SETTINGS
Account
Company Information
Credentials
DATE RANGE:
03/01/16-03/01/18
PAPER TYPE:
All
All
Glossy
Poster
Standard
REGION:
All
Paper O
shboard - Complex Media
2016 Sales by Region
400
300
200
100
0
323
286
398
362
Q1
Q2
Q3
Q4
region
MIDWEST
NORTHEAST
SOUTHEAST
WEST
Total Order Units
THIS MONTH
124K
11.1%
Sales by Type
POSTER
GLOSS
STANDARD
400
300
200
100
0



	DAY(order date)	Measure Names	Values
1	2016-12-27 00:00:00	gloss sales	1181
2	2016-12-27 00:00:00	poster sales	540
3	2016-12-27 00:00:00	standard sales	1768
4	2016-12-28 00:00:00	gloss sales	2164
5	2016-12-28 00:00:00	poster sales	1358
6	2016-12-28 00:00:00	standard sales	6987
7	2016-12-29 00:00:00	gloss sales	2222
8	2016-12-29 00:00:00	poster sales	1604
9	2016-12-29 00:00:00	standard sales	2745

everyone one of these apps is a one-off today

🤑 🕑 🚑

switch is a collection of typescript libraries and tools that let us build richer and more interactive data applications

the data layer between our database and our user interface

it lets us address some of the major challenges we face when we’re building our data apps

challenge number one

CHALLENGE #1

our users always want to slice and dice their data in ways that we don’t anticipate

CHALLENGE #1

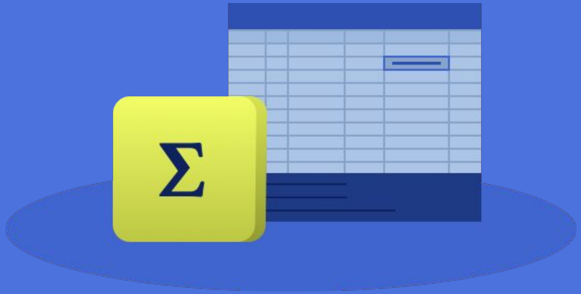
we don’t know what we’ll need ahead of time

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 give our users the tools they need to quickly and easily express data in new and different ways on their own



Introducing Formulas

an excel-like language for data expression

they let our users build custom calculations, even if they're not database or programming language experts

what can they 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!?

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

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

we can build interfaces that let users extend our apps with their own business logic and calculations for example at Mode we’re working on a formula editor that lets our users add custom calculations to their visualizations 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 every time we want to answer a different question amplifies 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

a common characteristic of custom data apps is custom data visualizations

CHALLENGE #2

we don’t want to write ad-hoc data transformation code every time we want to build a visualization

CHALLENGE #2

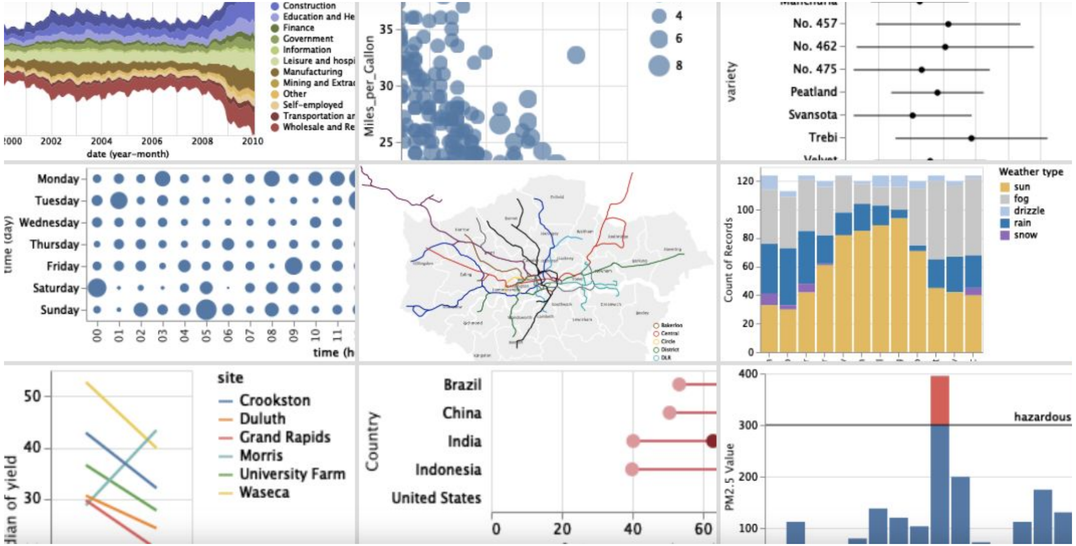
we should use a language that let’s us describe the data we need in way that matches the visualizations we’re trying to build

Introducing Queries

our queries speak the language of data visualization

QUERIES

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?

we define the data we want in our query

QUERIES

we use fields which are defined with a formula

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

QUERIES

they let us describe the data and calculations we want to get back in our query result

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

QUERIES

they’re the atomic unit of data in a query

```
“SUM([Quantity])”

“[Price] / [Quantity]”

“DATETRUNC(‘day’, [Date])”
```

QUERIES

there are two pre-defined fields 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]”;
}
```

QUERIES

we’ve got filters

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

QUERIES

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

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

QUERIES

we’ve got 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;
}
```

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 describe the layers of our visualization

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

MARKS

every layer is 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 map fields to 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

marks, filters, and sorts

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

    values: Field[];

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

PIVOT QUERY

more channels

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

    values: Field[];

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

PIVOT QUERY

column and row which let us facet data across or down our visualization

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

    values: Field[];

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

PIVOT QUERY

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

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

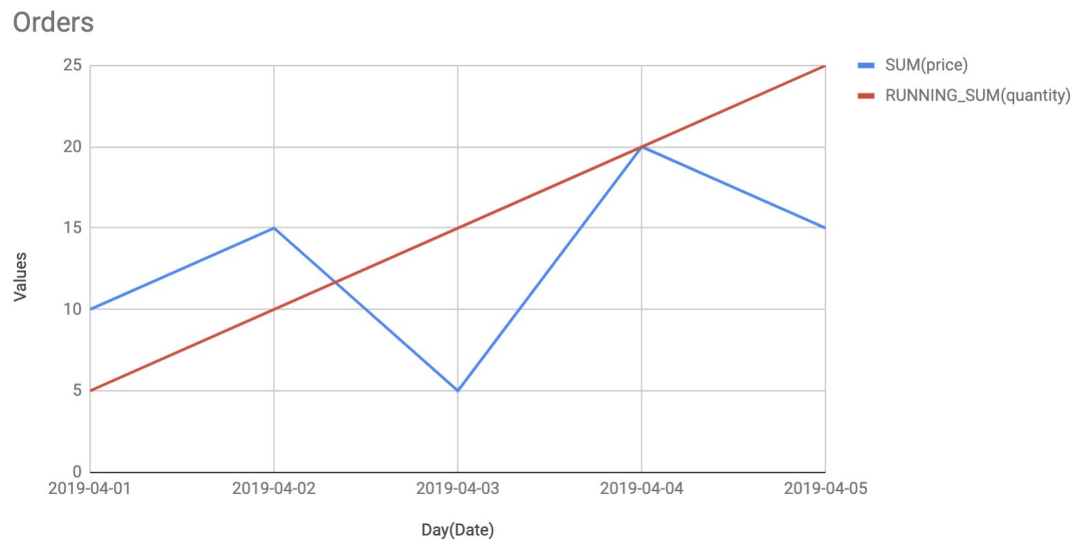
```
PivotQuery {
    column: Field[];
         x: Field[];
       row: Field[];
         y: Field[];

    values: Field[];

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

QUERIES

a beautiful chart



QUERIES

a beautiful query

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

EXAMPLE

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

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

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

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

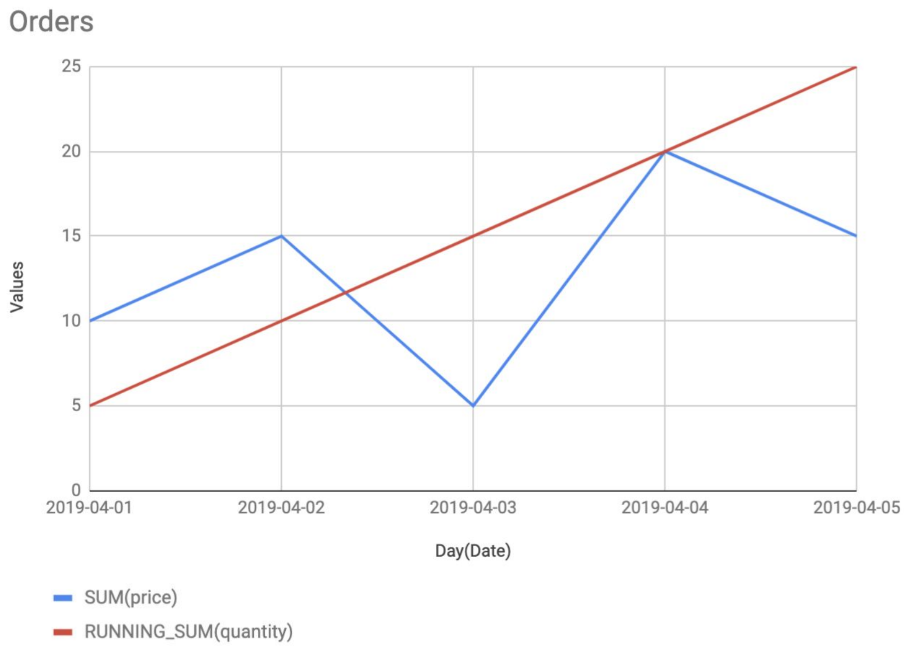
defined by our values field

```
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 try to materialize everything we might need to analyze ahead of time

CHALLENGE #3

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

Introducing Processors

they’re the secret sauce

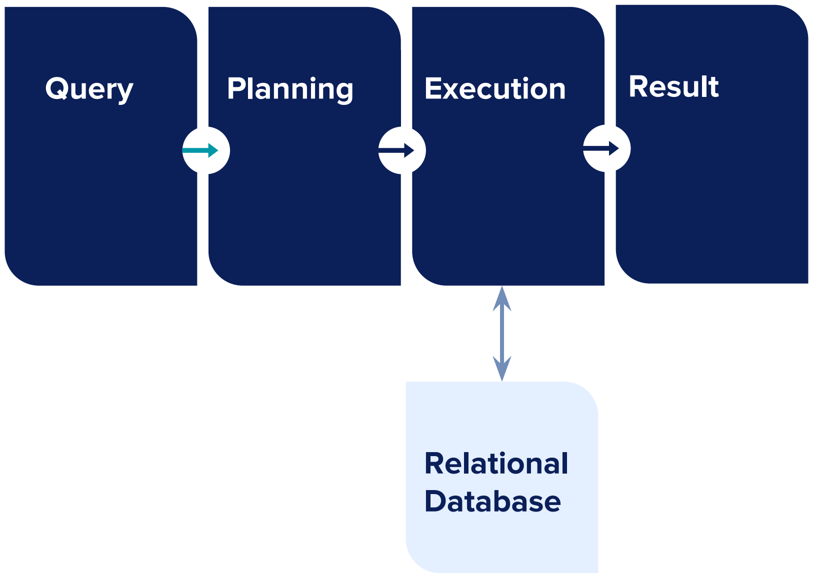
they make it possible for our data apps to take advantage of the high performance and massive scale of the databases we already have they’re our database’s analytical co-pilots

what do we mean by that?

let’s talk about how they work

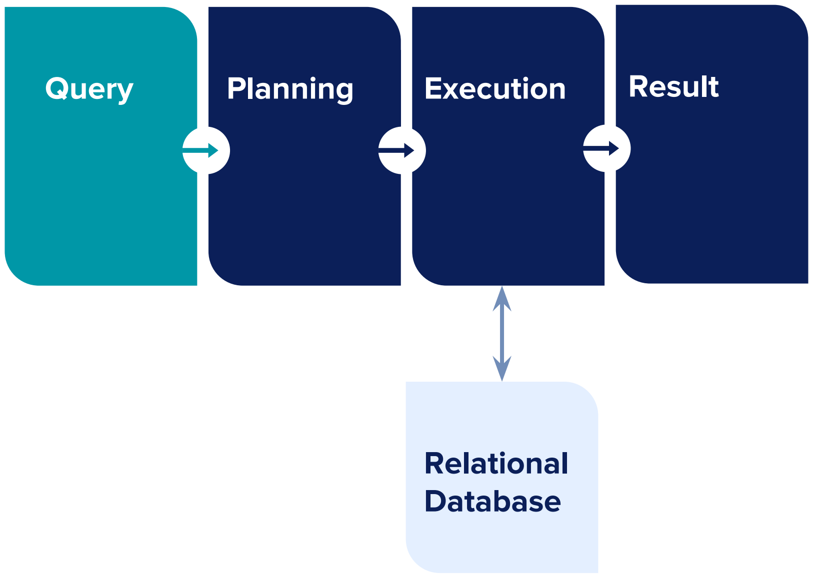
PROCESSORS

processors take in queries and compute results



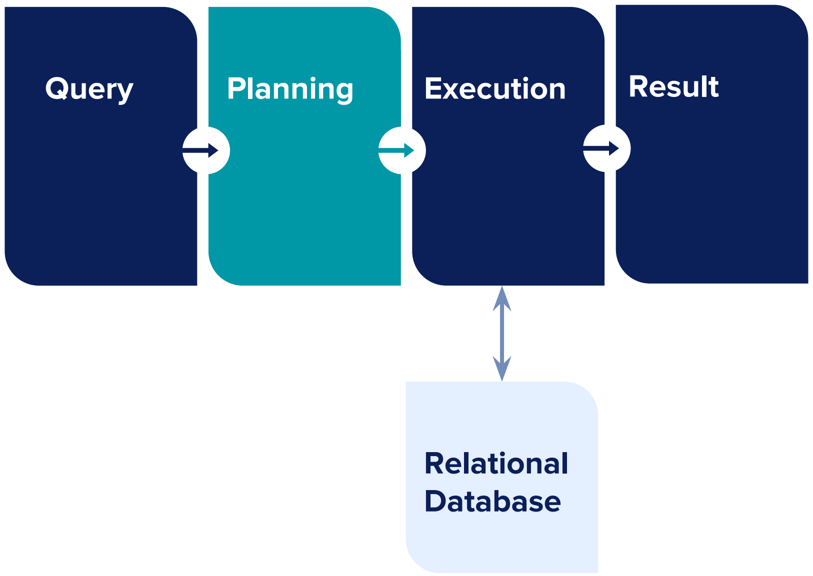
PROCESSORS

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



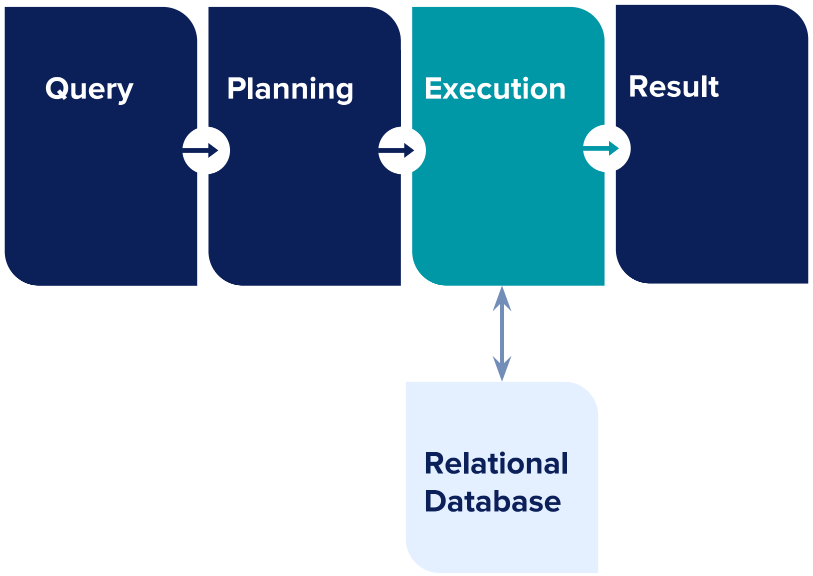
PROCESSORS

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



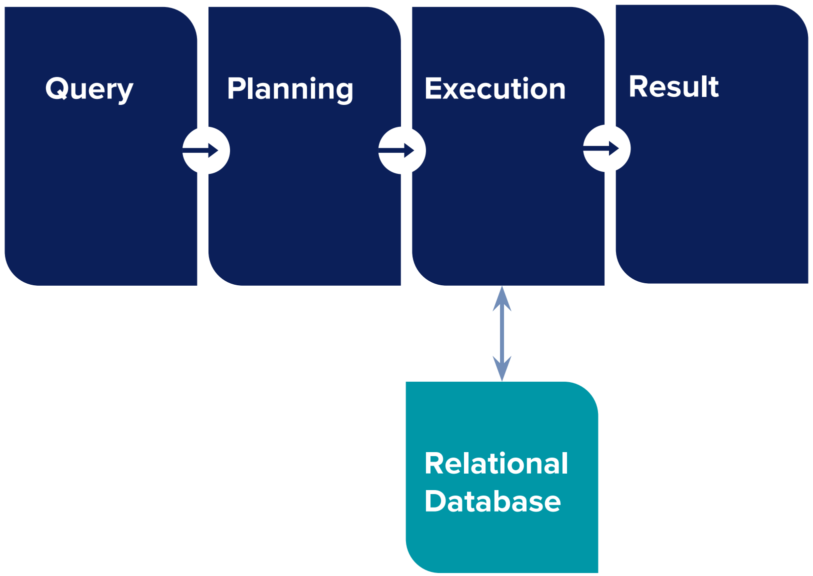
PROCESSORS

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



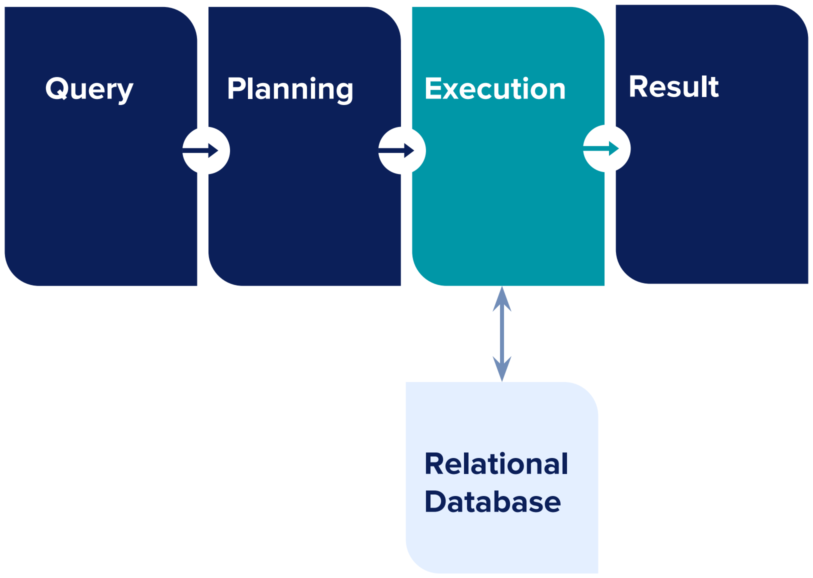
PROCESSORS

during execution, the processor will issue queries against our database



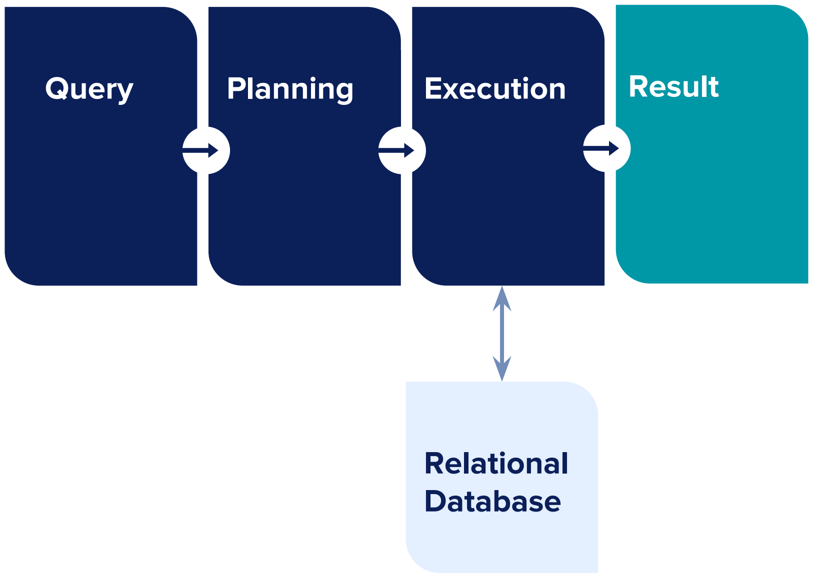
PROCESSORS

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

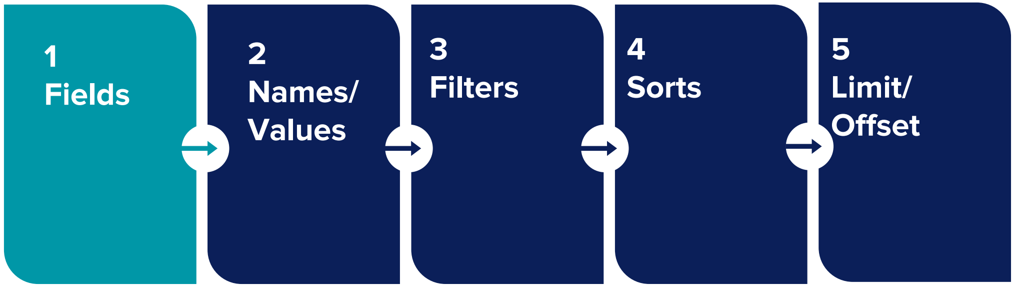


let’s look at how planning works first

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

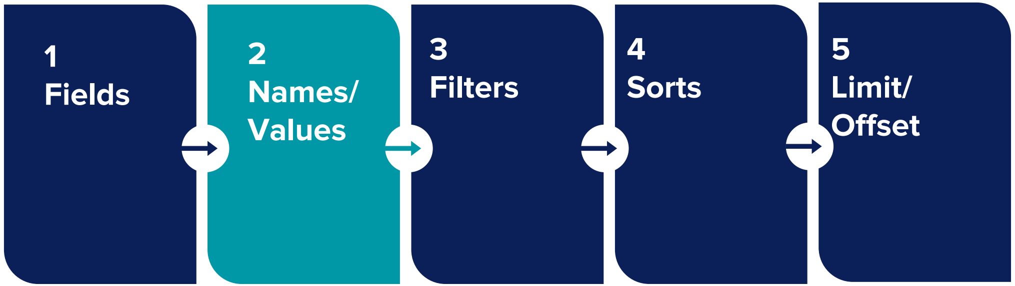
PROCESSORS

we go through the fields in each channel



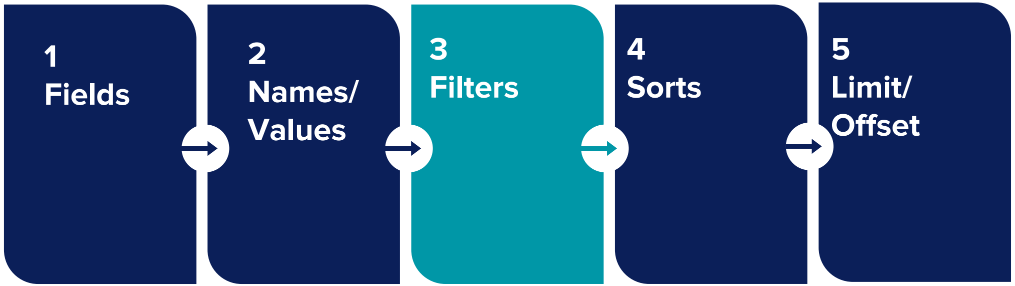
PROCESSORS

if we have names or values fields we add those to the plan



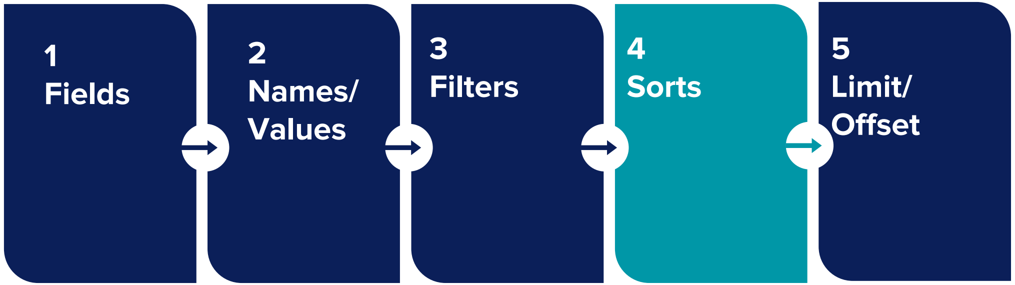
PROCESSORS

after that we plan all of the filters



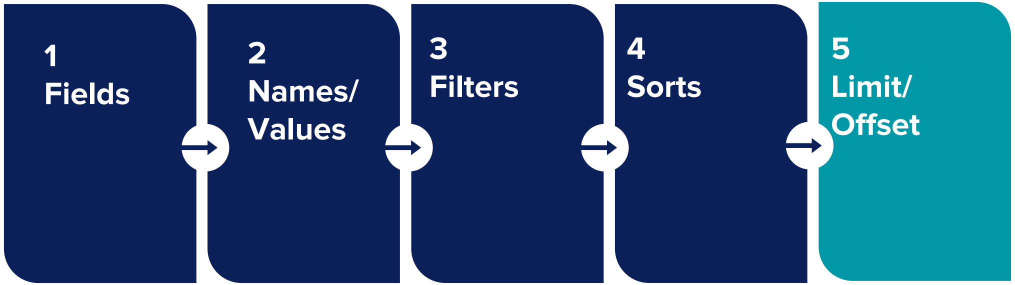
PROCESSORS

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 parts of the query we want to process in the database and what we want to process on the “client” how does it decide?

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

DATAFLOW

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

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

DATAFLOW

this 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 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 aggregate expression gets pushed down to the database

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

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

Pull-Up
?
```

DATAFLOW

the analytic expression 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 are pushed down get a unique alias that we use to reference the results

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

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

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

why don’t we do everything in the database?

organizations operate dozens of databases across almost as many vendors

we want a common data processing model that we can rely on across all of the apps in our organization as long as our databases can do basic stuff like select, group, aggregate, filter, and sort, we can handle the rest

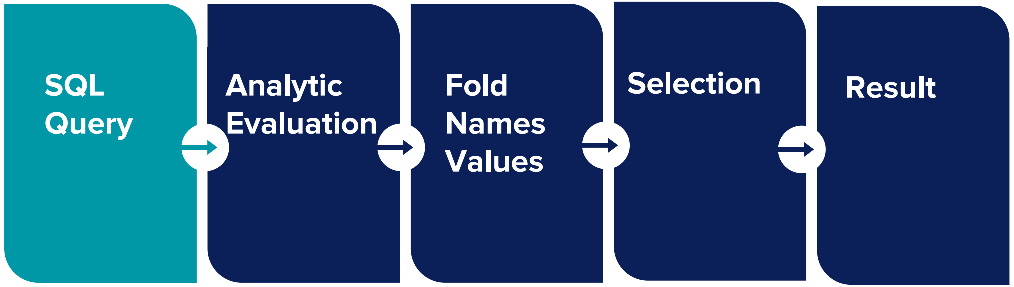
EXECUTION

we're going to walk through how we would execute the 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

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

day on our x-axis

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

EXECUTION

sum of price on values

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

EXECUTION

sum of quantity also from values

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

EXECUTION

that gives us 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 a data model which let’s the processor know about our database schema

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

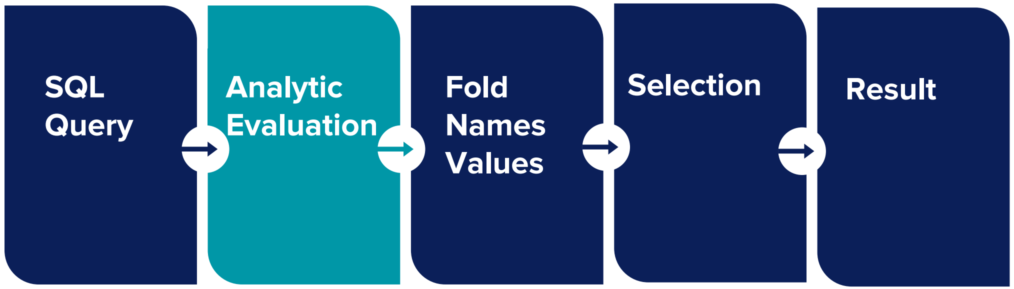
EXECUTION

this is what our database hands back

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

PROCESSORS

we take that and evaluate our analytic expressions



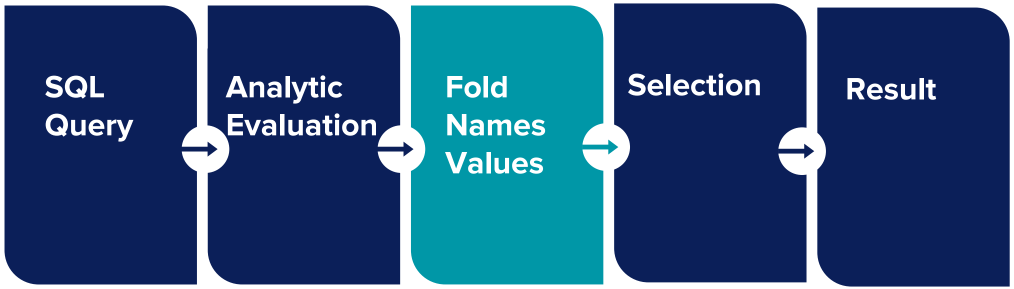
EXECUTION

+ running_sum

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

PROCESSORS

we use a fold transform to “unpivot” the result



EXECUTION

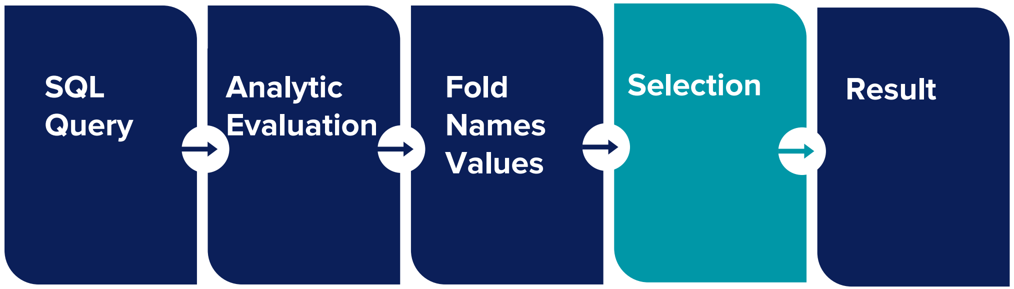
+ names
+ values

expand the number of rows

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

we select just the fields that we want in our result



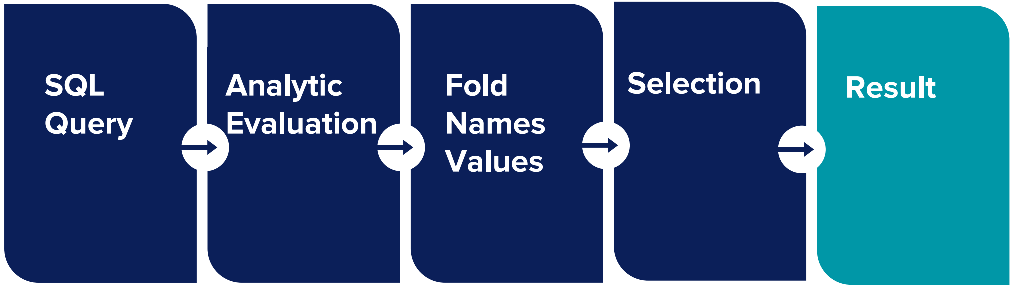
EXECUTION

- sum price
- sum quantity
- running_sum

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

results go 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

this strategy pays big dividends

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

where does that bring us?

a familiar excel-like formula language that lets our users explore data in different ways without new etl pipelines or app code

100x

a visual query language that lets us ask for the data we need in a way that matches the visualizations we’re trying deliver

10x

data processors that let us deploy our visualization queries on top of the high performance databases we already have

1000x

1,000,000x

a game changer for data teams and decision makers

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.**

COMMUNITY

how do I get involved?

COMMUNITY

head on over here 👇

github.com/switch-data/community

COMMUNITY

AND HIT THE STAR BUTTON 🌟

github.com/switch-data/community

COMMUNITY

how can I help you?

thank you again

Q&A

Come see me during office hours!