From pipelines to refineries: scaling big data applications

Tim Hunter

databricks

About Me

- Tim Hunter
- Software engineer @ Databricks
- Ph.D. from UC Berkeley in Machine Learning
- Very early Spark user
- Contributor to MLlib
- Author of TensorFrames and GraphFrames

Introduction

- Spark 2.2 in the release process
- Spark 2.3 being thought about
- This is the time to discuss Spark 3
- This presentation is a personal perspective on a future Spark

Introduction

There is nothing more practical than a good theory.

James Maxwell

As Spark applications grow in complexity, what challenges lie ahead.

Can we find some good foundations for building big data frameworks?

databricks

What category does Spark live in?

A monad?

A functor?

An applicative?

Introduction

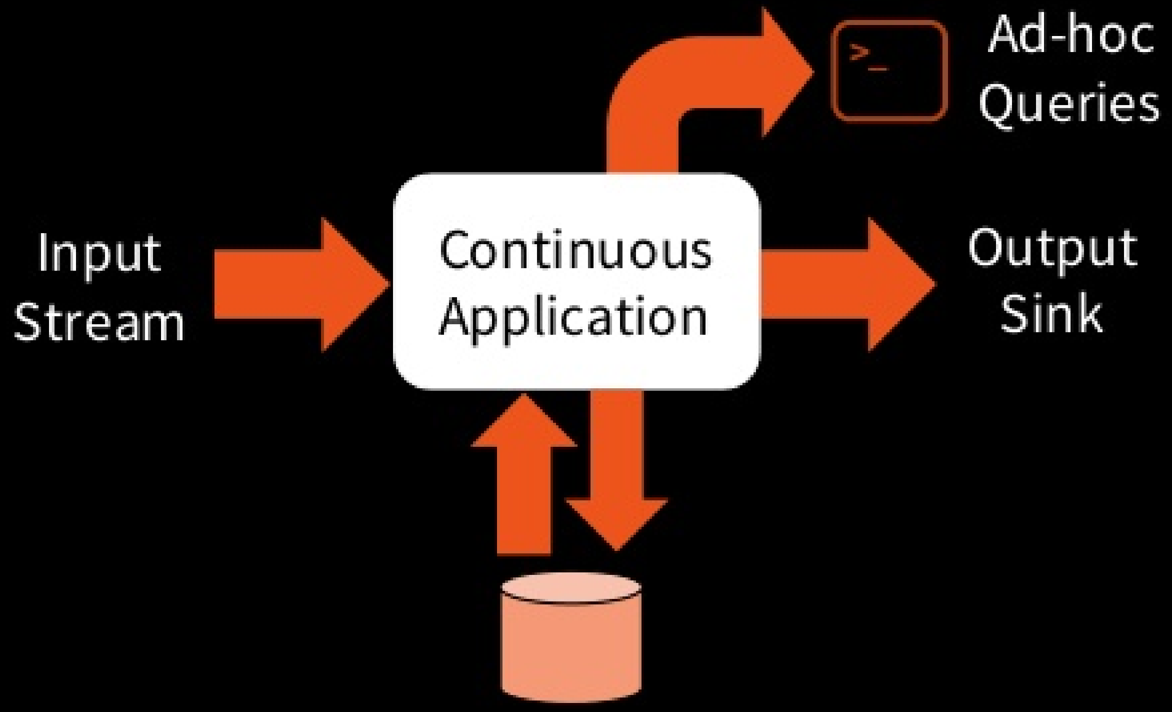
- Anything presented here can be implemented in a modern programming language
- No need to know Haskell or category theory

Outline

- State of the union
  - What is good about Spark?
  - What are the trends?
- Classics to the rescue
  - Fighting the four horsemen of the datapocalypse
  - Laziness to the rescue
- From theory to practice
  - Making data processing great again

State of the union

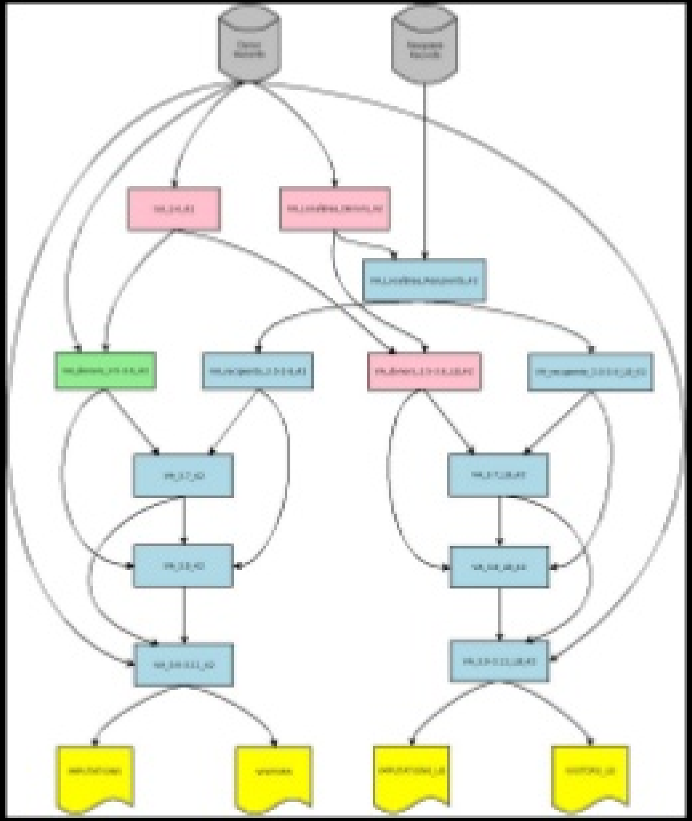
- What we strive for



databricks

State of the union

- What we deal with:
  - Coordinating a few tasks



databricks

State of the union

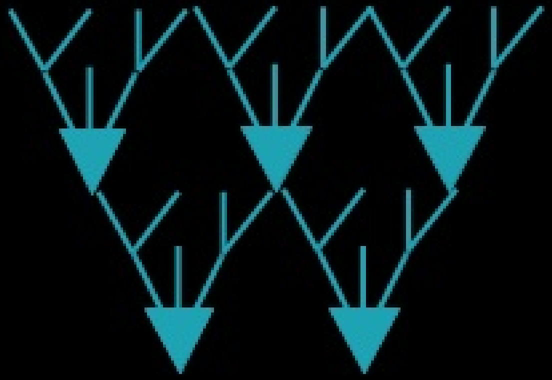
- The (rapidly approaching) future
  - Thousands of input sources
  - Thousands of concurrent requests
  - Mixing interactive, batch, streaming

- How do we enable this?

The state of the union

- The image of a pipeline gives you the illusion of simplicity
  - One input and one output

- Current big data systems: the tree paradigm
  - Combine multiple inputs into a single output
  - The SQL paradigm
  - Followed by Spark



- A forest is more than a set of trees
  - Multiple inputs, multiple outputs
  - The DAG paradigm

Data processing as a complex system

- Orchestrating complex flows is nothing new:
  - Tracking and scheduling: Gantt, JIRA, etc.
  - Build tools: Bazel (Blaze), Pants, Buck
  - General schedulers: AirFlow, FBLearner, Luigi
- Successfully used for orchestrating complex data processing
- But they are *general* tools:
  - They miss the fact that we are dealing with *data* pipelines
  - Separate system for scheduling and for expressing transforms
  - No notion of schema, no control of the side effects

Data processing as a complex system

- Any software becomes an exercise in managing complexity.
  - Complexity comes from interactions
  - How to reduce interactions?
- How to build more complex logic based on simpler elements?
  - How to compose data pipelines?
- Continuous evolution
  - Let's change the engines in mid-flight, because it sounds great

The ideal big processing system:

- *Scalability*
  - in quantity (big data) and diversity (lots of sources)
- *Chaining*
  - express the dependencies between the datasets
- *Composition*
  - assemble more complex programs out of simpler ones

- *Determinism*
  - given a set of input data, the output should be unique*
- *Coffee break threshold*
  - quick feedback to the user

How is Spark faring so far?

- You *can* do it, but it is not easy
- Spark includes multiple programming models:
- The RDD model: low-level manipulation of distributed datasets
  - Mostly imperative with some lazy aspects
- The Dataset/Dataframes model:
  - Same as RDD, with some restrictions and a domain-specific language
- The SQL model: only one output, only one query

What can go wrong with this program?

```
all_clicks = session.read.json("/tables/clicks/year=2017")
all_clicks.cache()
max_session_duration = all_clicks("session_duration").max()
top_sessions = all_clicks.filter(
            all_clicks("session_duration") >= 0.9 * max_session_duration)
top_ad_served = top_sessions("ad_idd")
top_ad_served.write.parquet("/output_tables/top_ads")
```

leak

a few hours…

typo

missing directory

The 4 horsemen of the datapocalypse

- Eager evaluation
- Missing source or sink
- Resource leak
- Typing (schema) mismatch

Classics to the rescue

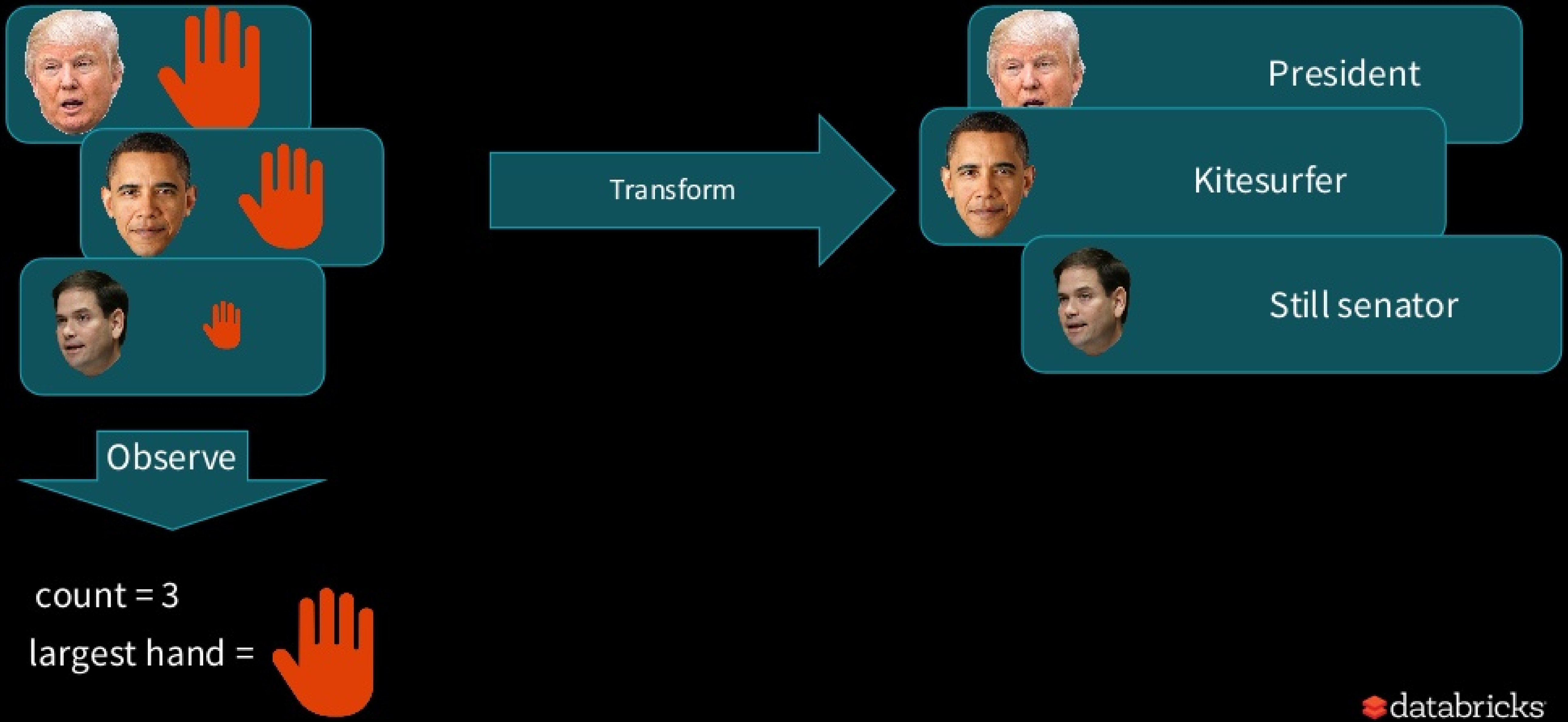
Theoretical foundations for a data system

- A dataset is a collection of elements, all of the same type
  - Scala: Dataset[T]

- Principle: the content of a dataset cannot be accessed directly
  - A dataset can be *queried*

- An observable is a single element, with a type
  - intuition: dataset with a single row
  - Scala: Observable[T]

Theoretical foundations for a data system



databricks

Theoretical foundations for a data system

- *Principle*: the observation only depends on the content of the dataset
  - You cannot observe partitions, ordering of elements, location on disk, etc.

- Mathematical consequence: all reduction operations on datasets are monoids:
  - $f(A \cup B) = f(A) + f(B) = f(B) + f(A)$
  - $f(\text{empty}) = 0$

Theoretical foundations for a data system

- *Principle*: closed world assumption
  - All the effects are modeled within the framework
  - The inputs and the transforms are sufficient to generate the outputs
- Practical consequence: strong checks and sci-fi optimizations

Examples of operations

- They are what you expect:
  - Dataset[Int] : a dataset of integers
  - Observable[Int] : an observation on a dataset
- max: Dataset[Int] => Observable[Int]
- union: (Dataset[Int], Dataset[Int]) => Dataset[Int]
- collect: Dataset[Int] => Observable[List[Int]]

Karps

- An implementation of these principles on top of Spark (Haskell + Scala)
- It outputs a graph of logical plans for Spark (or other systems)
- Follows the compiler principle:
  - I will try to prove you wrong until you have a good chance of being right
- Karps makes a number of correctness checks for your program
- It acts as a global optimizer for your pipeline

Demo 1

databricks

This is useless!

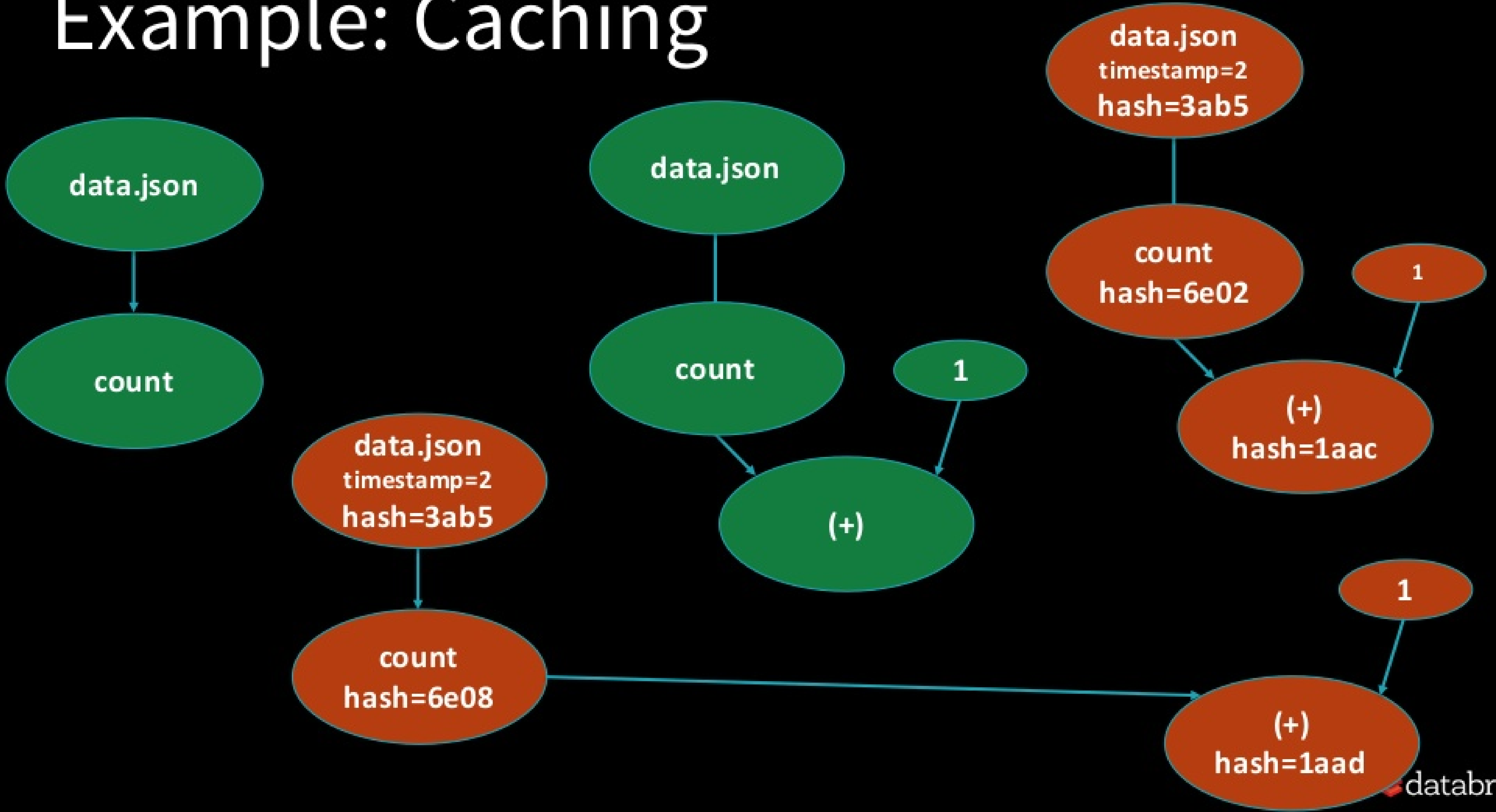
- Lazy construction of very complex programs
- Most operations in Spark can be translated to a small set of primitive actions with well-defined composition rules.
- The optimizer can then rewrite the program without changing the outcome
- Optimizations can leverage further SQL optimizations

Dealing with In and Out

- The only type of I/O: read and write datasets
- This is an observable
- Operations are deterministic + results are cached
  - -> only recompute when the data changes

- Demo

Example: Caching



databricks

Example: graph introspection

Future directions

- Python interface (interface + pandas backend)
- Writer module
- Finish GroupBy (cool stuff ahead)
- SQL (simple and cool stuff to do in this area)

Conclusion: trends in data processing

- How to manage the complexity of data flows?
- Taking inspiration from the functional world
- Spark provides solid foundation
- Laziness, declarative APIs alleviate complexity

Trying this demo

- https://github.com/krapsh/kraps-haskell
- Notebooks + Docker image:
  - https://github.com/krapsh/kraps-haskell/tree/master/notebooks
- Server part (scala):
  - https://github.com/krapsh/kraps-server

Discount code: BayAreaMU

SPARK SUMMIT 2017

DATA SCIENCE AND ENGINEERING AT SCALE

JUNE 5 – 7 | MOSCONE CENTER | SAN FRANCISCO

ORGANIZED BY databricks

spark-summit.org/2017

Thank You