Apache Spark Tutorial

Reynold Xin @rxin

BOSS workshop at VLDB 2017

databricks

Apache Spark

- The most popular and de-facto framework for big data (science)
- APIs in SQL, R, Python, Scala, Java
- Support for SQL, ETL, machine learning/deep learning, graph …

- This tutorial (with hands-on components):
  - Brief Intro to Spark's DataFrame/Dataset API (and internals)
  - Deep Dive into Structured Streaming
  - Deep Learning for the Masses (with simple APIs and less data to train)

Who is this guy?

#1 committer on Spark project

Databricks Cofounder & Chief Architect

UC Berkeley AMPLab PhD (on leave since 2013)

Some Setup First

- https://community.cloud.databricks.com

- http://tinyurl.com/vldb2017

Abstractions in Spark 2.0+

- RDD
  - Old, basic abstraction (in NSDI paper)
- ML Pipelines
  - Self-evident
- DataFrame
  - Similar to relational table
  - Imperative-like programming model, but declarative
  - Supports both batch and streaming
- Dataset
  - DataFrame, with compile-time type safety

databricks

DataFrame

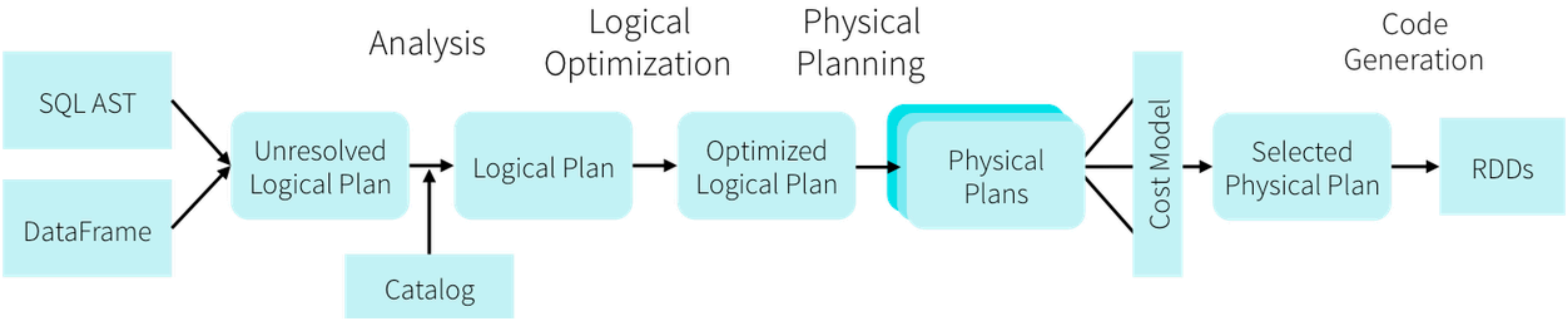
- Distributed collection of data grouped into named columns (i.e. RDD with schema)
- DSL designed for common tasks
  - Metadata
  - Sampling
  - Project, filter, aggregation, join, …
  - UDFs
- Available in Python, Scala, Java, and R (via SparkR)

DataFrame Internals (SIGMOD'15)

- Represented internally as a "logical plan"

- Execution is lazy, allowing it to be optimized by a query optimizer

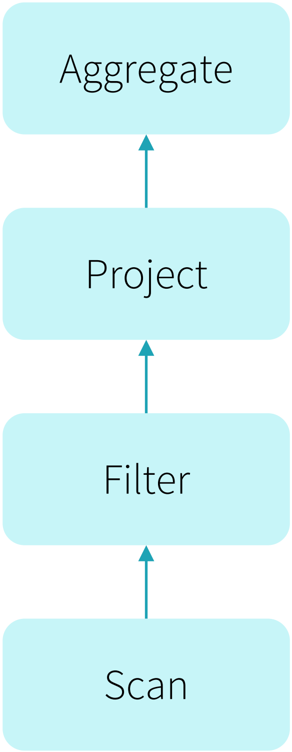
Plan Optimization & Execution



databricks

HyPer-inspired Whole Stage Code Generation

```
select count(*) from store_sales
where ss_item_sk = 1000
```



Volcano Iterator Model

Standard for 30 years: almost all databases do it

Each operator is an "iterator" that consumes records from its input operator

```
class Filter {
  def next(): Boolean = {
    var found = false
    while (!found && child.next()) {
      found = predicate(child.fetch())
    }
    return found
  }

  def fetch(): InternalRow = {
    child.fetch()
  }
  …
}
```

What if we hire a college freshman to implement this query in Java in 10 mins?

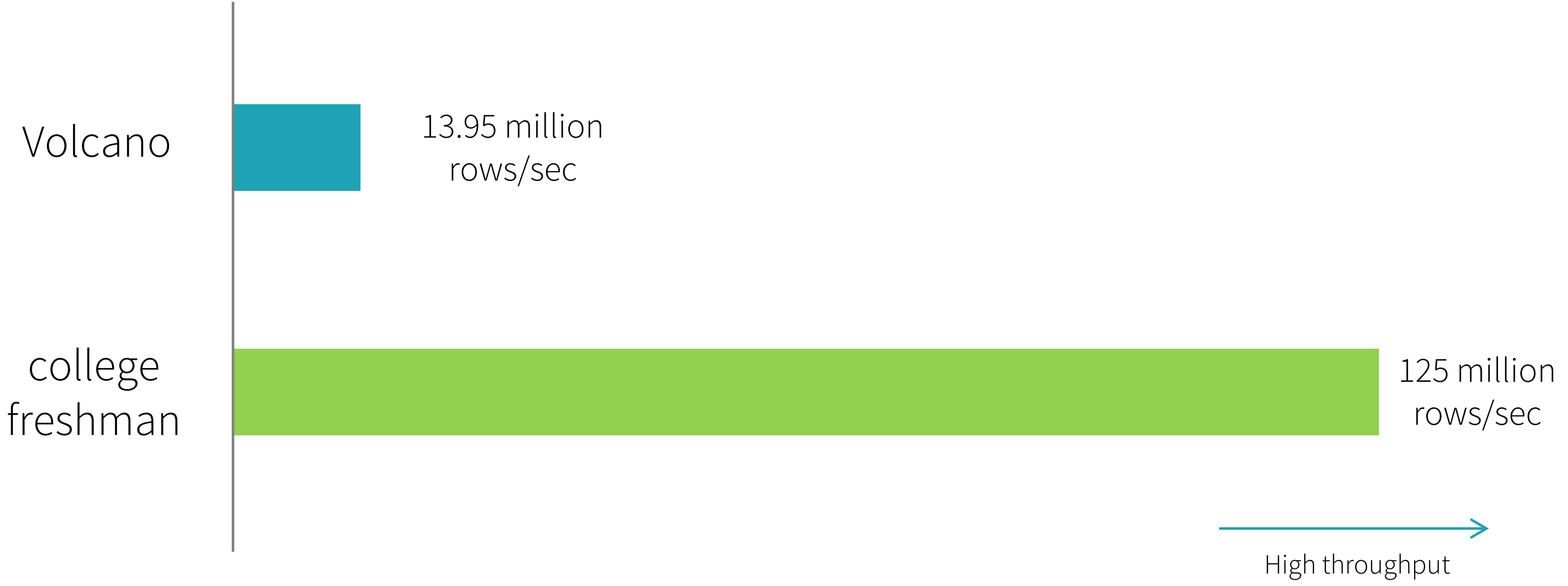
```
select count(*) from store_sales
where ss_item_sk = 1000
```

```
var count = 0
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

Volcano model
30+ years of database research

vs

college freshman
hand-written code in 10 mins



Note: End-to-end, single thread, single column, and data originated in Parquet on disk

How does a student beat 30 years of research?

Volcano

1. Many virtual function calls
2. Data in memory (or cache)
3. No loop unrolling, SIMD, pipelining

hand-written code

1. No virtual function calls
2. Data in CPU registers
3. Compiler loop unrolling, SIMD, pipelining

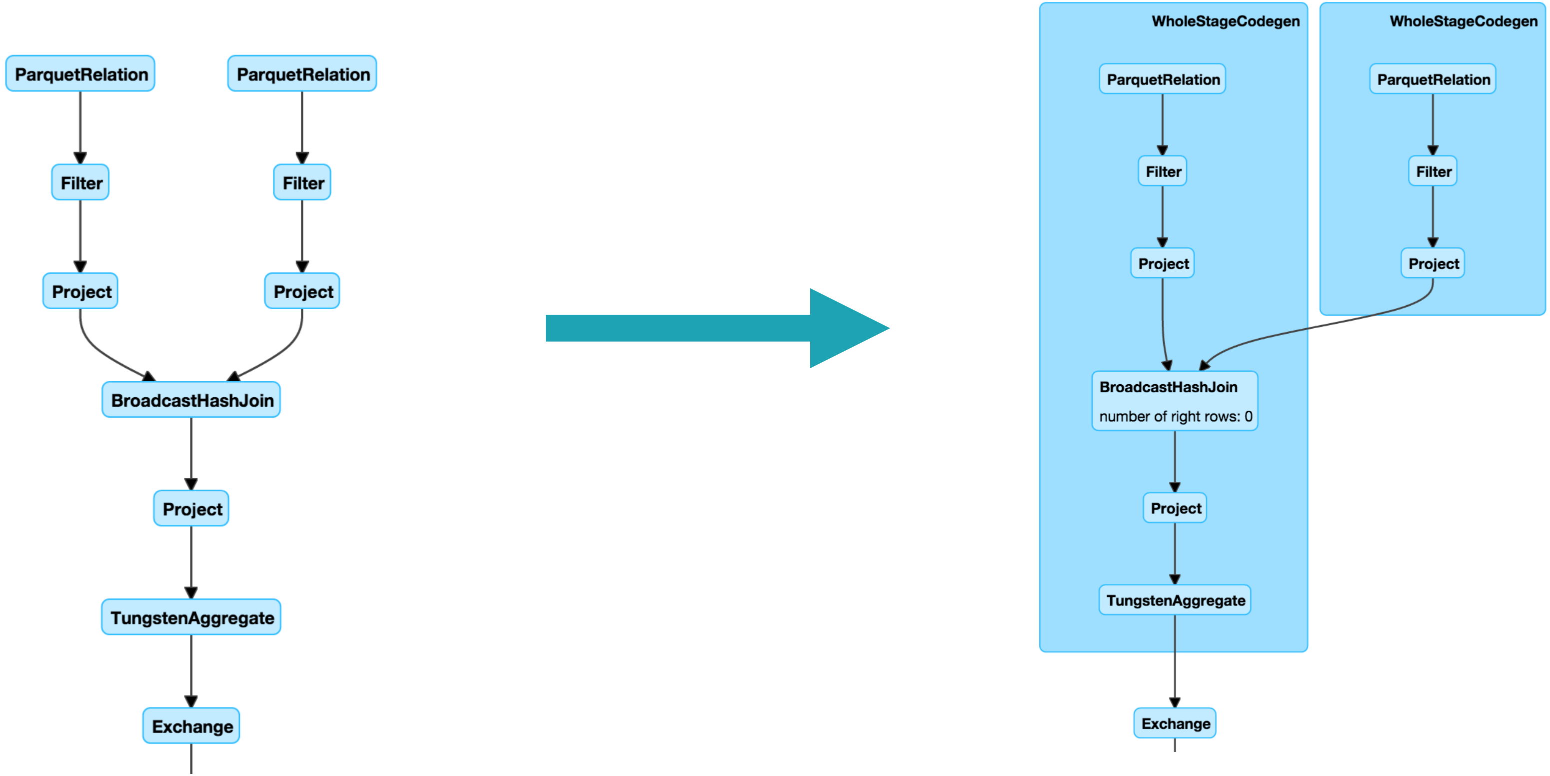
Take advantage of all the information that is known **after** query compilation

Whole-stage Codegen

Fusing operators together so the generated code looks like hand optimized code:

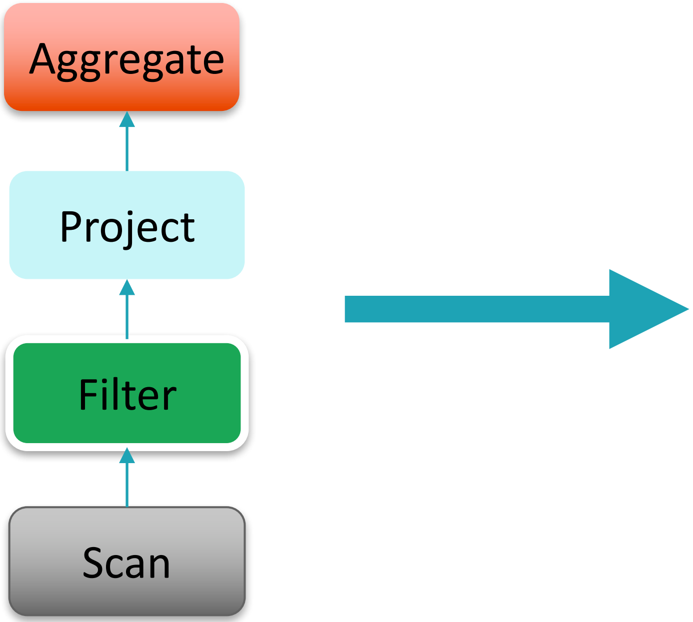
- Identity chains of operators ("stages")
- Compile each stage into a single function
- Functionality of a general purpose execution engine; performance as if hand built system just to run your query

Whole-stage Codegen: Planner



databricks

Whole-stage Codegen: Spark as a “Compiler”




```
long count = 0;
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1;
  }
}
```

Exercise

http://tinyurl.com/vldb2017

Easy, Scalable, Fault-tolerant Stream Processing with **Structured Streaming**

databricks

building robust
stream processing
apps is hard

Complexities in stream processing

COMPLEX DATA

Diverse data formats
(json, avro, binary, …)

Data can be dirty,
late, out-of-order

COMPLEX WORKLOADS

Combining streaming with interactive queries

Machine learning

COMPLEX SYSTEMS

Diverse storage systems
(Kafka, S3, Kinesis, RDBMS, …)

System failures

Structured Streaming

stream processing on Spark SQL engine
fast, scalable, fault-tolerant

rich, unified, high level APIs
deal with *complex data* and *complex workloads*

rich ecosystem of data sources
integrate with many *storage systems*

you
should not have to
reason about streaming

you

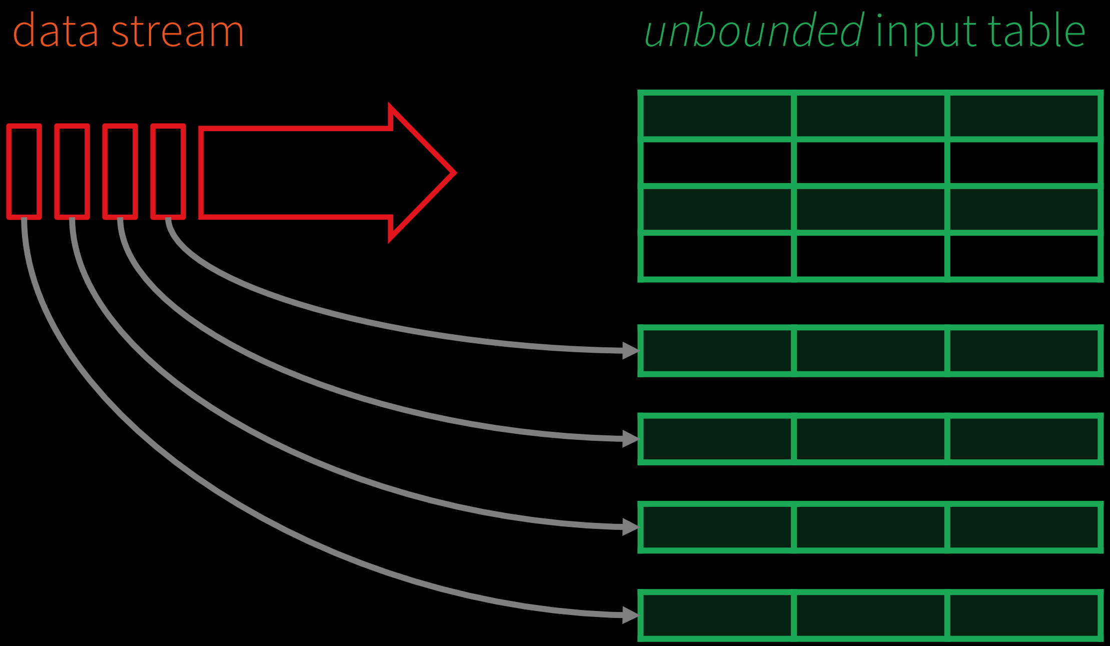
should write simple queries

&

Spark

should continuously update the answer

Treat Streams as Unbounded Tables



new data in the data stream

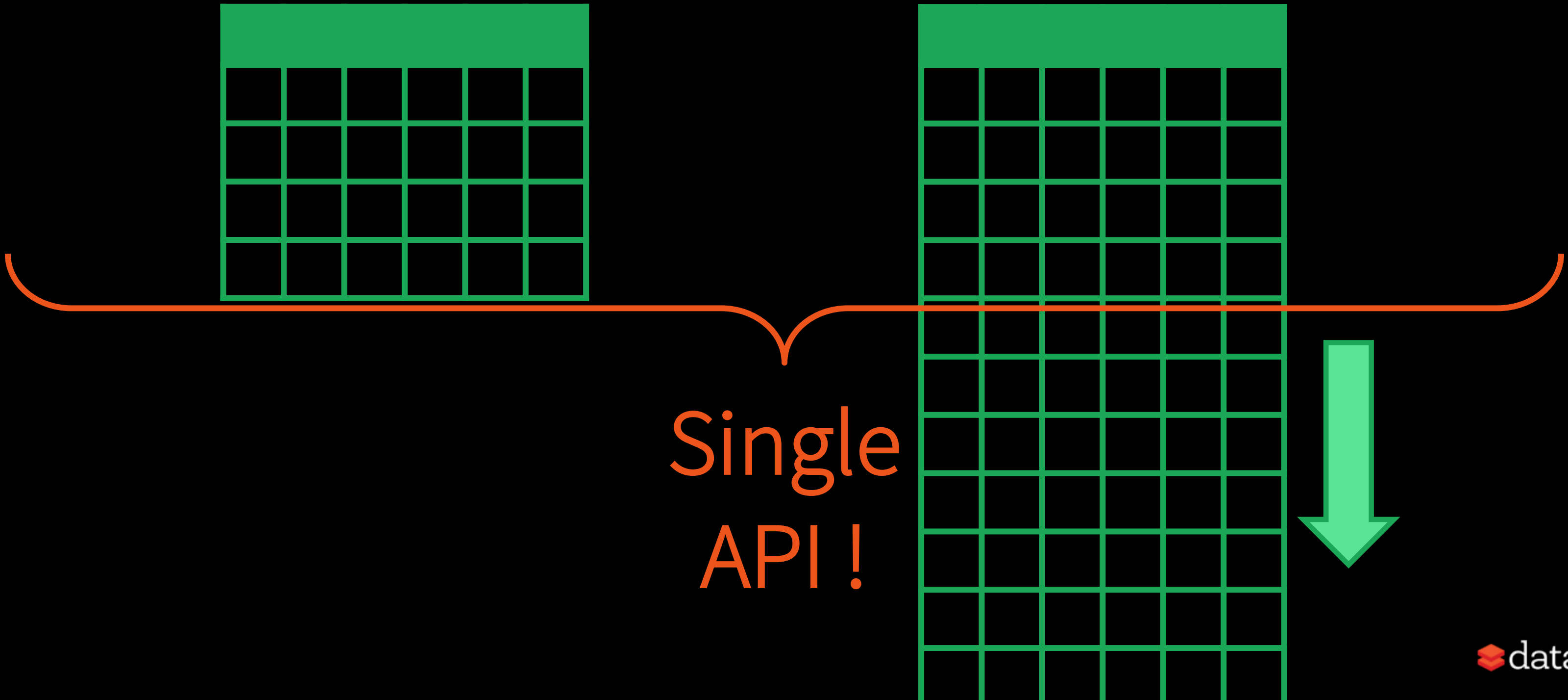
=

new rows appended to a unbounded table

Table ⇔ Dataset/DataFrame

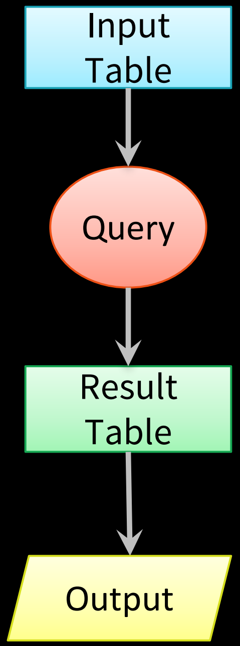
static data =
bounded table

streaming data =
unbounded table



databricks

Batch Queries with DataFrames




```
input = spark.read
      .format("json")
      .load("source-path")
```

Create input DF from Json file

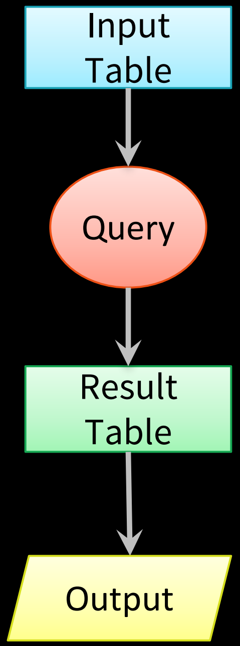
```
result = input
      .select("device", "signal")
      .where("signal > 15")
```

Create result DF by querying for some devices to create

```
result.write
      .format("parquet")
      .save("dest-path")
```

Output result to parquet file

Streaming Queries with DataFrames




```
input = spark.readStream
      .format("json")
      .load("source-path")
```

Create input DF from Kafa

Replace read with readStream

```
result = input
      .select("device", "signal")
      .where("signal > 15")
```

Select some devices

Query does not change

```
result.writeStream
      .format("parquet")
      .start("dest-path")
```

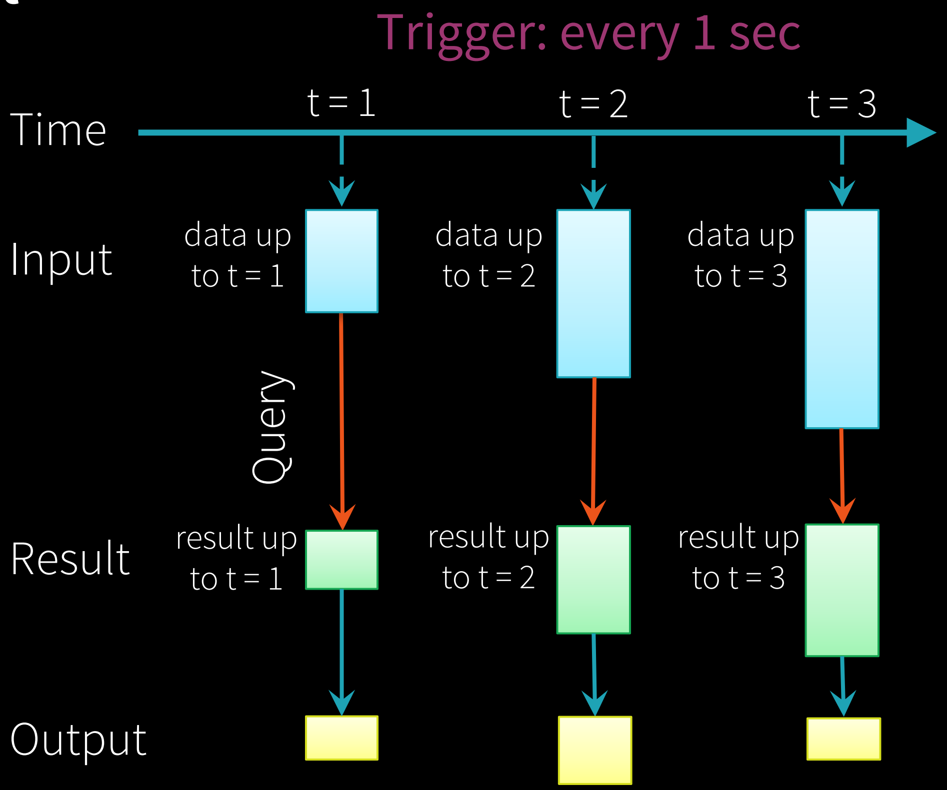
Write to Parquet file stream

Replace save() with start()

Conceptual Model

As the input table grows with new data, the result table changes

Every trigger interval, we can output the changes in the result

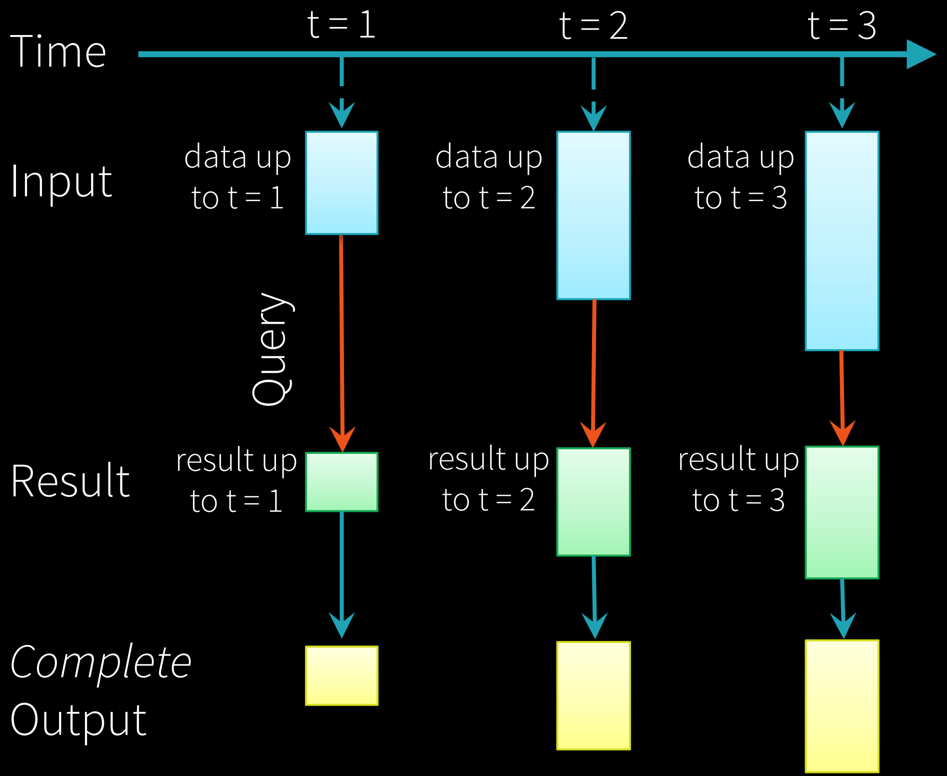


databricks

Conceptual Model

Output mode defines what changes to output

Complete mode outputs the entire result



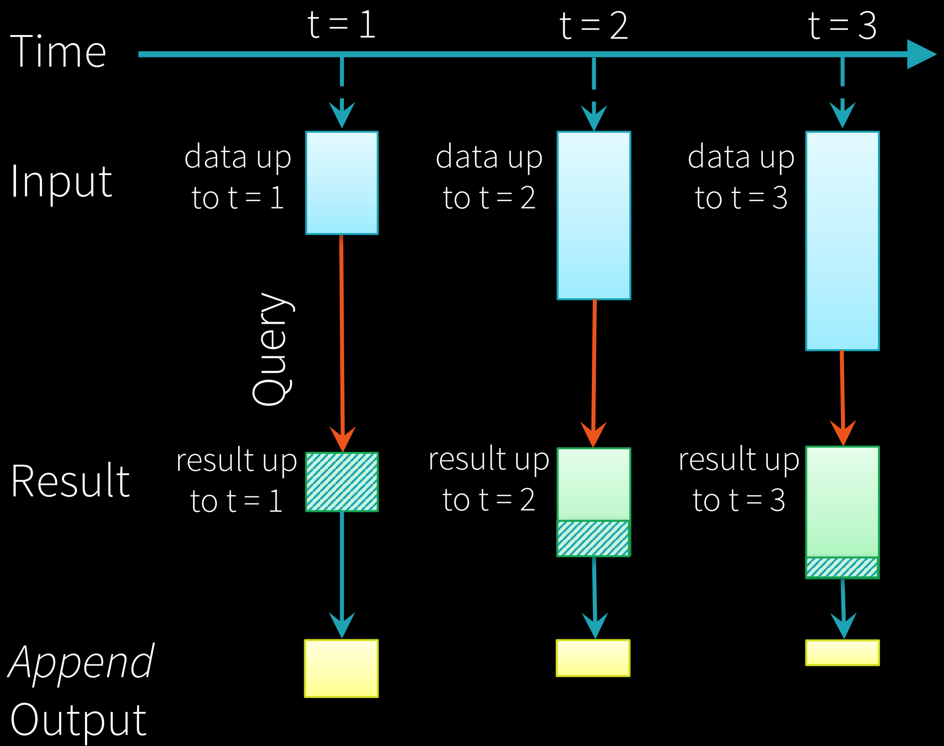
databricks

Conceptual Model

Output mode defines what changes to output

Append mode outputs new tuples only

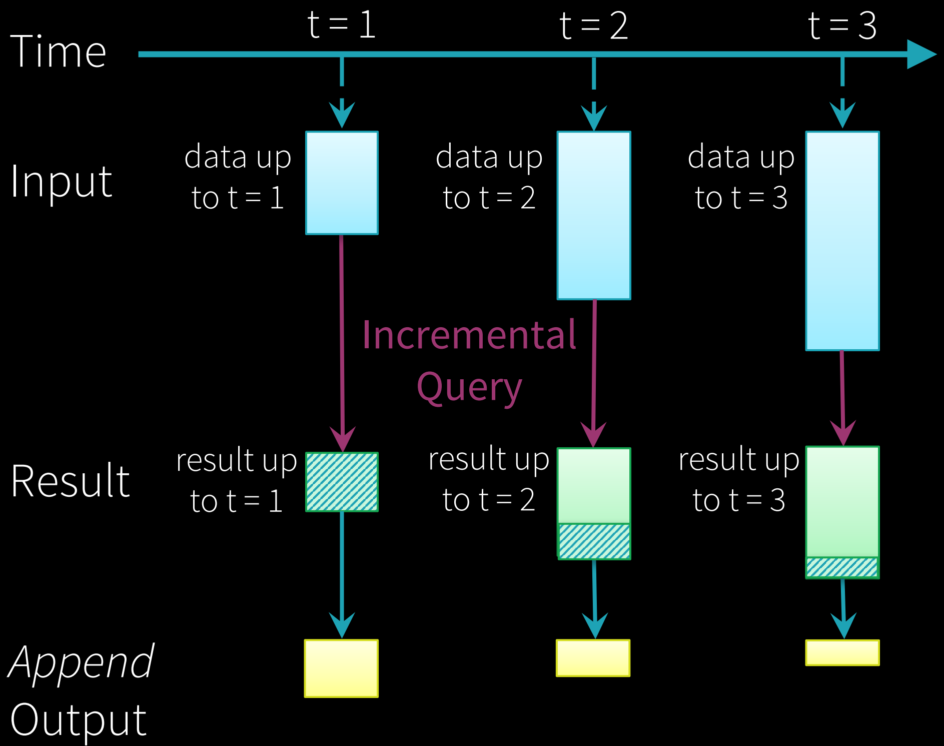
Update mode output tuples that have changed since the last trigger



Conceptual Model

Full input does not need to be processed every trigger

Engine converts query to an incremental query that operates only on new data to generate output



databricks

Anatomy of a Streaming Query

Streaming word count

Anatomy of a Streaming Query

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
```

Source

- Specify one or more locations to read data from
- Built in support for Files/Kafka/Socket, pluggable.
- Can include multiple sources of different types using `union()`

Anatomy of a Streaming Query

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy('value.cast("string") as 'key)
  .agg(count("*") as 'value)
```

Transformation

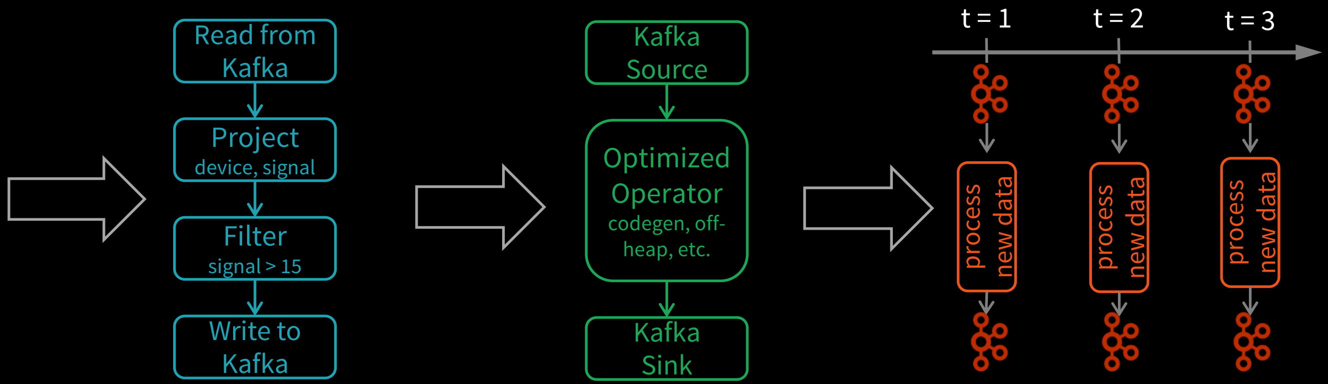
- Using DataFrames, Datasets and/or SQL.
- Catalyst figures out how to execute the transformation incrementally.
- Internal processing always exactly-once.

Spark automatically streamifies!

```
input = spark.readStream
   .format("kafka")
   .option("subscribe", "topic")
   .load()

result = input
   .select("device", "signal")
   .where("signal > 15")

result.writeStream
   .format("parquet")
   .start("dest-path")
```



DataFrames, Datasets, SQL

Logical Plan

Optimized Physical Plan

Series of Incremental Execution Plans

Spark SQL converts batch-like query to a series of incremental execution plans operating on new batches of data

Anatomy of a Streaming Query

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy('value.cast("string") as 'key)
  .agg(count("*") as 'value)
  .writeStream
  .format("kafka")
  .option("topic", "output")
```

Sink

- Accepts the output of each batch.
- When supported sinks are transactional and exactly once (Files).
- Use `foreach` to execute arbitrary code.

Anatomy of a Streaming Query

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy('value.cast("string") as 'key)
  .agg(count("*") as 'value)
  .writeStream
  .format("kafka")
  .option("topic", "output")
  .trigger("1 minute")
  .outputMode("update")
```

Output mode – What's output

- Complete – Output the whole answer every time
- Update – Output changed rows
- Append – Output new rows only

Trigger – When to output

- Specified as a time, eventually supports data size
- No trigger means as fast as possible

Anatomy of a Streaming Query

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy('value.cast("string") as 'key)
  .agg(count("*") as 'value)
  .writeStream
  .format("kafka")
  .option("topic", "output")
  .trigger("1 minute")
  .outputMode("update")
  .option("checkpointLocation", "…")
  .start()
```

Checkpoint

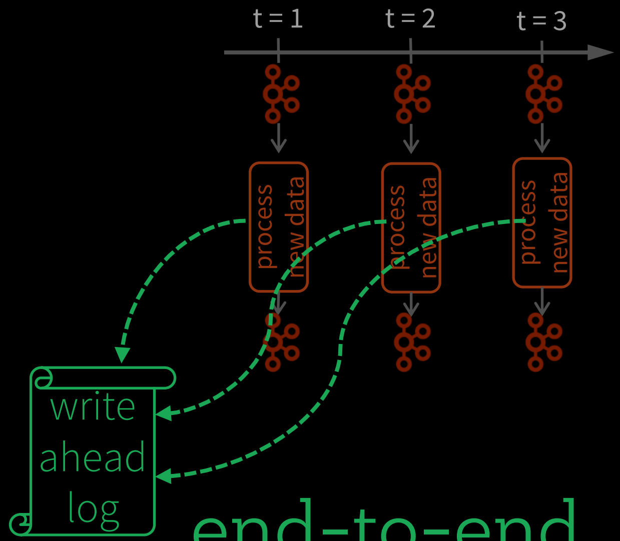
- Tracks the progress of a query in persistent storage
- Can be used to restart the query if there is a failure.

databricks

Fault-tolerance with Checkpointing

Checkpointing – tracks progress (offsets) of consuming data from the source and intermediate state.

- Offsets and metadata saved as JSON
- Can resume after changing your streaming transformations



end-to-end exactly-once guarantees

Complex Streaming ETL

Traditional ETL

Raw, dirty, un/semi-structured is data dumped as files

Periodic jobs run every few hours to convert raw data to structured data ready for further analytics

Traditional ETL

Hours of delay before taking decisions on latest data

Unacceptable when time is of essence
[intrusion detection, anomaly detection, etc.]

Streaming ETL w/ Structured Streaming

Structured Streaming enables raw data to be available as structured data as soon as possible

Streaming ETL w/ Structured Streaming

Example

Json data being received in Kafka

Parse nested json and flatten it

Store in structured Parquet table

Get end-to-end failure guarantees

```
val rawData = spark.readStream
  .format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()

val parsedData = rawData
  .selectExpr("cast (value as string) as json"))
  .select(from_json("json", schema).as("data"))
  .select("data.*")

val query = parsedData.writeStream
  .option("checkpointLocation", "/checkpoint")
  .partitionBy("date")
  .format("parquet")
  .start("/parquetTable")
```

databricks

Reading from Kafka

Specify options to configure

```
val rawData = spark.readStream
      .format("kafka")
      .option("kafka.boostrap.servers",...)
      .option("subscribe", "topic")
      .load()
```

How?

```
kafka.boostrap.servers => broker1,broker2
```

What?

```
subscribe        =>  topic1,topic2,topic3   // fixed list of topics
subscribePattern =>  topic*                 // dynamic list of topics
assign           =>  {"topicA":[0,1] }      // specific partitions
```

Where?

```
startingOffsets => latest(default) / earliest / {"topicA":{"0":23,"1":345} }
```

databricks

Reading from Kafka

```
val rawData = spark.readStream
    .format("kafka")
    .option("kafka.boostrap.servers",...)
    .option("subscribe", "topic")
    .load()
```

rawData dataframe has the following columns

key	value	topic	partition	offset	timestamp
[binary]	*[binary]*	"topicA"	0	345	1486087873
[binary]	*[binary]*	"topicB"	3	2890	1486086721

Transforming Data

Cast binary *value* to string
Name it column *json*

```
val parsedData = rawData
    .selectExpr("cast (value as string) as json")
    .select(from_json("json", schema).as("data"))
    .select("data.*")
```

databricks

Transforming Data

Cast binary *value* to string
Name it column *json*

```
val parsedData = rawData
    .selectExpr("cast (value as string) as json")
    .select(from_json("json", schema).as("data"))
    .select("data.*")
```

Parse *json* string and expand into nested columns, name it *data*

json
{ "timestamp": 1486087873, "device": "devA", …}
{ "timestamp": 1486082418, "device": "devX", …}

from_json("json") as "data"

data (nested)		
timestamp	device	…
1486087873	devA	…
1486086721	devX	…

Transforming Data

Cast binary *value* to string
Name it column *json*

```
val parsedData = rawData
    .selectExpr("cast (value as string) as json")
    .select(from_json("json", schema).as("data"))
    .select("data.*")
```

Parse *json* string and expand into nested columns, name it *data*

Flatten the nested columns

data (nested)		
timestamp	device	…
1486087873	devA	…
1486086721	devX	…

select("data.*")

(not nested)

timestamp	device	…
1486087873	devA	…
1486086721	devX	…

Transforming Data

Cast binary *value* to string
Name it column *json*

```
val parsedData = rawData
    .selectExpr("cast (value as string) as json")
    .select(from_json("json", schema).as("data"))
    .select("data.*")
```

Parse *json* string and expand into nested columns, name it data

Flatten the nested columns

powerful built-in APIs to perform complex data transformations

from_json, to_json, explode, ...
100s of functions

(see our blog post)

Writing to Parquet

Save parsed data as Parquet table in the given path

Partition files by date so that future queries on time slices of data is fast

- e.g. query on last 48 hours of data

```
val query = parsedData.writeStream
      .option("checkpointLocation", ...)
      .partitionBy("date")
      .format("parquet")
      .start("/parquetTable")
```

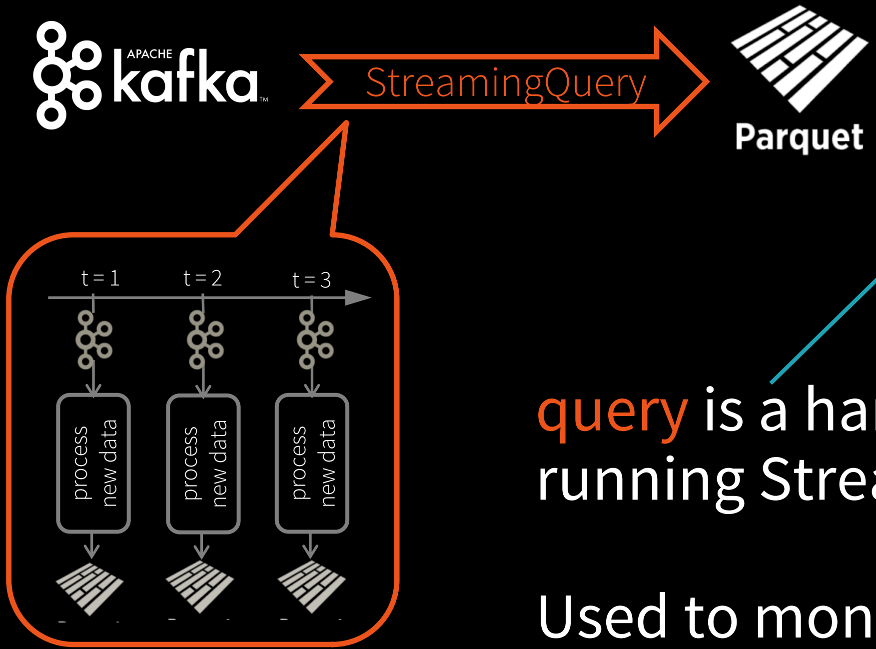
Checkpointing

Enable checkpointing by setting the checkpoint location to save offset logs

`start` actually starts a continuous running StreamingQuery in the Spark cluster

```
val query = parsedData.writeStream
      .option("checkpointLocation", ...)
      .format("parquet")
      .partitionBy("date")
      .start("/parquetTable/")
```

Streaming Query




```
val query = parsedData.writeStream
  .option("checkpointLocation", ...)
  .format("parquet")
  .partitionBy("date")
  .start("/parquetTable")
```

query is a handle to the continuously running StreamingQuery

Used to monitor and manage the execution

Data Consistency on Ad-hoc Queries

Data available for complex, ad-hoc analytics within seconds

Parquet table is updated atomically, ensures *prefix integrity*

Even if distributed, ad-hoc queries will see either all updates from streaming query or none, read more in our blog

https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html

Working With Time

Event Time

Many use cases require aggregate statistics by event time

- E.g. what's the #errors in each system in the 1 hour windows?

Many challenges

- Extracting event time from data, handling late, out-of-order data

DStream APIs were insufficient for event-time stuff

Event time Aggregations

Windowing is just another type of grouping in Struct. Streaming

number of records every hour

```
parsedData
      .groupBy(window("timestamp","1 hour"))
      .count()
```

avg signal strength of each device every 10 mins

```
parsedData
      .groupBy(
            "device",
            window("timestamp","10 mins"))
      .avg("signal")
```

Support UDAFs!

databricks

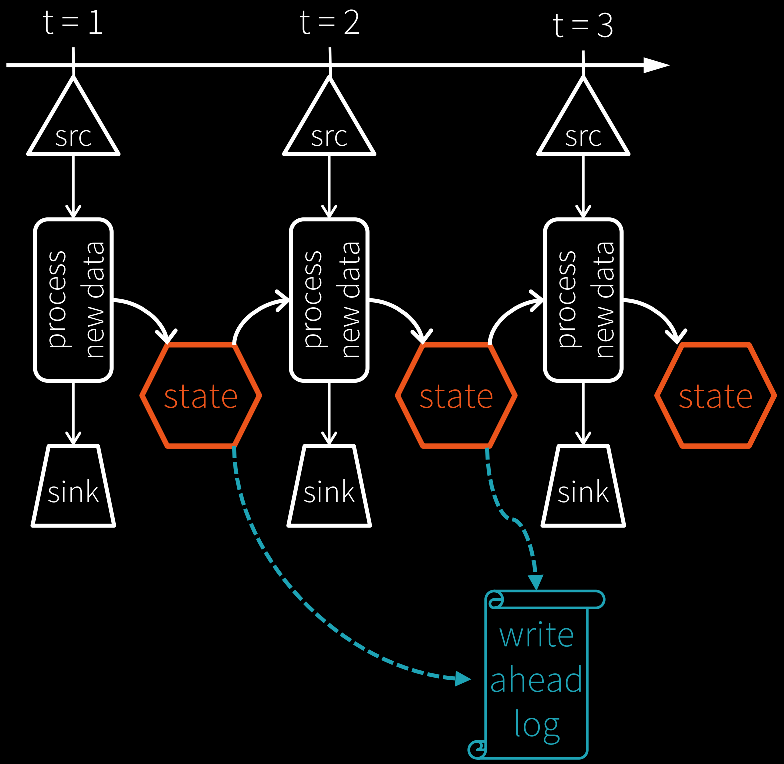
Stateful Processing for Aggregations

Aggregates has to be saved as distributed state between triggers

Each trigger reads previous state and writes updated state

State stored in memory, backed by *write ahead log* in HDFS/S3

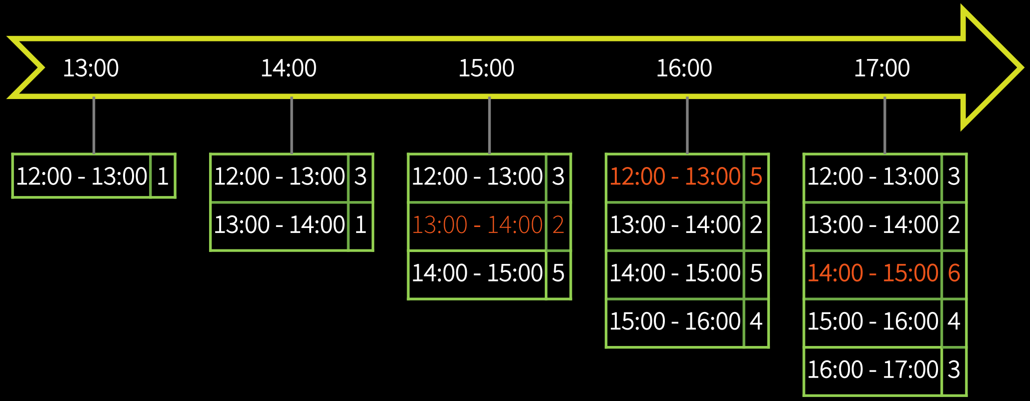
Fault-tolerant, exactly-once guarantee!



databricks

Automatically handles Late Data

Keeping state allows late data to update counts of old windows



But size of the state increases indefinitely if old windows are not dropped

red = state updated with late data

Watermarking

Watermark - moving threshold of how late data is expected to be and when to drop old state

Trails behind max seen event time

Trailing gap is configurable

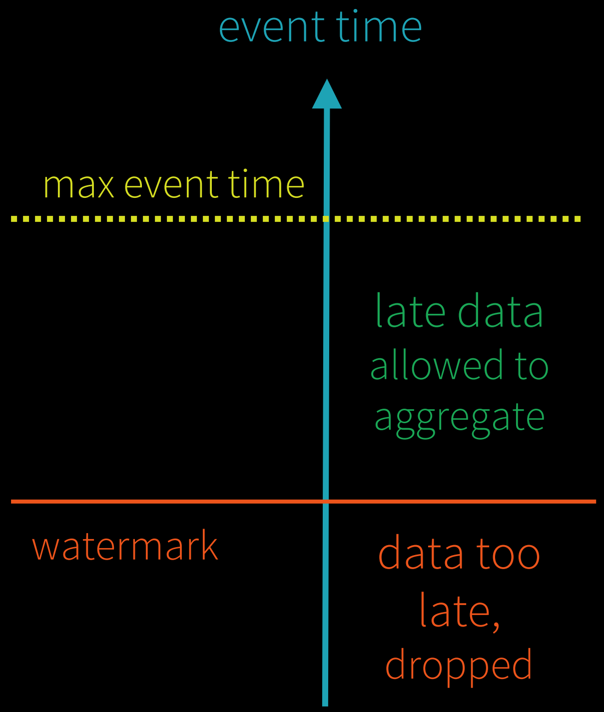
databricks

Watermarking

Data newer than watermark may be late, but allowed to aggregate

Data older than watermark is "too late" and dropped

Windows older than watermark automatically deleted to limit the amount of intermediate state



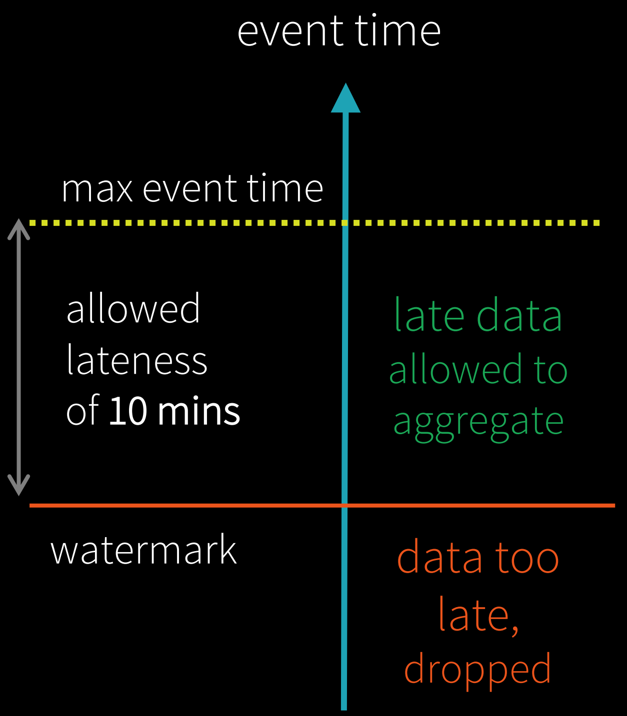
Watermarking

Useful only in stateful operations

(streaming aggs, dropDuplicates, mapGroupsWithState, ...)

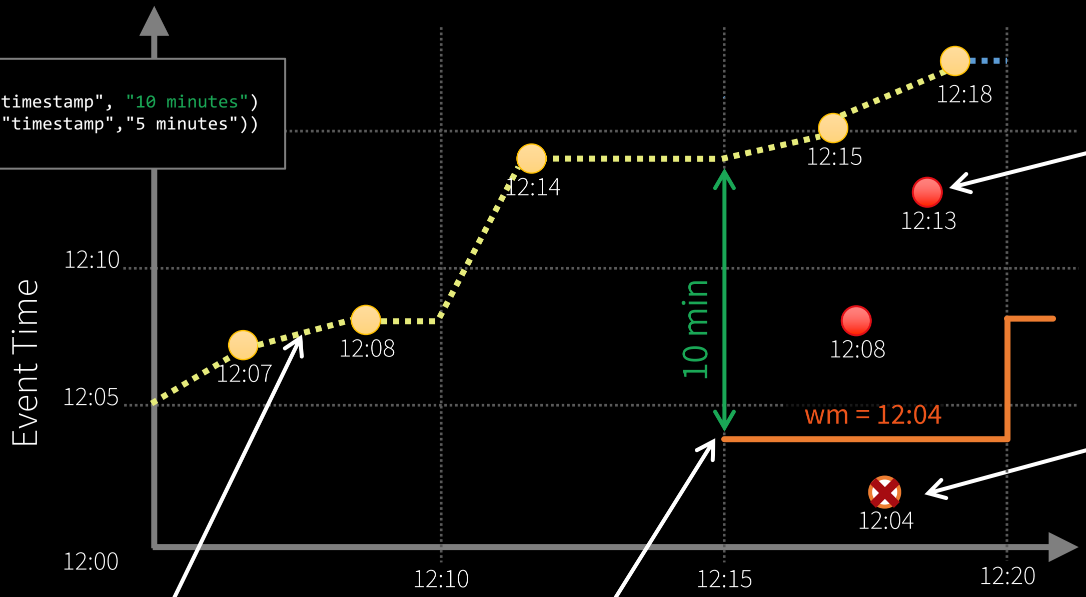
Ignored in non-stateful streaming queries and batch queries

```
parsedData
      .withWatermark("timestamp", "10 minutes")
      .groupBy(window("timestamp","5 minutes"))
      .count()
```



Watermarking

```
parsedData
    .withWatermark("timestamp", "10 minutes")
    .groupBy(window("timestamp","5 minutes"))
    .count()
```



data is late, but considered in counts

data too late, ignored in counts, state dropped

system tracks max observed event time

watermark updated to
12:14 - 10m = 12:04
for next trigger,
state < 12:04 deleted

More details in my [blog post]()

databricks

Clean separation of concerns

Query Semantics

separated from

Processing Details

```
parsedData
   .withWatermark("timestamp", "10 minutes")
   .groupBy(window("timestamp","5 minutes"))
   .count()
   .writeStream
   .trigger("10 seconds")
   .start()
```

databricks

Clean separation of concerns

Query Semantics

How to group data by time?
(same for batch & streaming)

Processing Details

```
parsedData
  .withWatermark("timestamp", "10 minutes")
  .groupBy(window("timestamp","5 minutes"))
  .count()
  .writeStream
  .trigger("10 seconds")
  .start()
```

Clean separation of concerns

<u>Query Semantics</u>

How to group data by time?
(same for batch & streaming)

<u>Processing Details</u>

How late can data be?

```
parsedData
   .withWatermark("timestamp", "10 minutes")
   .groupBy(window("timestamp","5 minutes"))
   .count()
   .writeStream
   .trigger("10 seconds")
   .start()
```

Clean separation of concerns

Query Semantics

How to group data by time?
(same for batch & streaming)

Processing Details

How late can data be?

How often to emit updates?

```
parsedData
   .withWatermark("timestamp", "10 minutes")
   .groupBy(window("timestamp","5 minutes"))
   .count()
   .writeStream
   .trigger("10 seconds")
   .start()
```

Arbitrary Stateful Operations [Spark 2.2]

mapGroupsWithState allows any user-defined stateful function to a user-defined state

Direct support for per-key timeouts in event-time or processing-time

Supports Scala and Java

```
ds.groupByKey(_.id)
  .mapGroupsWithState
    (timeoutConf)
    (mappingWithStateFunc)


def mappingWithStateFunc(
     key: K,
     values: Iterator[V],
     state: GroupState[S]): U = {
       // update or remove state
       // set timeouts
       // return mapped value
}
```

Other interesting operations

Streaming Deduplication

Watermarks to limit state

```
parsedData.dropDuplicates("eventId")
```

Stream-batch Joins

```
val batchData = spark.read
     .format("parquet")
     .load("/additional-data")
parsedData.join(batchData, "device")
```

Stream-stream Joins

Can use mapGroupsWithState

Direct support oming soon!

Building Complex Continuous Apps

Metric Processing @ databricks

Events generated by user actions (logins, clicks, spark job updates)

ETL	Clean, normalize and store historical data
Dashboards	Analyze trends in usage as they occur
Alerts	Notify engineers of critical issues
Ad-hoc Analysis	Diagnose issues when they occur

Metric Processing @ databricks

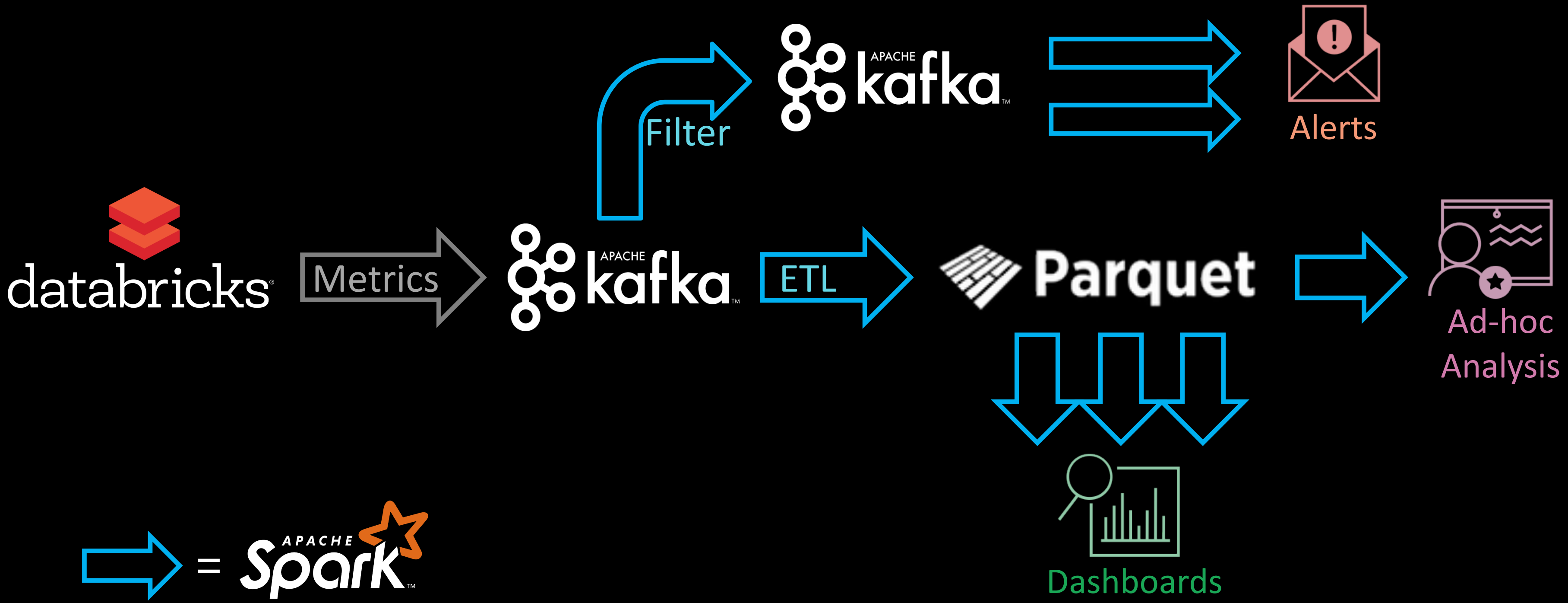
Difficult with only streaming frameworks

- ETL
- Dashboards
- Alerts
- Ad-hoc Analysis

kafka

- Limited retention in streaming storage
- Inefficient for ad-hoc queries
- Hard for novice users (limited or no SQL support)

Metric Processing @ databricks



Read from Apache Kafka


```
rawLogs = spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", ...)
    .option("subscribe", "rawLogs")
    .load()

augmentedLogs = rawLogs
    .withColumn("msg",
       from_json($"value".cast("string"),
       schema))
    .select("timestamp", "msg.*")
    .join(table("customers"), ["customer_id"])
```

DataFrames can be reused for multiple streams

Can build libraries of useful DataFrames and share code between applications

Write to Parquet

APACHE kafka → ETL → Parquet

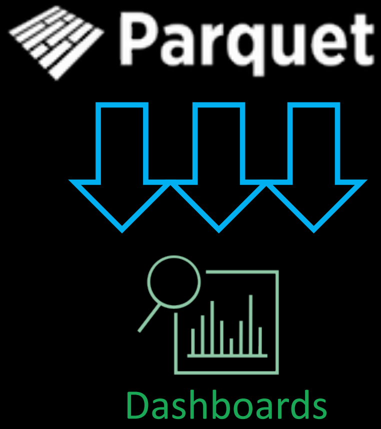
Store augmented stream as efficient columnar data for later processing

Latency: ~1 minute

```
augmented
  .repartition(1)
  .writeStream
  .format("parquet")
  .option("path", "/data/metrics")
  .trigger("1 minute")
  .start()
```

Buffer data and write one large file every minute for efficient reads

Dashboards



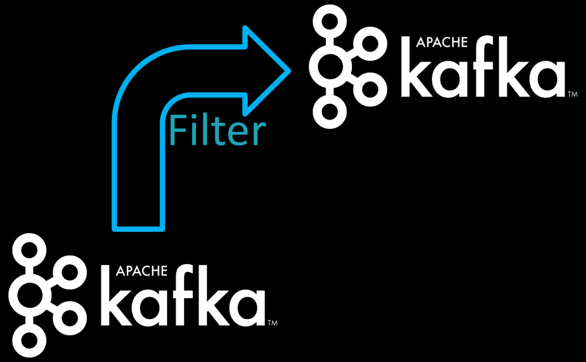
Always up-to-date visualizations of important business trends

Latency: ~1 minute to hours (configurable)

```
logins = spark.readStream.parquet("/data/metrics")
  .where("metric = 'login'")
  .groupBy(window("timestamp", "1 minute"))
  .count()

display(logins)      // Visualize in Databricks notebooks
```

Filter and write to APACHE kafka



Forward filtered and augmented events back to Kafka

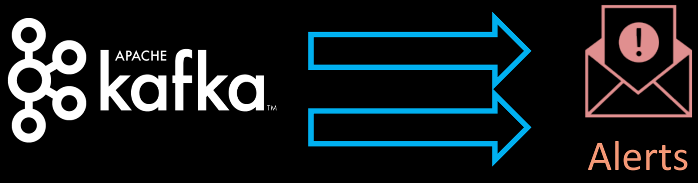
Latency: ~100ms average

```
filteredLogs = augmentedLogs
  .where("eventType = 'clusterHeartbeat'")
  .selectExpr("to_json(struct("*")) as value")

filteredLogs.writeStream
  .format("kafka")
  .option("kafka.bootstrap.servers", ...)
  .option("topic", "clusterHeartbeats")
  .start()
```

to_json() to convert columns back into json string, and then save as different Kafka topic

Simple Alerts



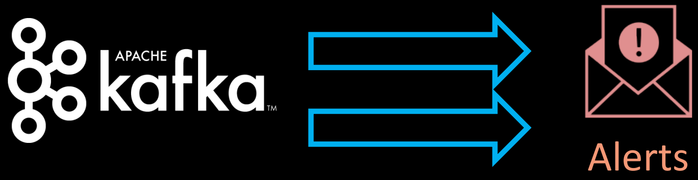
E.g. Alert when Spark cluster load > threshold

Latency: ~100 ms

```
sparkErrors
  .as[ClusterHeartBeat]
  .filter(_.load > 99)
  .writeStream
  .foreach(new PagerdutySink(credentials))
```

Notify PagerDuty

Complex Alerts



E.g. Monitor health of Spark clusters using custom stateful logic

Latency: ~10 seconds

React if no heartbeat from cluster for 1 min

```
sparkErrors
  .as[ClusterHeartBeat]
  .groupBy(_.id)
  .flatMapGroupsWithState(Update, ProcessingTimeTimeout("1 minute")) {
    (id: Int, events: Iterator[ClusterHeartBeat], state: GroupState[ClusterState]) =>
    ... // check if cluster non-responsive for a while
  }
```

Ad-hoc Analysis

Trouble shoot problems as they occur with latest information

Latency: ~1 minute

```
SELECT *
FROM parquet.`/data/metrics`
WHERE level IN ('WARN', 'ERROR')
  AND customer = "…"
  AND timestamp < now() - INTERVAL 1 HOUR
```

will read latest data when query executed

Metric Processing @ databricks

14+ billion records / hour
with 10 nodes

meet diverse latency requirements
as efficiently as possible

More Info

Structured Streaming Programming Guide

http://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Databricks blog posts for more focused discussions

https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html

https://databricks.com/blog/2017/01/19/real-time-streaming-etl-structured-streaming-apache-spark-2-1.html

https://databricks.com/blog/2017/02/23/working-complex-data-formats-structured-streaming-apache-spark-2-1.html

https://databricks.com/blog/2017/04/26/processing-data-in-apache-kafka-with-structured-streaming-in-apache-spark-2-2.html

https://databricks.com/blog/2017/05/08/event-time-aggregation-watermarking-apache-sparks-structured-streaming.html

and more to come, stay tuned!!

Deep Learning

- https://databricks.com/blog/2017/06/06/databricks-vision-simplify-large-scale-deep-learning.html

- rxin@databricks.com