A Deep Dive into Structured Streaming

Tathagata "*TD*" Das

@tathadas

Spark Summit 2016

databricks™

Who am I?

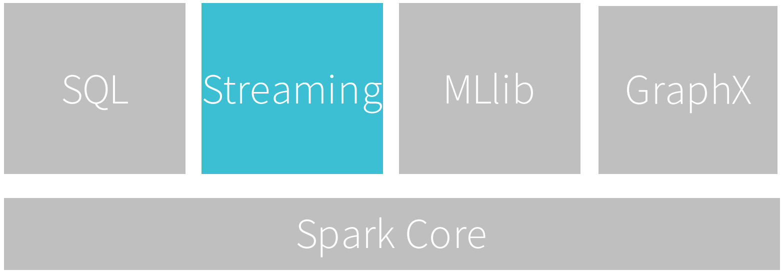
Project Mgmt. Committee (PMC) member of Apache Spark

Started Spark Streaming in grad school - AMPLab, UC Berkeley

Software engineer at **Databricks** and involved with all things streaming in Spark

Streaming in Apache Spark

Spark Streaming changed how people write streaming apps



Functional, concise and expressive

Fault-tolerant state management

Unified stack with batch processing

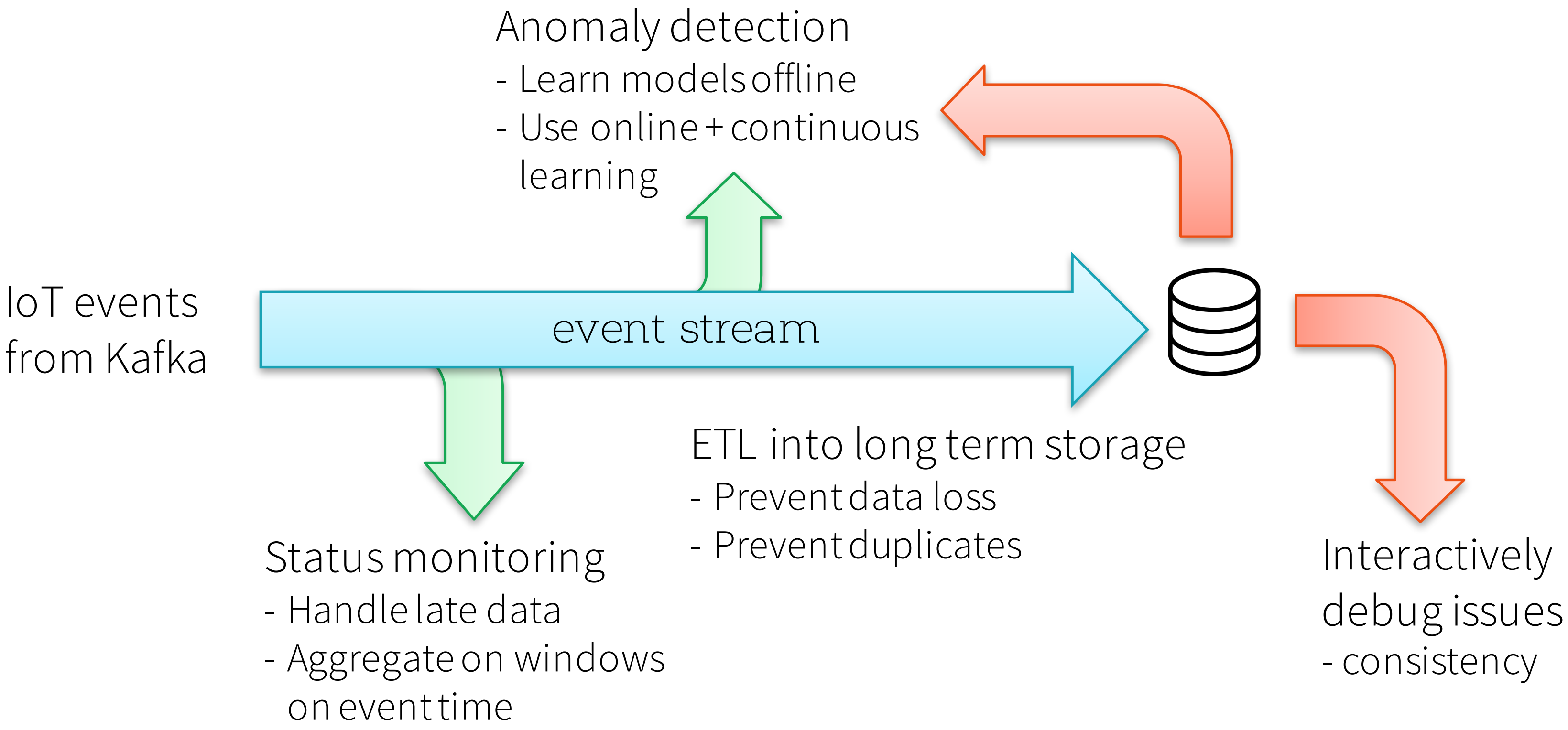
More than 50% users consider most important part of Apache Spark

Streaming apps are growing more complex

Streaming computations don't run in isolation

Need to interact with batch data,
interactive analysis, machine learning, etc.

Use case: IoT Device Monitoring



Continuous Applications

Not just *streaming* any more

Use case: IoT Device Monitoring

Anomaly detection
- Learn models offline
- Use online + continuous learning

IoT events from Kafka

event stream

ETL into long term storage
- Prevent data loss
- Prevent duplicates

Status monitoring
- Handle late data
- Aggregate on windows on event time

Interactively debug issues
- consistency

databricks

Pain points with DStreams

1. Processing with event-time, dealing with late data
   - DStream API exposes batch time, hard to incorporate event-time

2. Interoperate streaming with batch AND interactive
   - RDD/DStream has *similar* API, but still requires translation

3. Reasoning about end-to-end guarantees
   - Requires carefully constructing sinks that handle failures correctly
   - Data consistency in the storage while being updated

Structured Streaming

databricks™

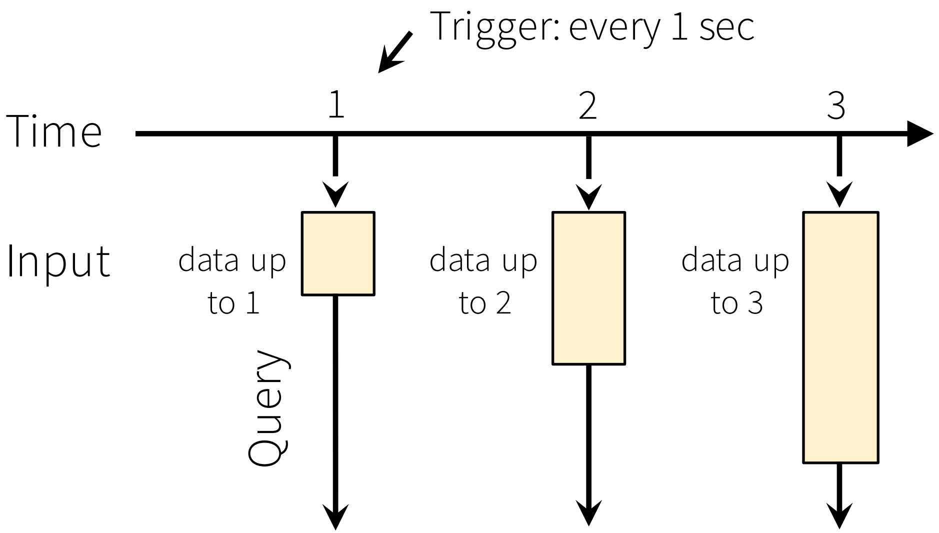
The simplest way to perform streaming analytics is not having to ***reason*** about streaming at all

New Model

Input: data from source as an append-only table

Trigger: how frequently to check input for new data

Query: operations on input
usual map/filter/reduce
new window, session ops

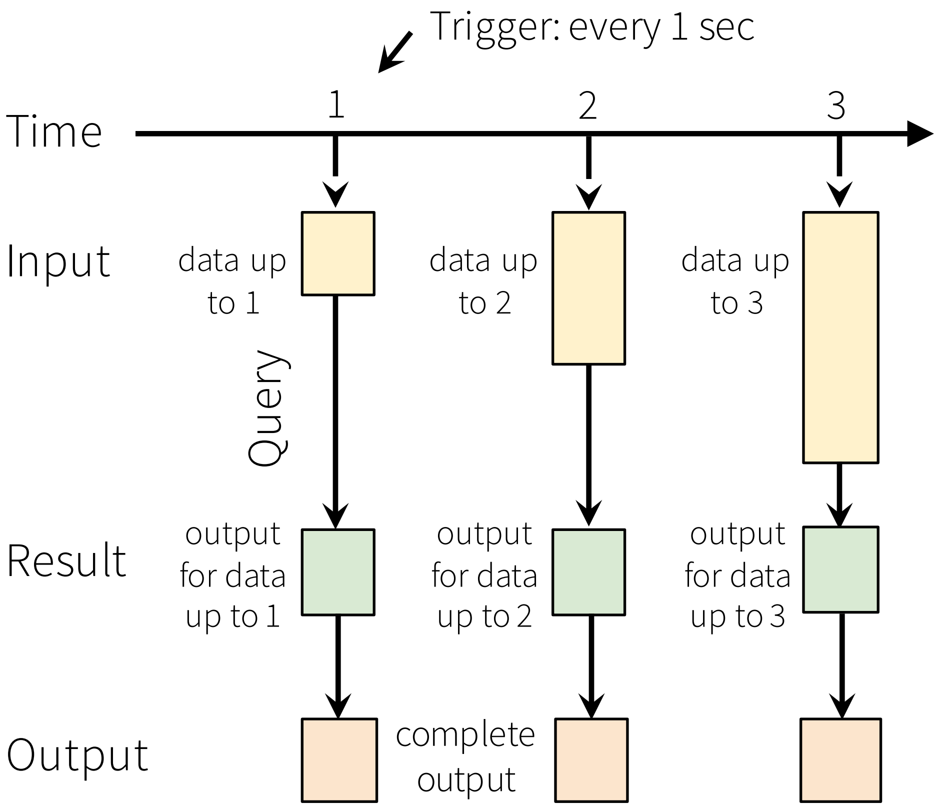


New Model

Result: final operated table
updated every trigger interval

Output: what part of result to write
to data sink after every trigger

Complete output: Write full result table every time



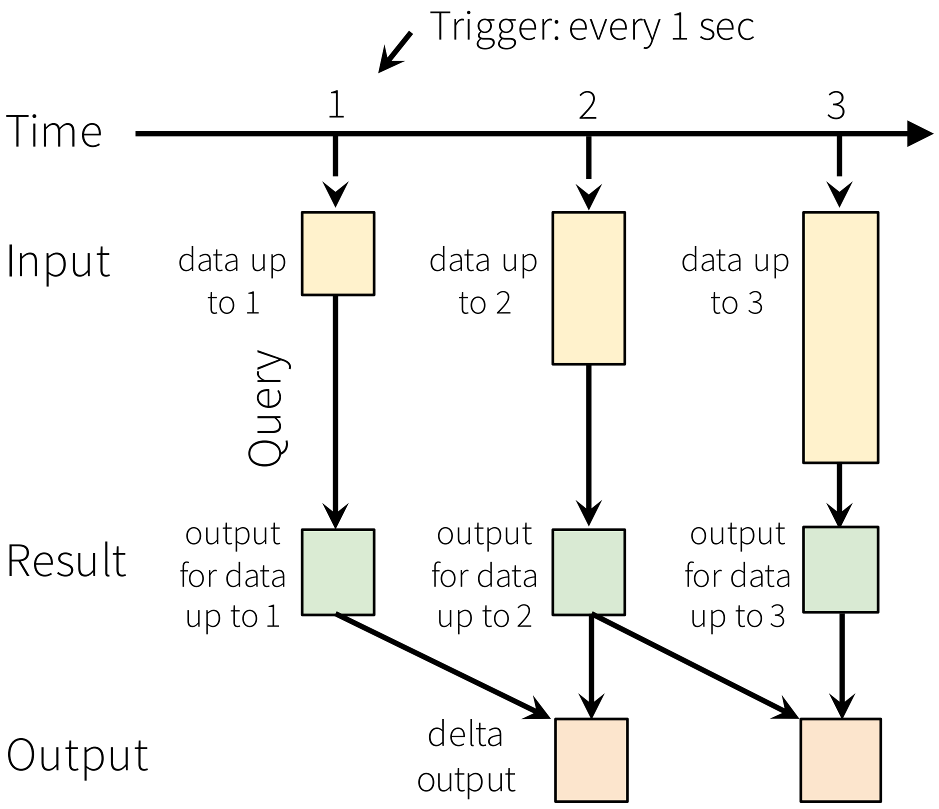
New Model

Result: final operated table updated every trigger interval

Output: what part of result to write to data sink after every trigger

Complete output: Write full result table every time
Delta output: Write only the rows that changed in result from previous batch
Append output: Write only new rows

*Not all output modes are feasible with all queries

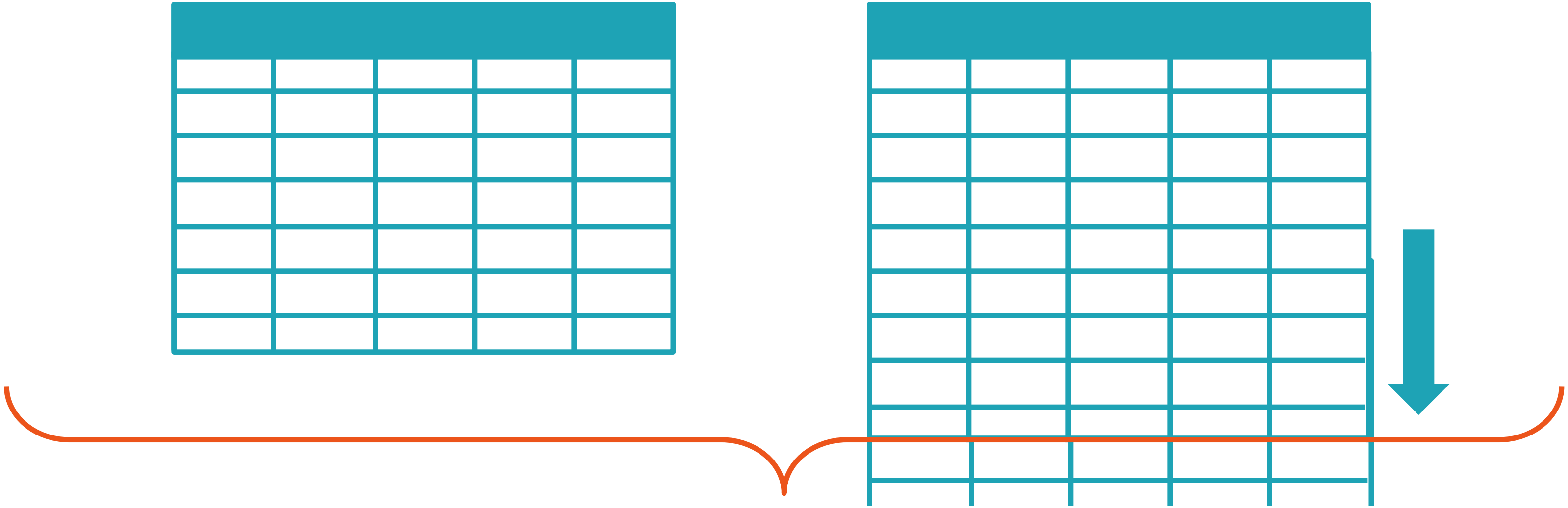


databricks

API - *Dataset/DataFrame*

Static, bounded data

Streaming, unbounded data



Single API !

databricks

Batch ETL with DataFrames

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

Read from Json file

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

Select some devices

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

Write to parquet file

Streaming ETL with DataFrames

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

Read from Json file stream

Replace `load()` with `stream()`

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

Select some devices

Code does not change

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

Write to Parquet file stream

Replace `save()` with `startStream()`

Streaming ETL with DataFrames

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

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

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

read…stream() creates a streaming DataFrame, does not start any of the computation

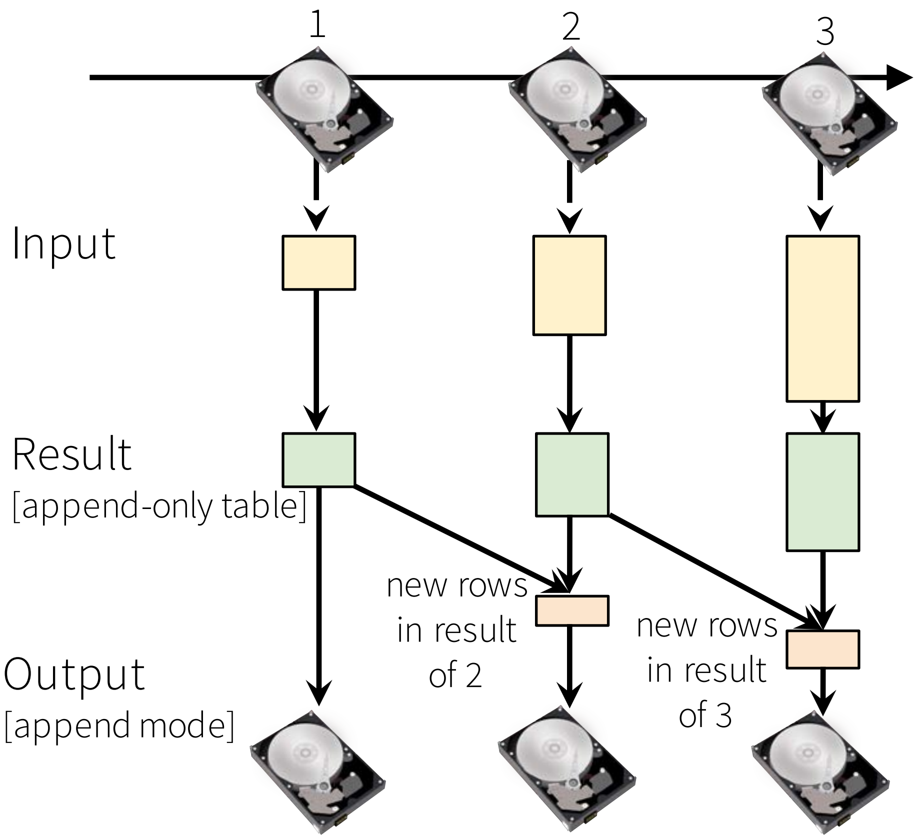
write…startStream() defines where & how to output the data and starts the processing

Streaming ETL with DataFrames

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

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

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



databricks

Continuous Aggregations

```
input.avg("signal")
```

Continuously compute *average* signal *across all devices*

```
input.groupBy("device-type")
     .avg("signal")
```

Continuously compute *average* signal of *each type of device*

Continuous Windowed Aggregations

```
input.groupBy(
        $"device-type",
        window($"event-time-col", "10 min"))
     .avg("signal")
```

Continuously compute *average* signal of *each type of device* in last 10 minutes using *event-time*

Simplifies event-time stream processing (not possible in DStreams)
Works on both, streaming and batch jobs

Joining streams with static data

```
kafkaDataset = spark.read
 .kafka("iot-updates")
 .stream()

staticDataset = ctxt.read
 .jdbc("jdbc://", "iot-device-info")

joinedDataset =
 kafkaDataset.join(
   staticDataset, "device-type")
```

Join streaming data from Kafka with static data via JDBC to enrich the streaming data …

… without having to think that you are joining streaming data

Output Modes

Defines what is outputted every time there is a trigger
Different output modes make sense for different queries

Append mode with
non-aggregation queries

```
input.select("device", "signal")
     .write
     .outputMode("append")
     .format("parquet")
     .startStream("dest-path")
```

Complete mode with
aggregation queries

```
input.agg(count("*"))
     .write
     .outputMode("complete")
     .format("parquet")
     .startStream("dest-path")
```

Query Management

```
query = result.write
      .format("parquet")
      .outputMode("append")
      .startStream("dest-path")


query.stop()
query.awaitTermination()
query.exception()

query.sourceStatuses()
query.sinkStatus()
```

query: a handle to the running streaming computation for managing it

- Stop it, wait for it to terminate
- Get status
- Get error, if terminated

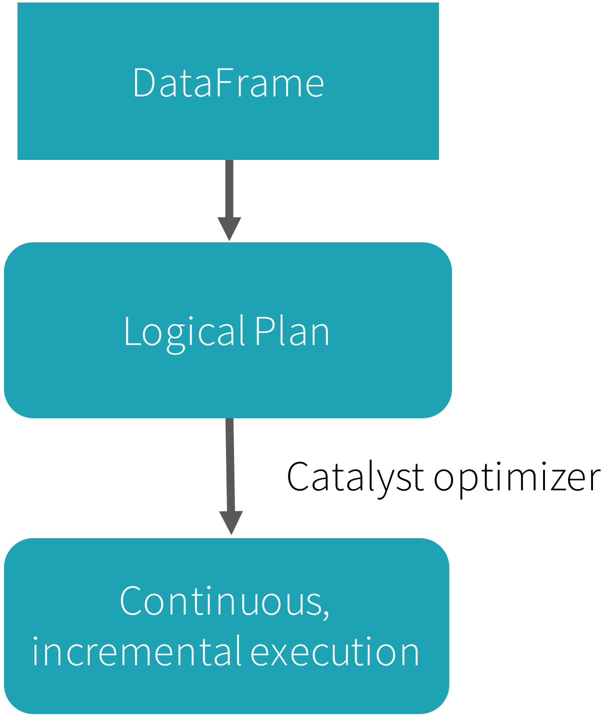
Multiple queries can be active at the same time

Each query has unique name for keeping track

Query Execution

Logically:
Dataset operations on table
(i.e. as easy to understand as batch)

Physically:
Spark automatically runs the query in streaming fashion
(i.e. incrementally and continuously)



Structured Streaming

High-level streaming API built on Datasets/DataFrames

- Event time, windowing, sessions, sources & sinks
- End-to-end exactly once semantics

Unifies streaming, interactive and batch queries

- Aggregate data in a stream, then serve using JDBC
- Add, remove, change queries at runtime
- Build and apply ML models

What can you do with this that's hard with other engines?

True unification

- Same code + same super-optimized engine for everything

Flexible API tightly integrated with the engine

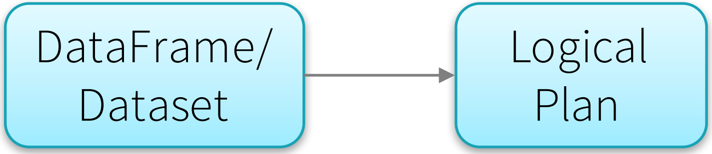
- Choose your own tool - Dataset/DataFrame/SQL
- Greater debuggability and performance

Benefits of Spark

- in-memory computing, elastic scaling, fault-tolerance, straggler mitigation, …

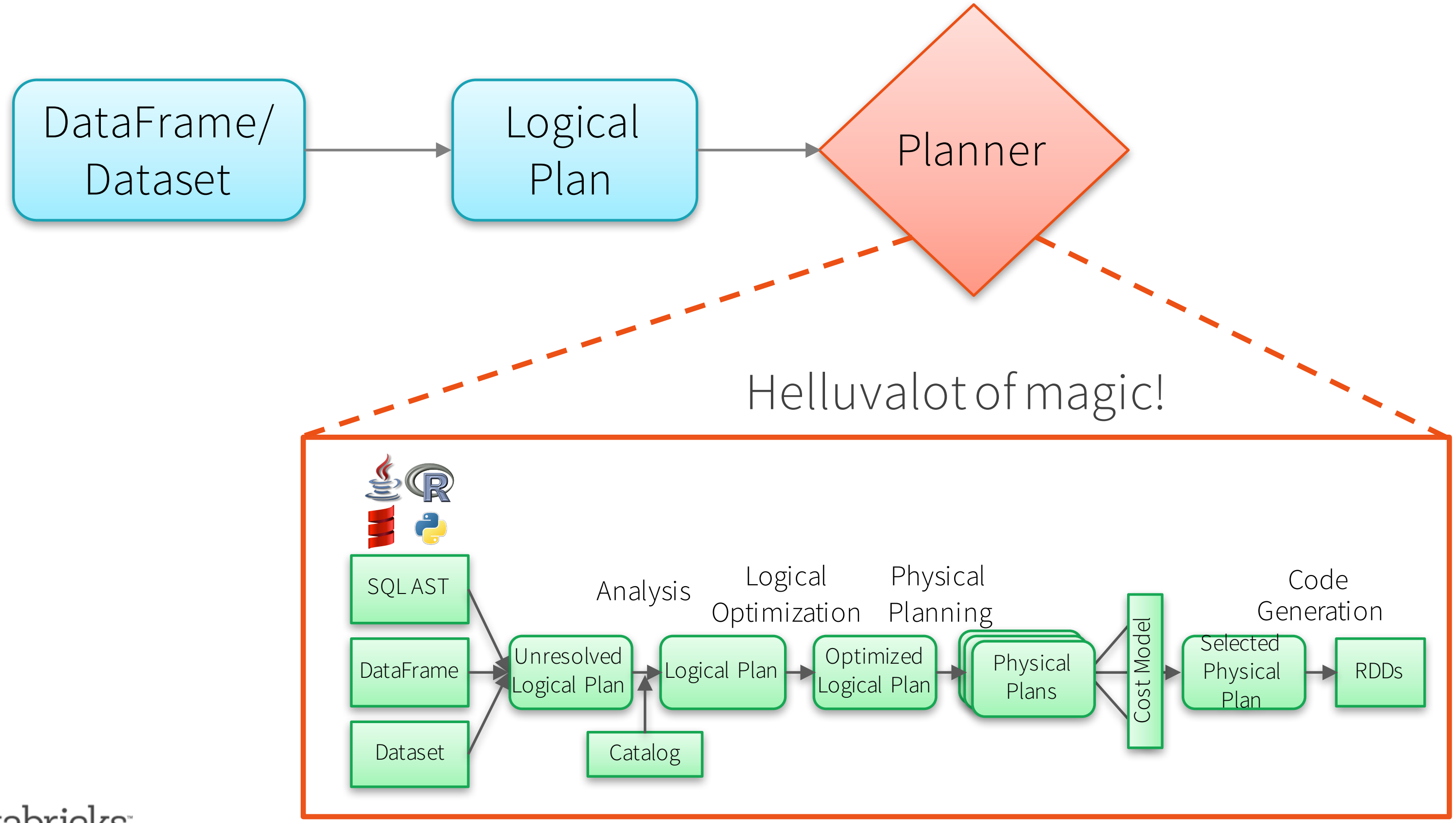
Underneath the Hood

Batch Execution on Spark SQL

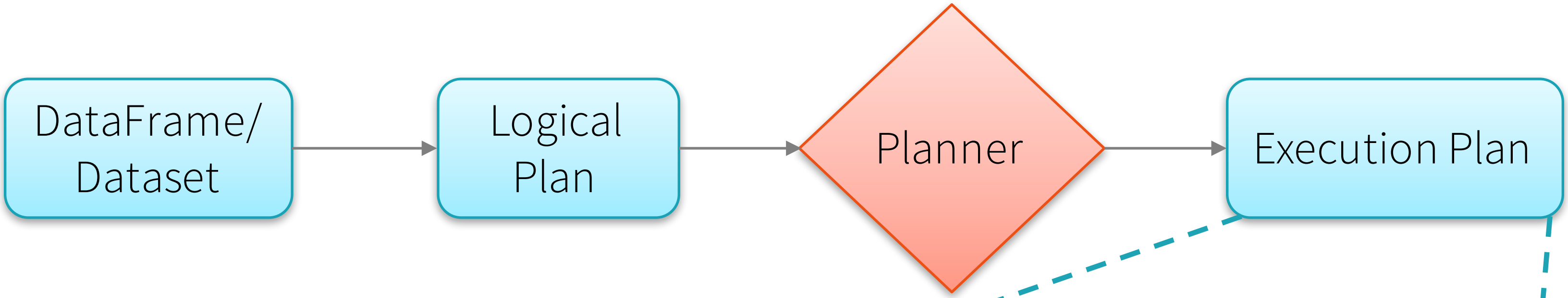


Abstract representation of query

Batch Execution on Spark SQL



Batch Execution on Spark SQL



Run super-optimized Spark jobs to compute results

Project Tungsten - Phase 1 and 2

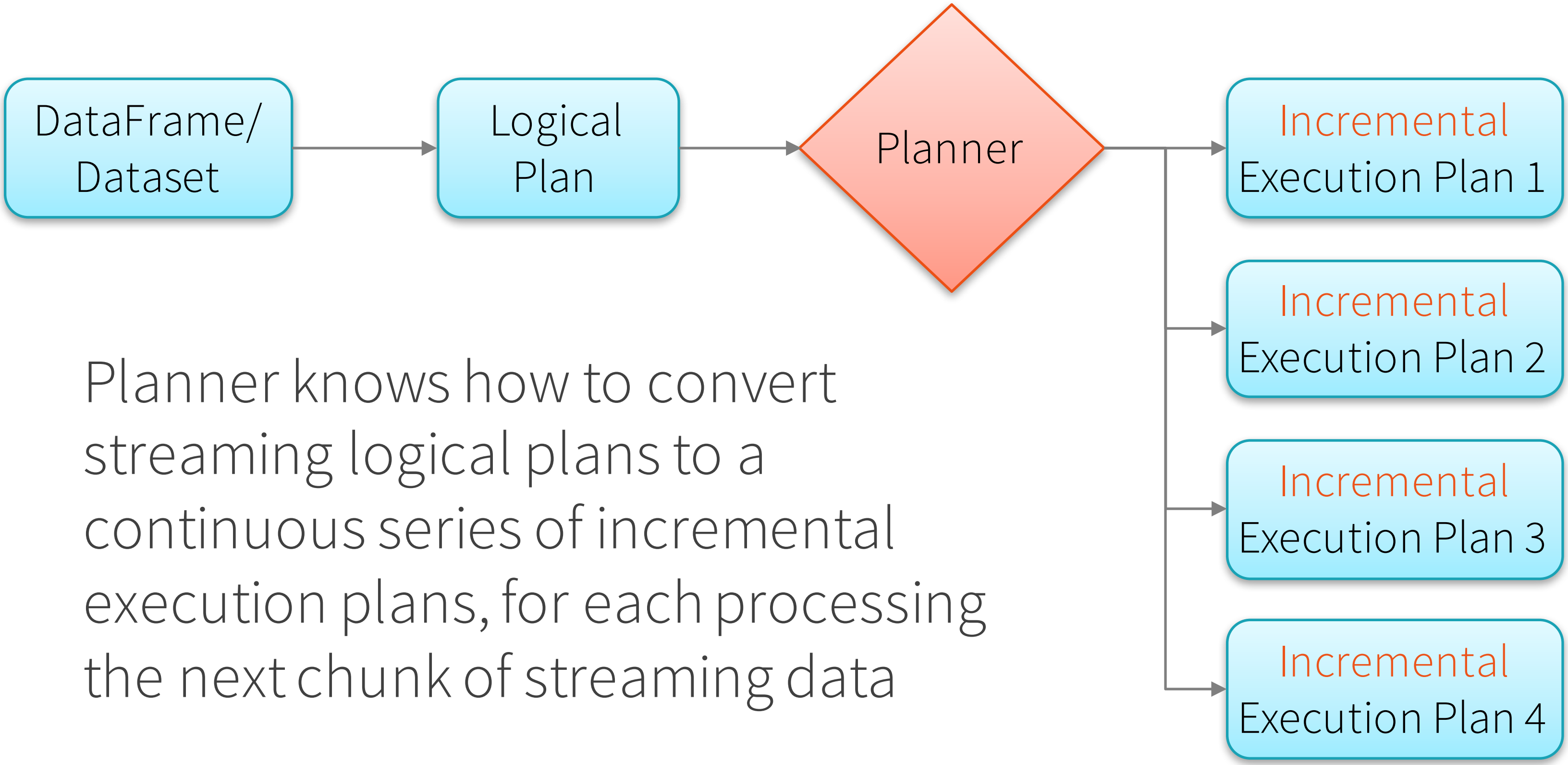
Code Optimizations

Bytecode generation
JVM intrinsics, vectorization
Operations on serialized data

Memory Optimizations

Compact and fast encoding
Offheap memory

Continuous Incremental Execution

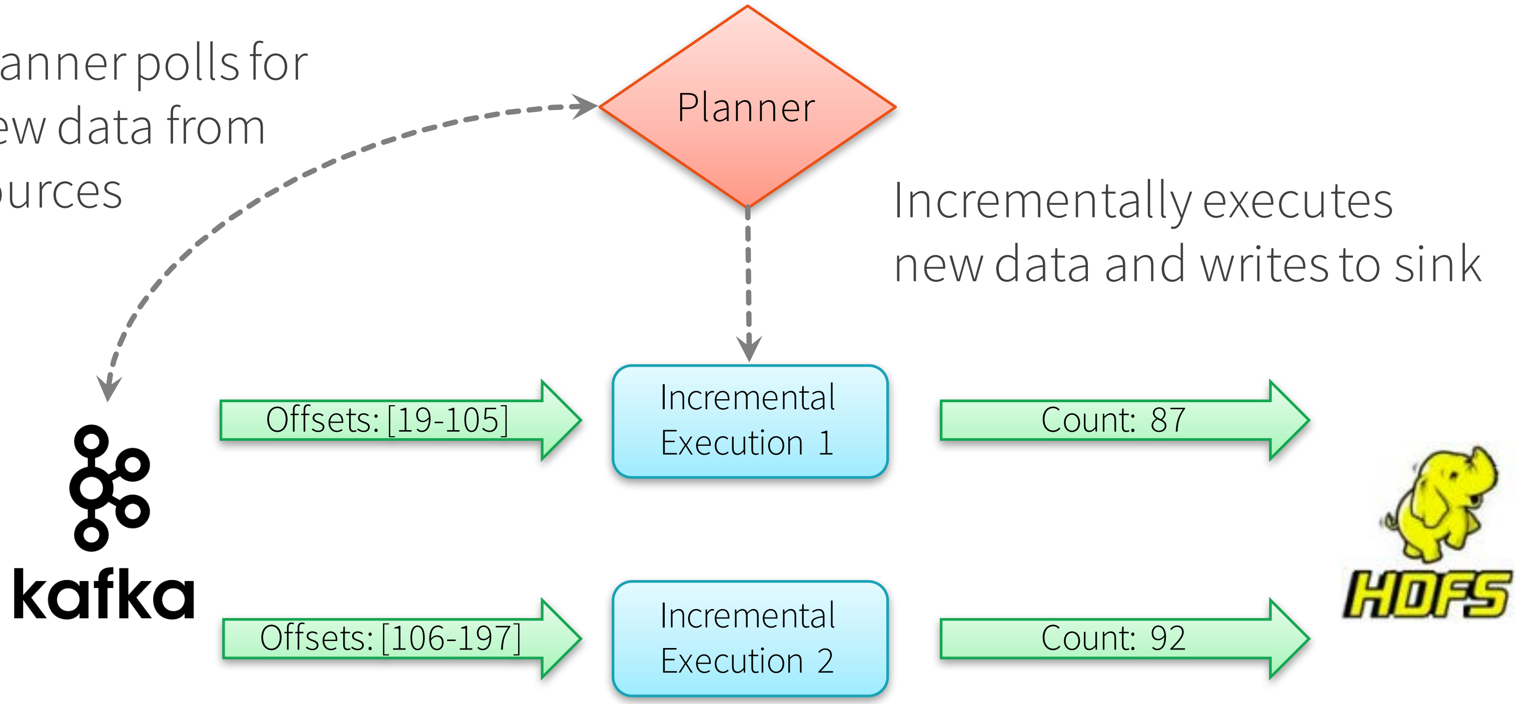


Planner knows how to convert streaming logical plans to a continuous series of incremental execution plans, for each processing the next chunk of streaming data

Continuous Incremental Execution

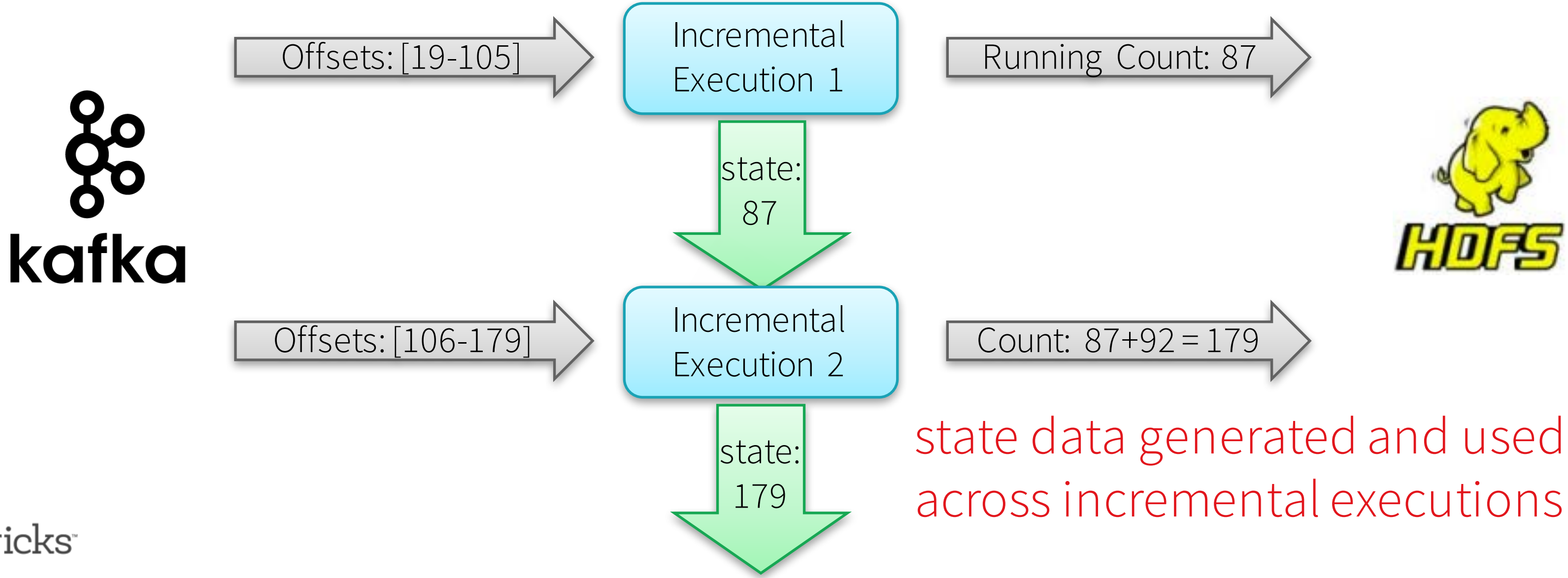
Planner polls for new data from sources

Incrementally executes new data and writes to sink



Continuous Aggregations

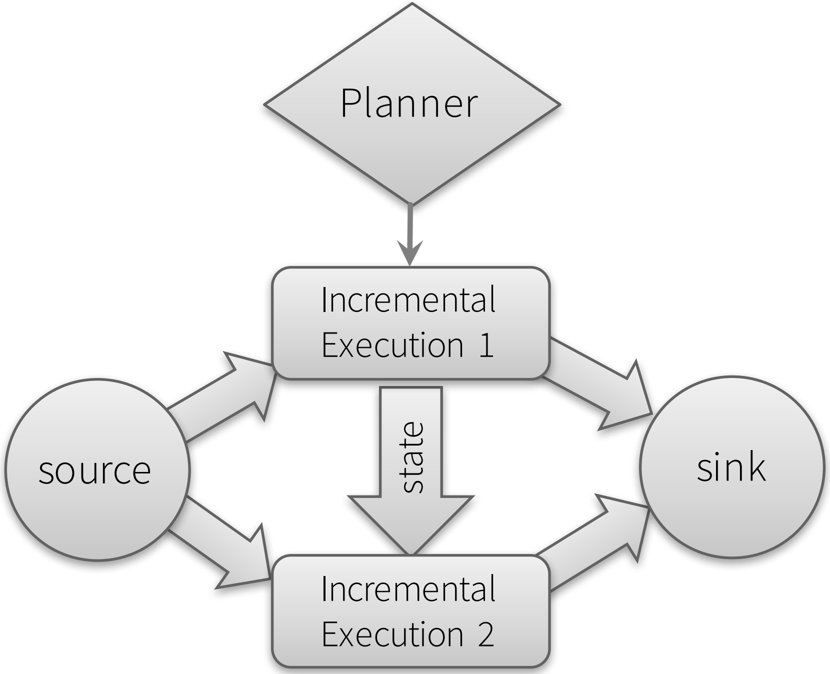
Maintain running aggregate as in-memory state backed by WAL in file system for fault-tolerance



state data generated and used across incremental executions

Fault-tolerance

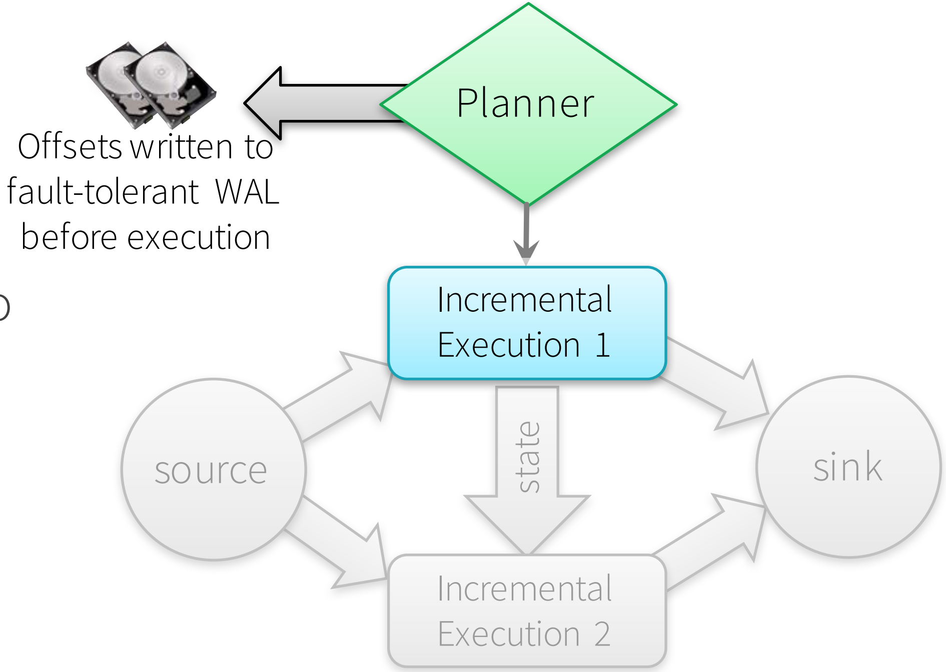
All data and metadata in the system needs to be recoverable / replayable



Fault-tolerance

Fault-tolerant Planner

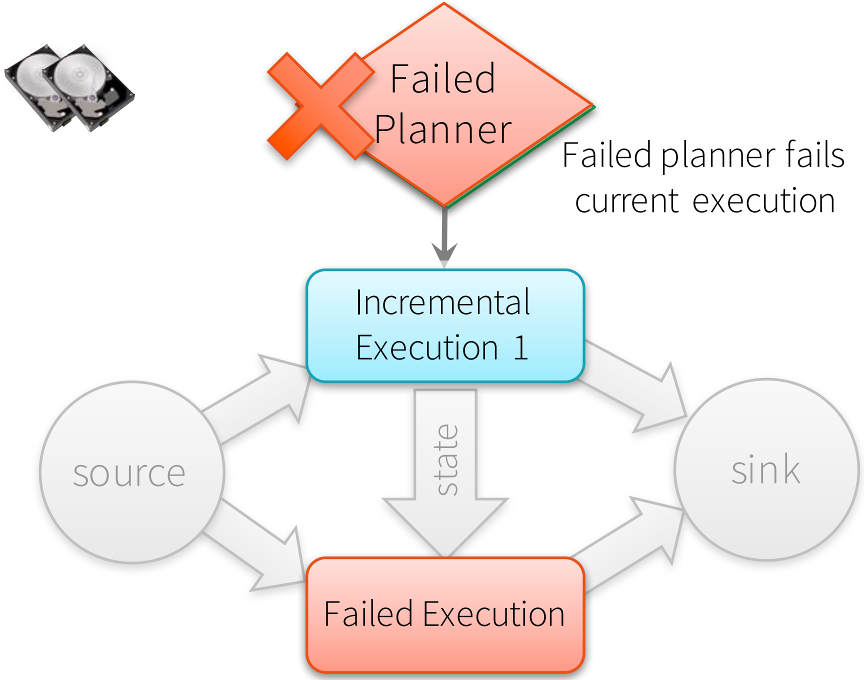
Tracks offsets by writing the offset range of each execution to a write ahead log (WAL) in HDFS



Fault-tolerance

Fault-tolerant Planner

Tracks offsets by writing the offset range of each execution to a write ahead log (WAL) in HDFS

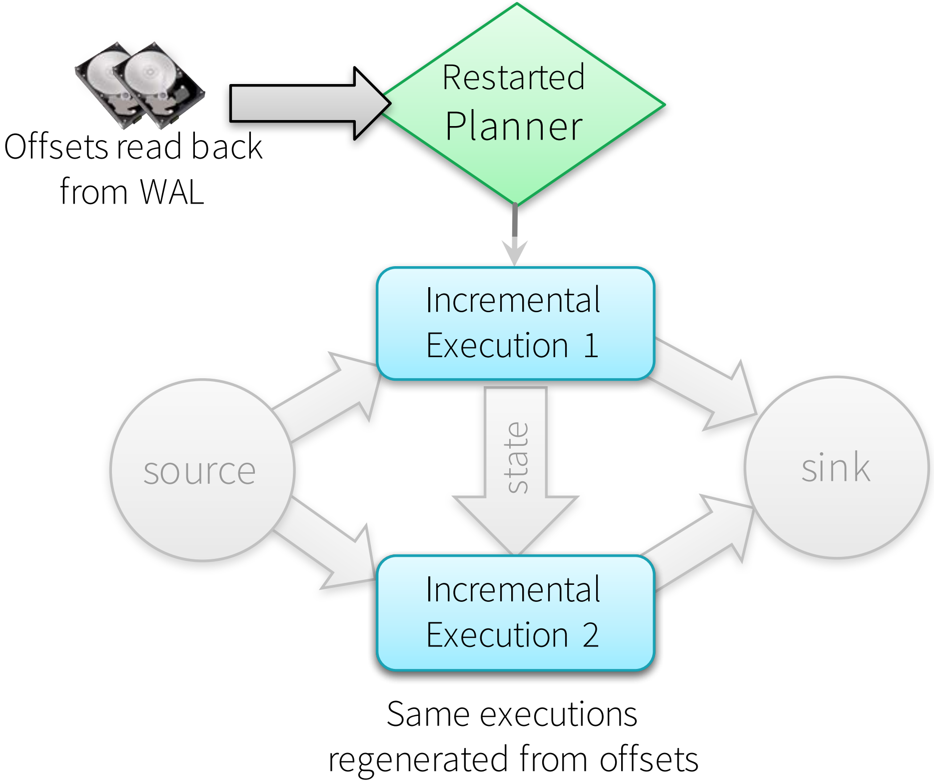


Fault-tolerance

Fault-tolerant Planner

Tracks offsets by writing the offset range of each execution to a write ahead log (WAL) in HDFS

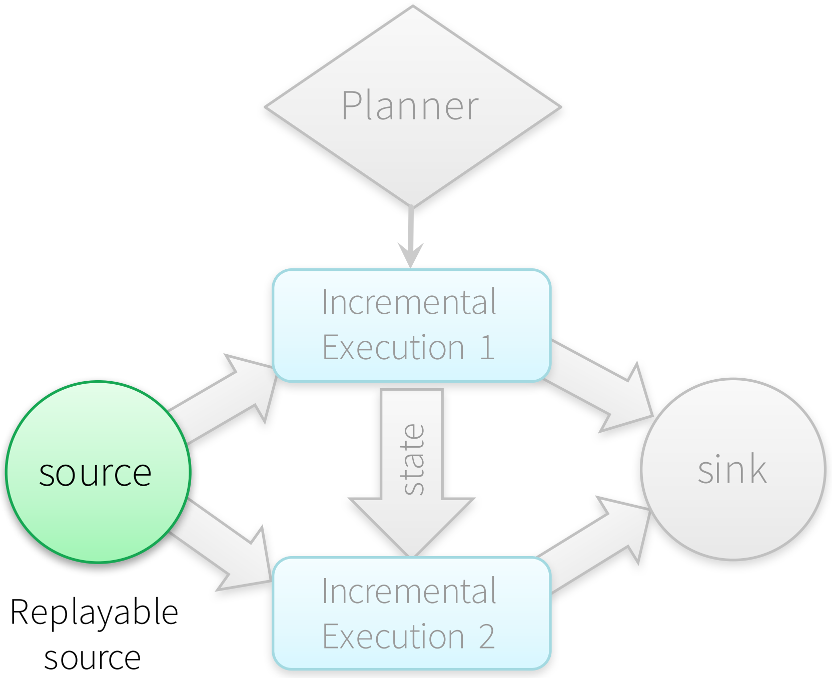
Reads log to recover from failures, and re-execute exact range of offsets



Fault-tolerance

Fault-tolerant Sources

Structured streaming sources are by design replayable (e.g. Kafka, Kinesis, files) and generate the exactly same data given offsets recovered by planner

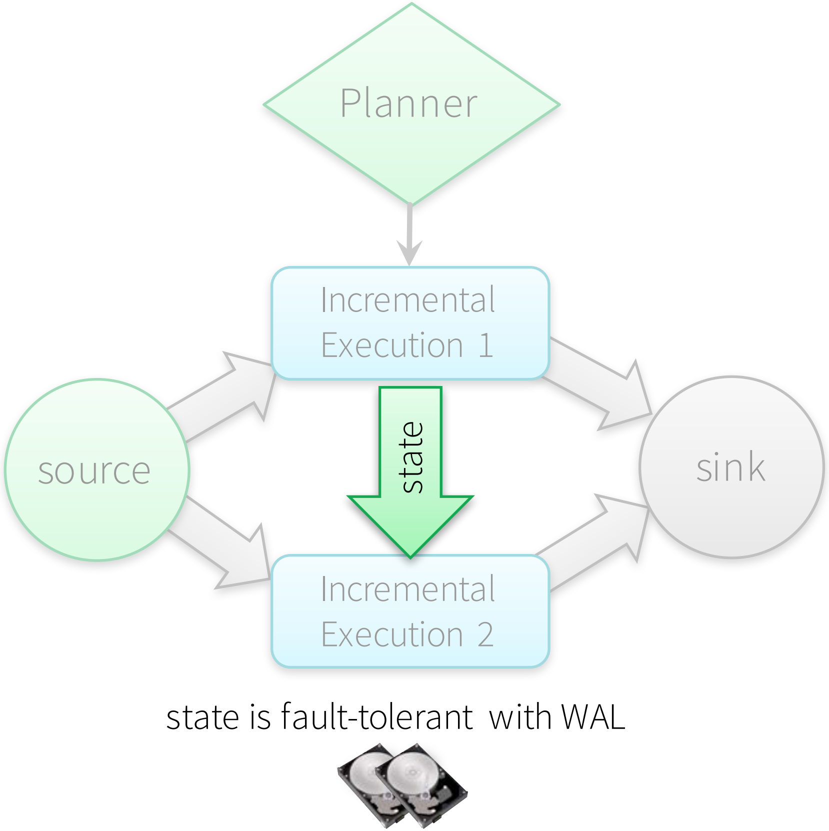


Fault-tolerance

Fault-tolerant State

Intermediate "state data" is a maintained in versioned, key-value maps in Spark workers, backed by HDFS

Planner makes sure "correct version" of state used to re-execute after failure

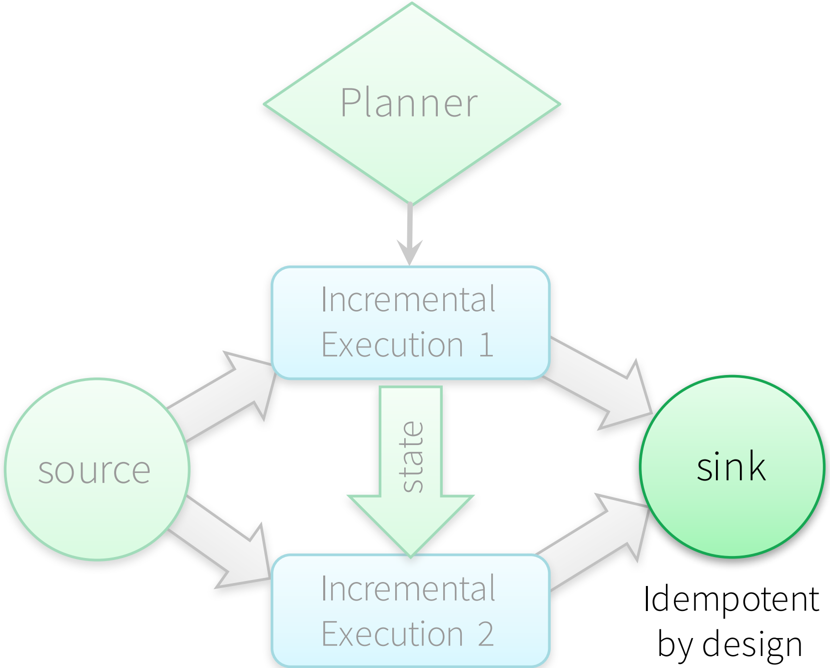


databricks

Fault-tolerance

Fault-tolerant Sink

Sink are by design idempotent, and handles re-executions to avoid double committing the output



offset tracking in WAL

+

state management

+

fault-tolerant sources and sinks

=

end-to-end exactly-once guarantees

Fast, fault-tolerant, exactly-once

stateful stream processing

without having to *reason* about streaming

Release Plan: Spark 2.0 [June 2016]

Basic infrastructure and API

- Event time, windows, aggregations
- Append and Complete output modes
- Support for a subset of batch queries

Source and sink

- Sources: Files (*Kafka coming soon after 2.0 release)
- Sinks: Files and in-memory table

Experimental release to set the future direction

Not ready for production but good to experiment with and provide feedback

Release Plan: Spark 2.1+

- Stability and scalability
- Support for more queries
  - Multiple aggregations
  - Sessionization
  - More output modes
  - Watermarks and late data
- Sources and Sinks
  - Public APIs
- ML integrations

Make Structured Streaming ready for production workloads as soon as possible

Try Apache Spark with Databricks

Stay tuned on our Databricks blogs for more information and examples on Structured Streaming

Try latest version of Apache Spark and preview of Spark 2.0

http://databricks.com/try

Structured Streaming

Making Continuous Applications
easier, *faster*, and *smarter*

AMA @
Databricks Booth

Today: Now - 2:00 PM
Tomorrow: 12:15 PM - 1:00 PM

Follow me @tathadas

databricks™