A Deep Dive into Structured Streaming

范文臣

databricks

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

building robust stream processing apps is hard

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

databricks

you

should write simple queries

&

Spark

should continuously update the answer

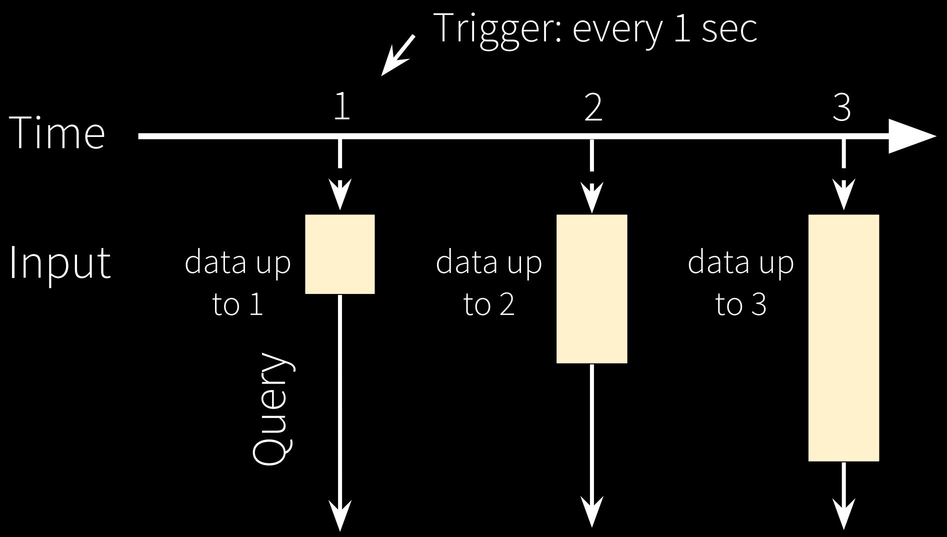
Structured Streaming Model

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

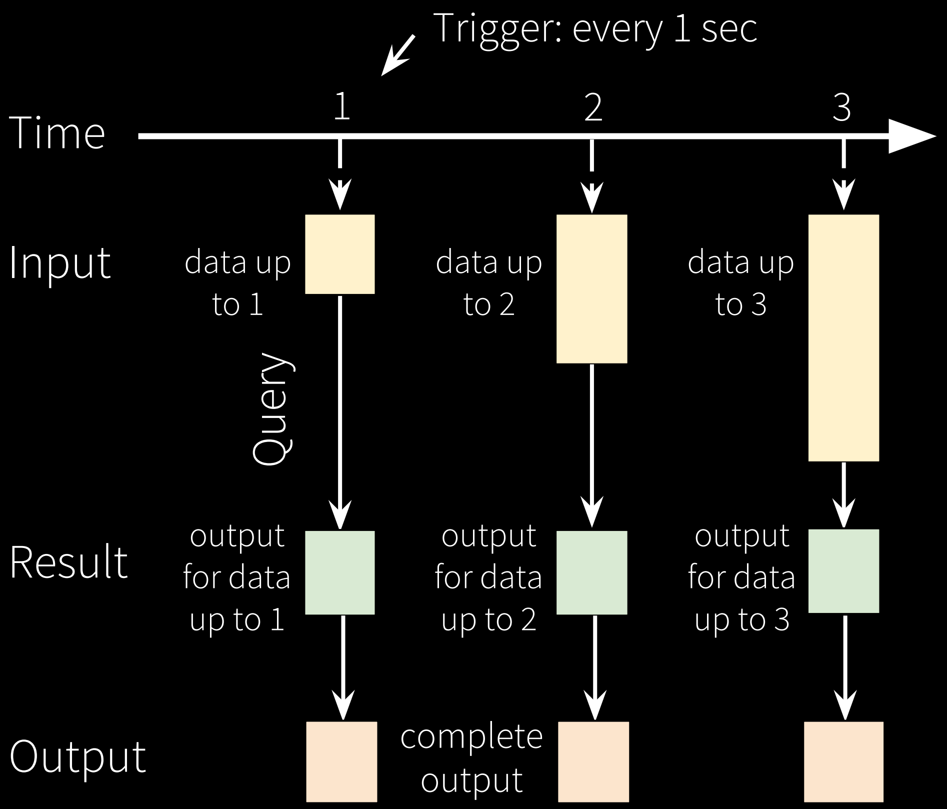


Model

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



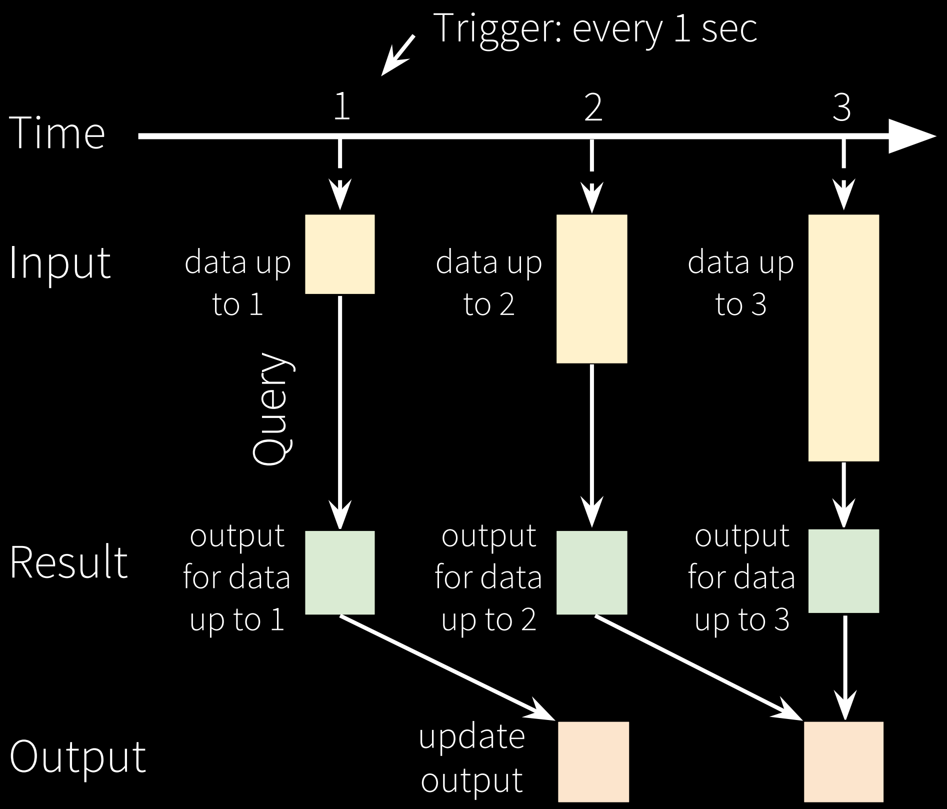
Model

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
Update 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

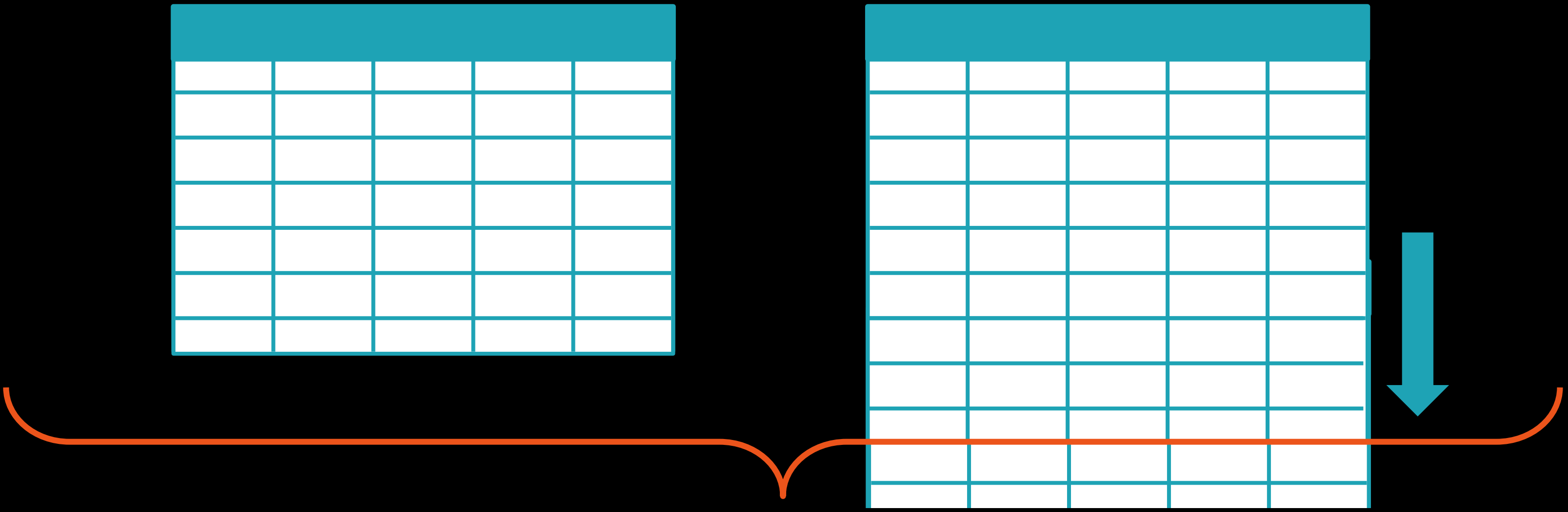


databricks

API - *Dataset/DataFrame*

Static, bounded data

Streaming, unbounded data



Single API !

Streaming word count

databricks

Anatomy of a Streaming Word Count

```
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.

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.

databricks

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("append")
```

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

databricks

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("append")
  .option("checkpointLocation", "…")
  .start()
```

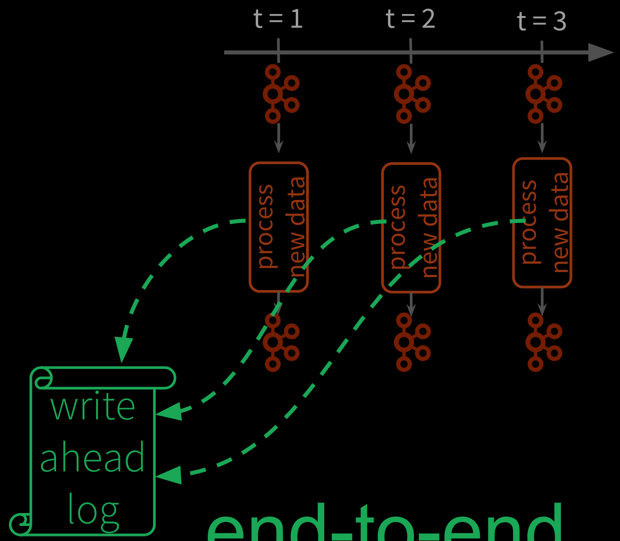
Checkpoint

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

Fault-tolerance with Checkpointing

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

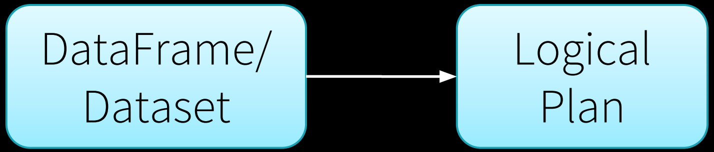
- Offsets and metadata saved as JSON
- Can resume after changing your streaming transformations



end-to-end exactly-once guarantees

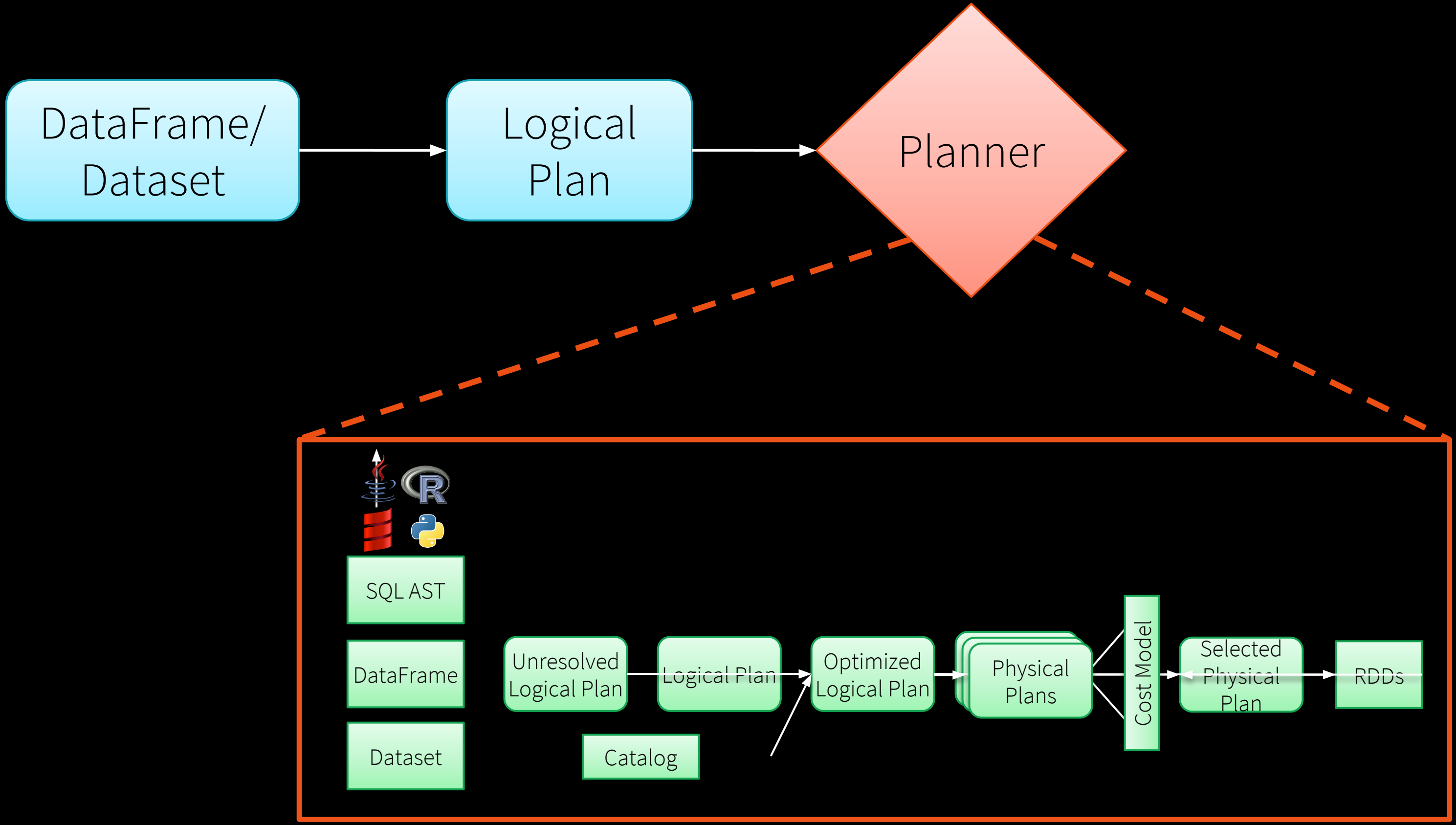
Underneath the Hood

Batch Execution on Spark SQL

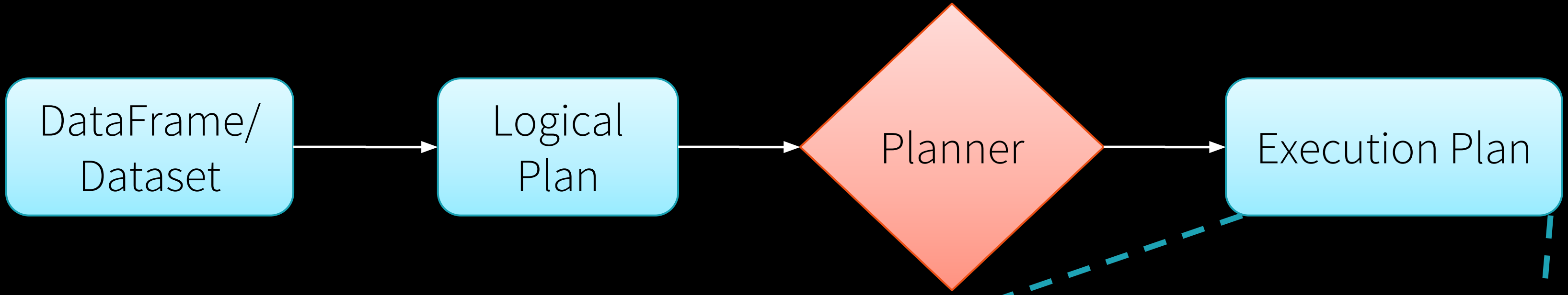


Abstract representation of query

Batch Execution on Spark SQL



Batch Execution on Spark SQL



Run super-optimized Spark jobs to compute results

Project Tungsten

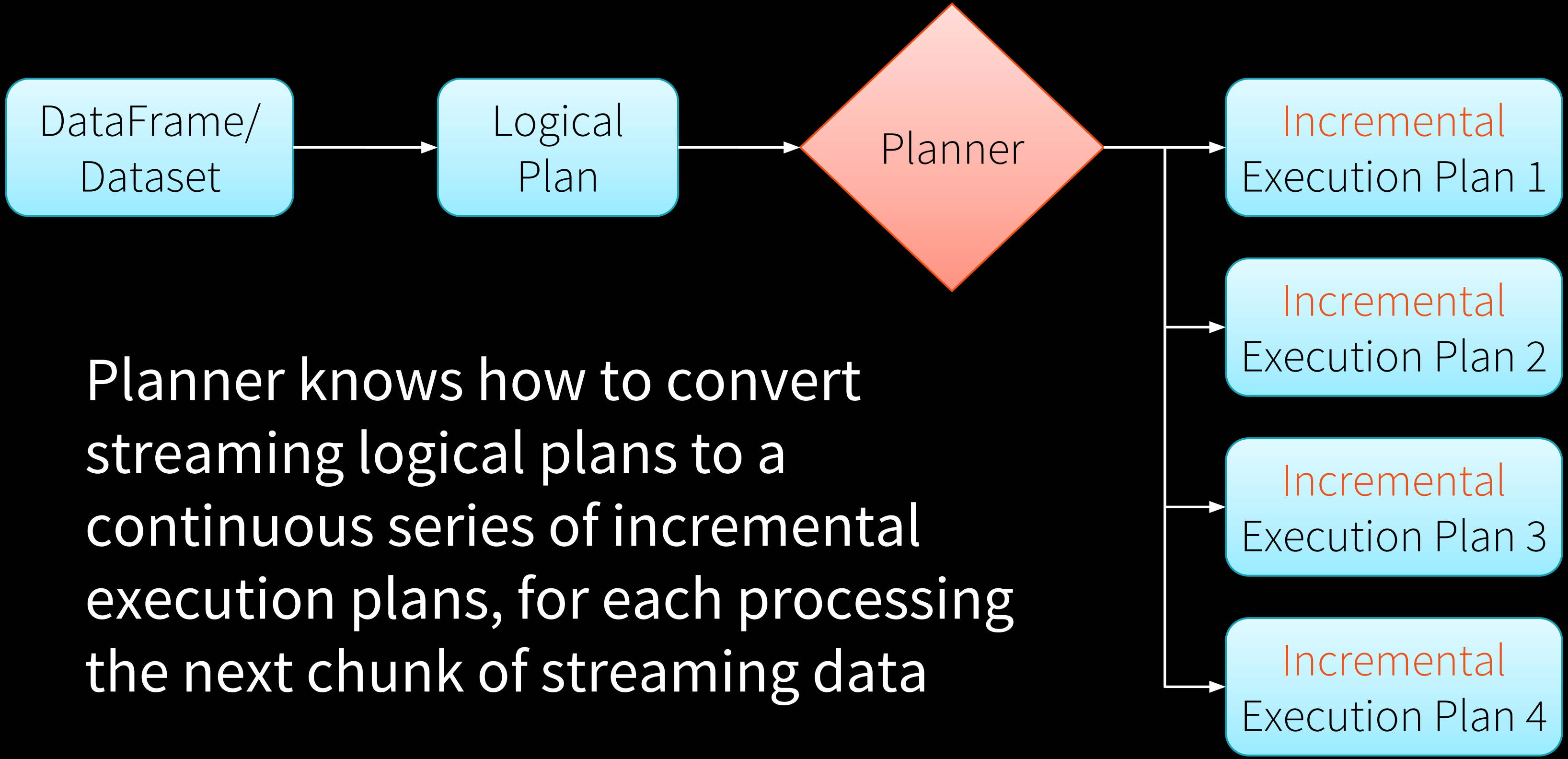
Code Optimizations

Bytecode generation
JVM intrinsics, vectorization
Operations on serialized data

Memory Optimizations

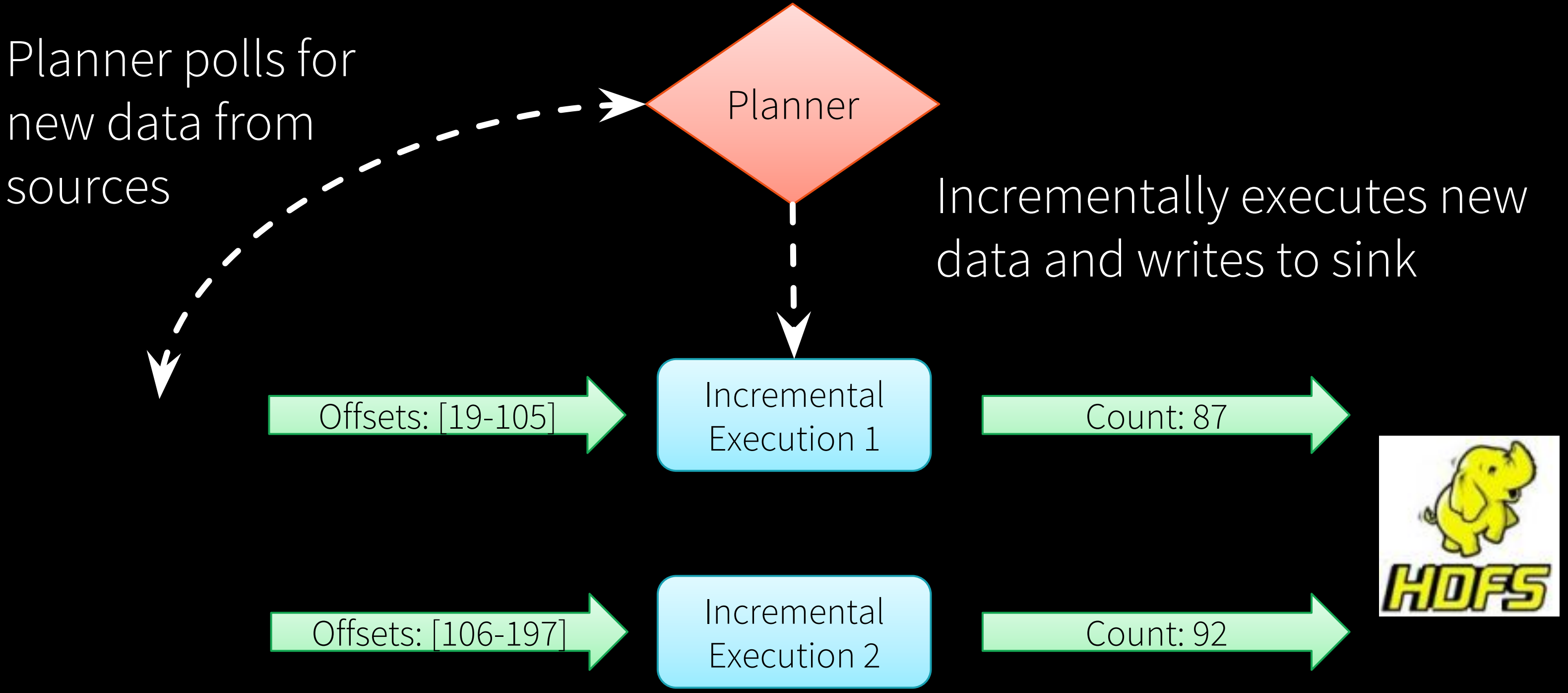
Compact and fast encoding
Offheap memory

Continuous Incremental Execution



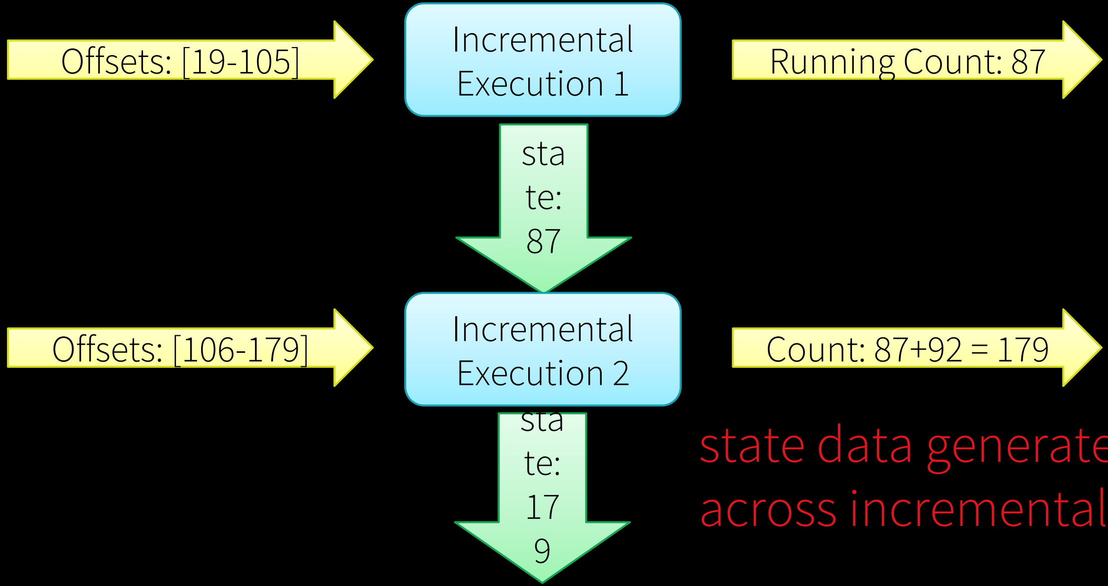
Planner knows how to convert streaming logical plans to a continuous series of incremental execution plans, for each processing the next chunk of streaming data

Continuous Incremental Execution



Continuous Aggregations

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

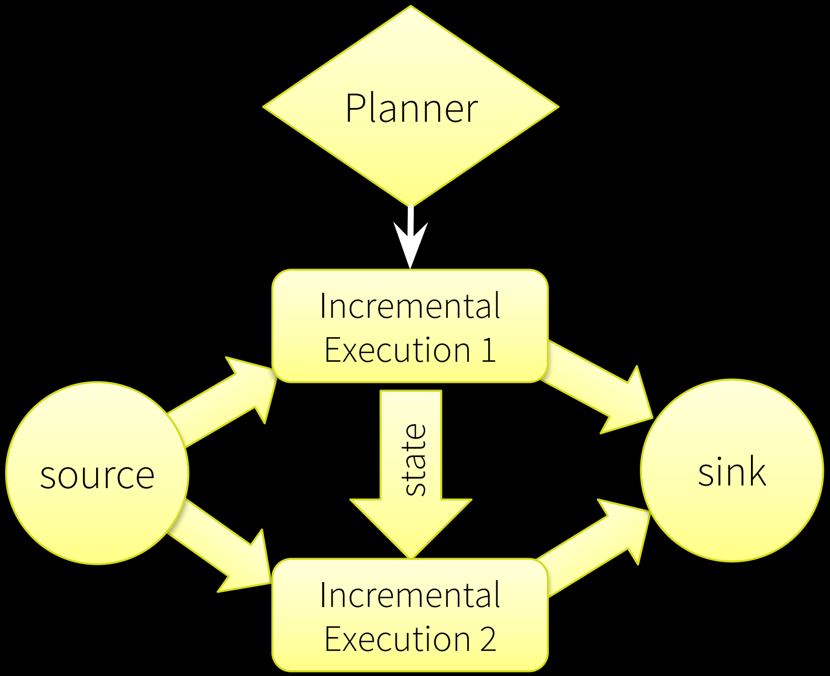


HDFS

state data generated and used across incremental executions

Fault-tolerance

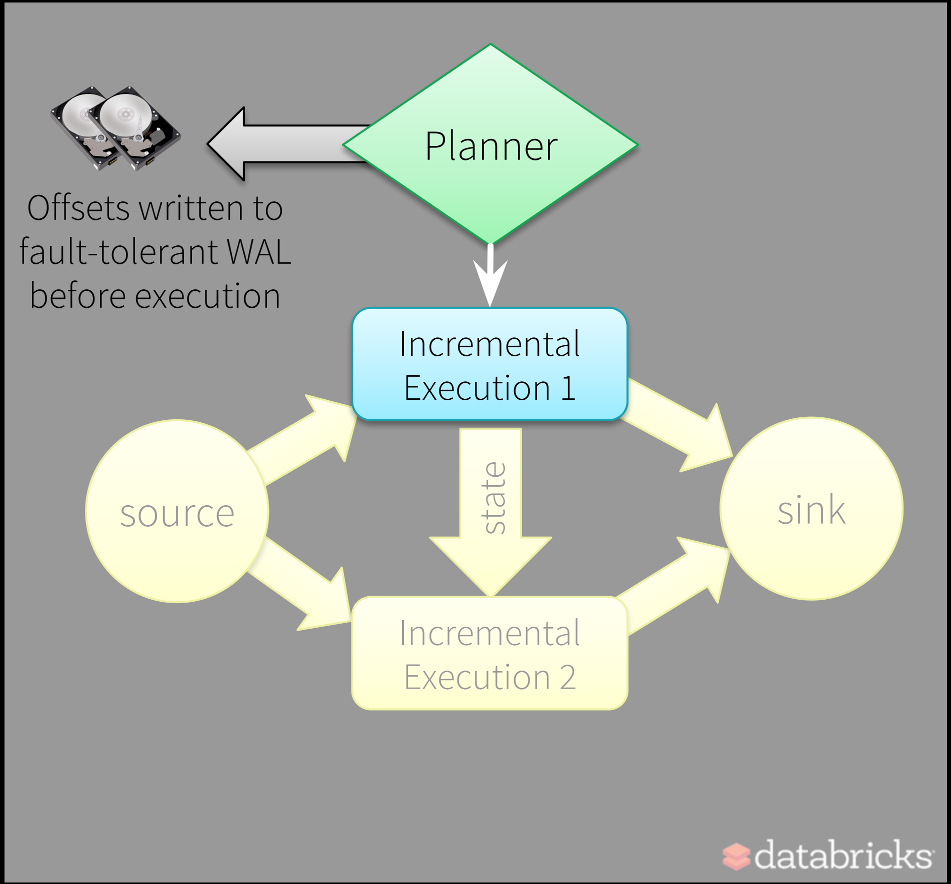
All data and metadata in the system needs to be recoverable / replayable



Fault-tolerance

Fault-tolerant Planner

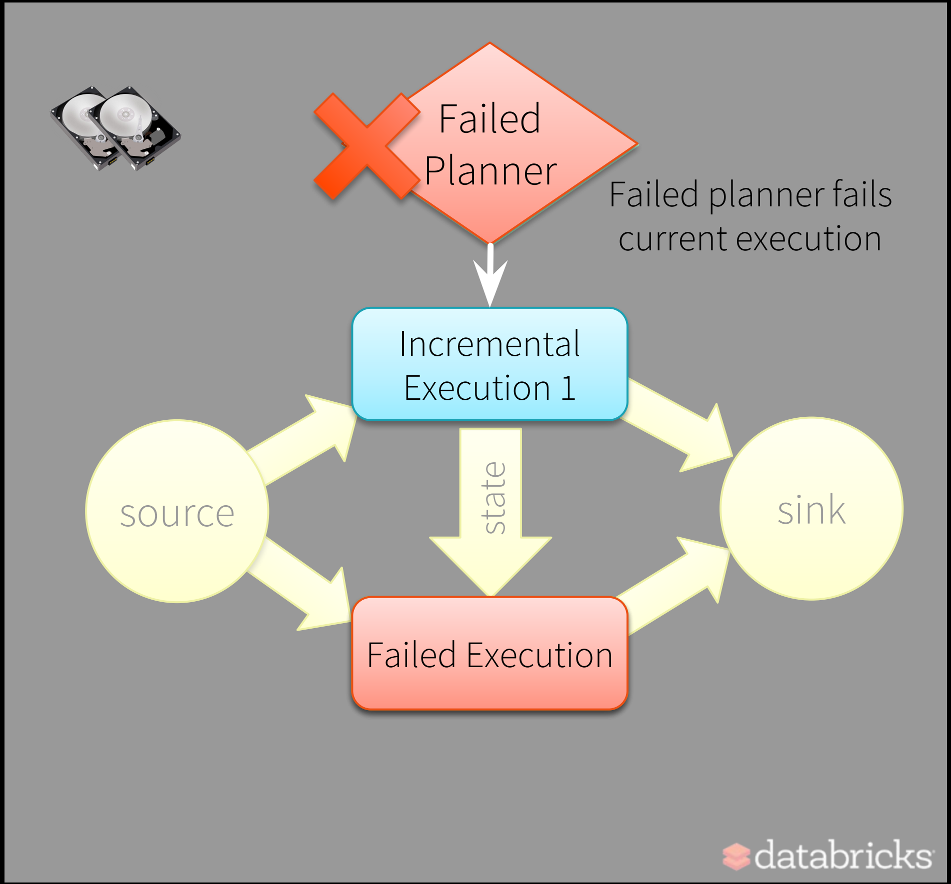
Tracks offsets by writing the offset range of each execution to a write ahead log (WAL) in HDFS



Fault-tolerance

Fault-tolerant Planner

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

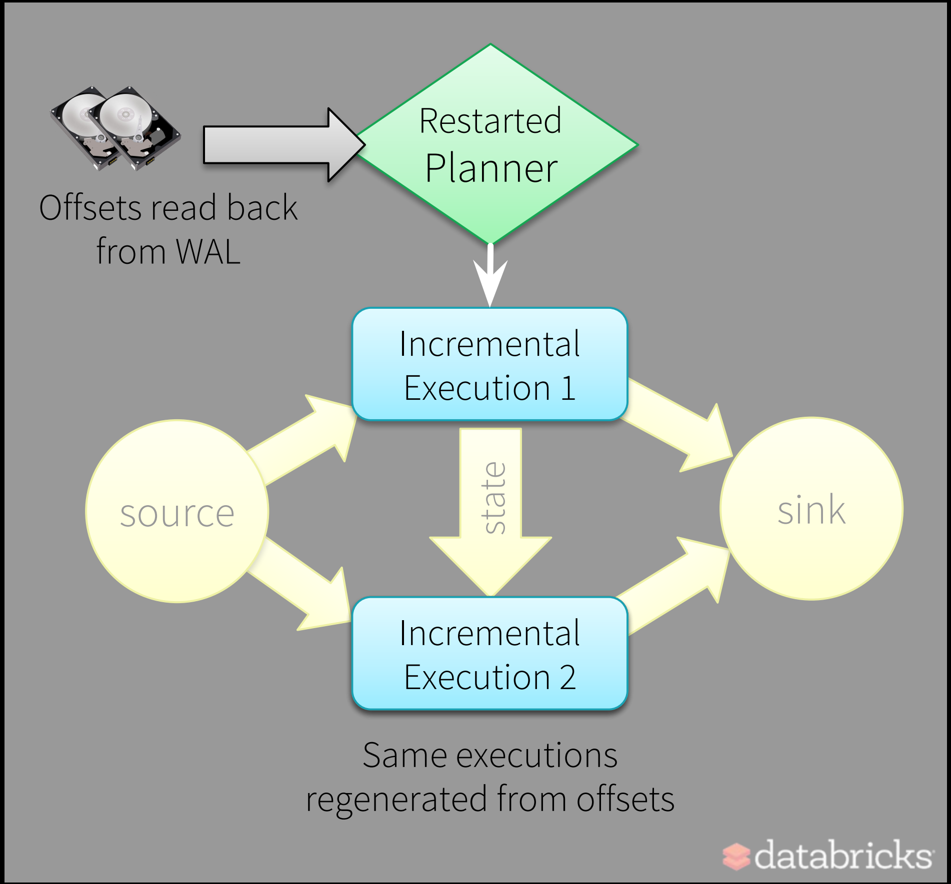


Fault-tolerance

Fault-tolerant Planner

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

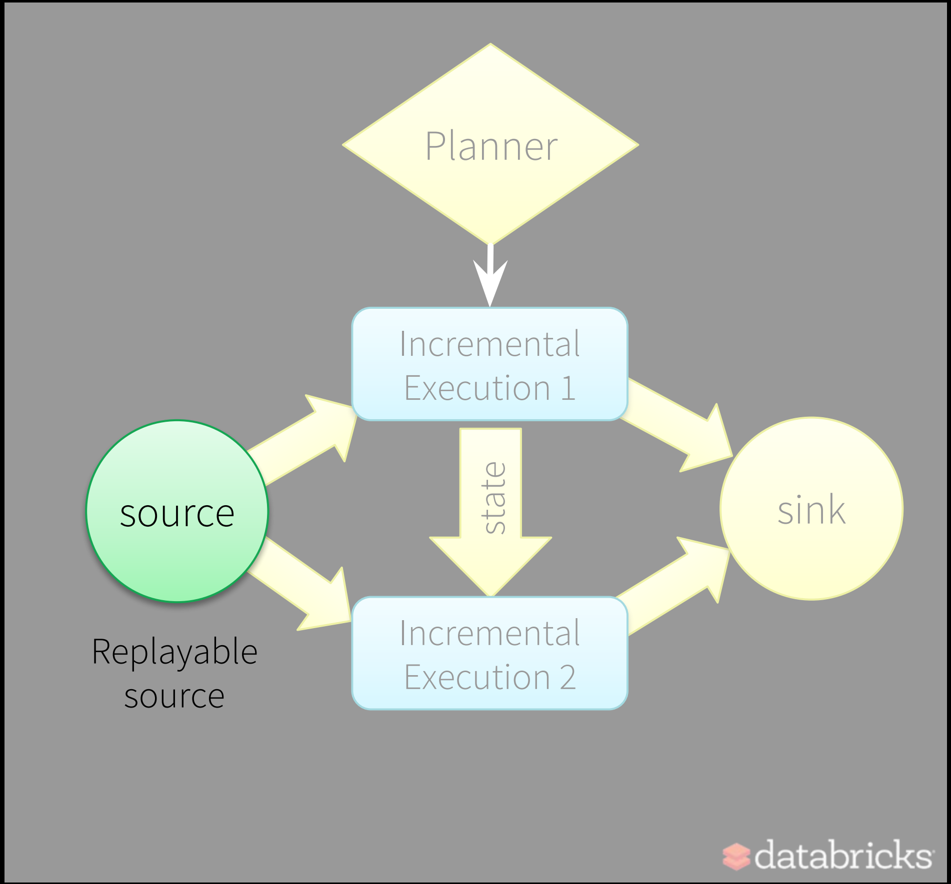
Reads log to recover from failures, and re-execute exact range of offsets



Fault-tolerance

Fault-tolerant Sources

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

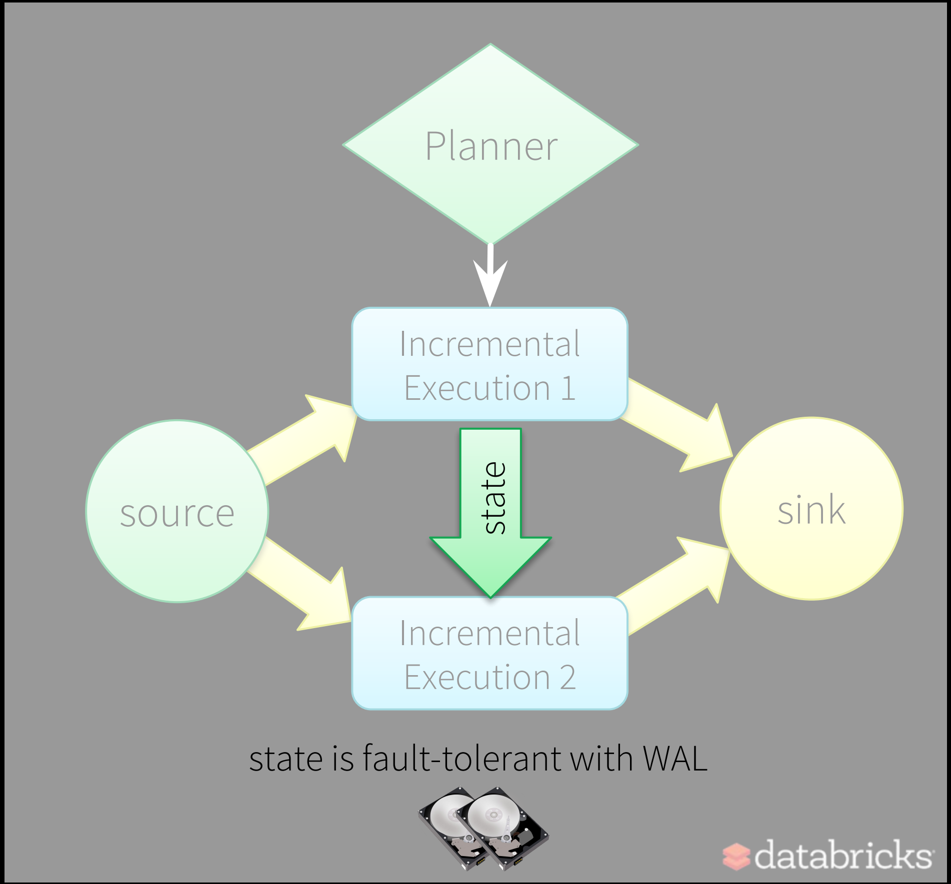


Fault-tolerance

Fault-tolerant State

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

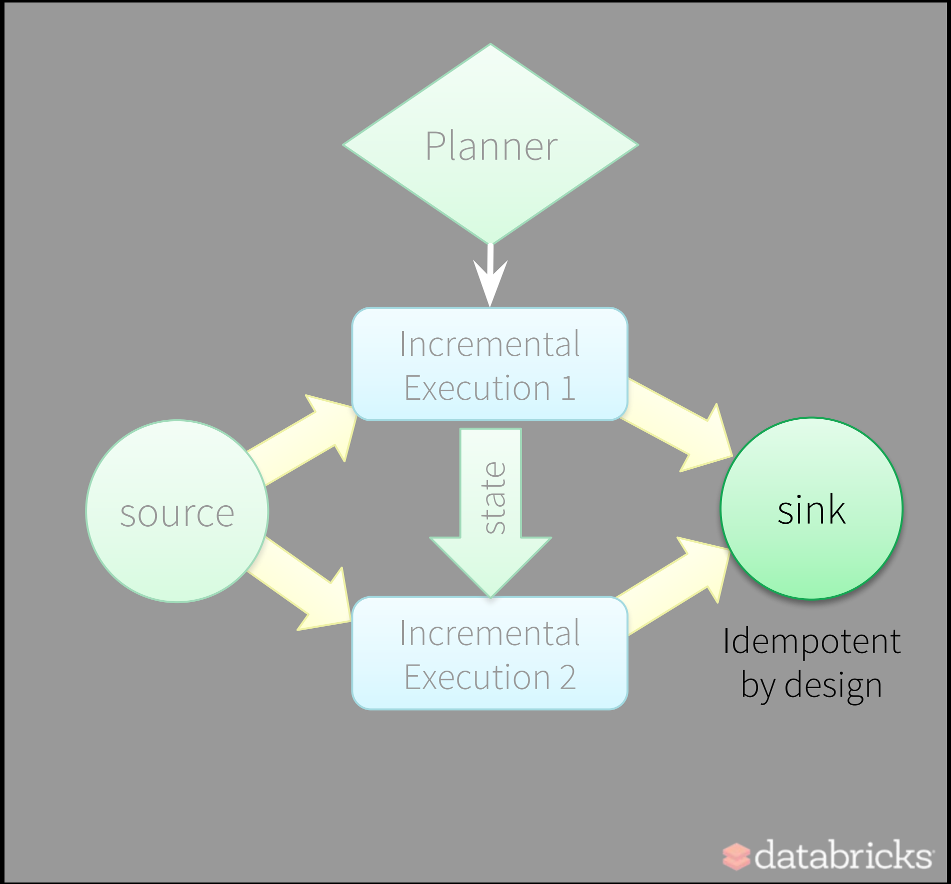
Planner makes sure "correct version" of state used to re-execute after failure



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

Continuous Processing

Continuous processing mode to run without micro-batches

Long running Spark tasks and checkpoint periodically

<=1 ms latency (same as per-record streaming systems)

1 line change: **trigger(Trigger.Continuous("1 second"))**

Added in Spark 2.3

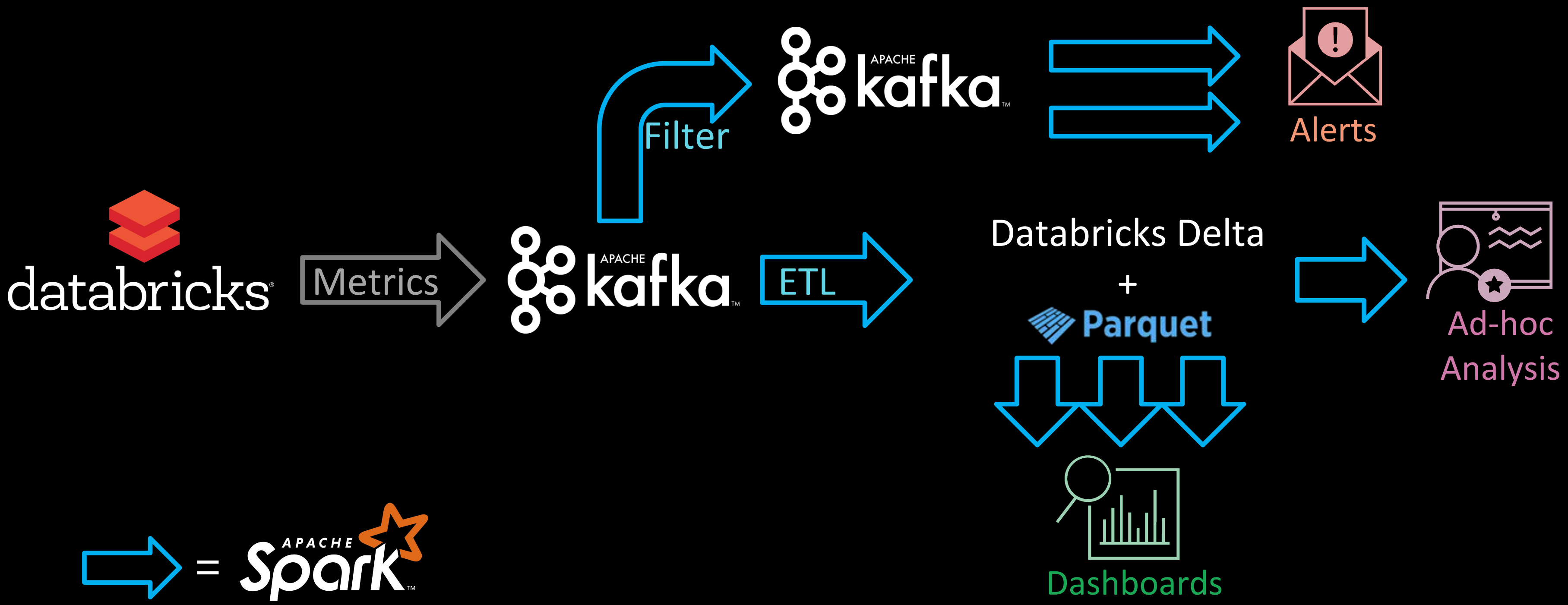
Building Complex Continuous Apps

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



Read from 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 → Databricks Delta + Parquet

Store augmented stream as efficient columnar data for later processing

Latency: ~1 minute

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

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

Dashboards

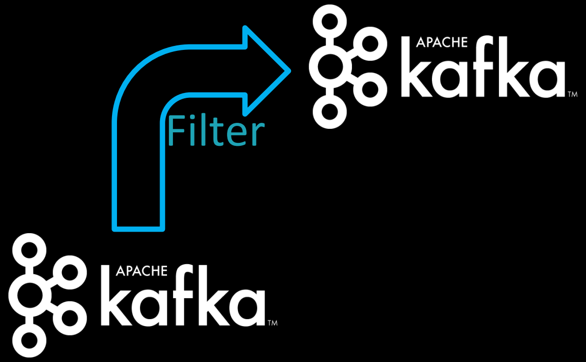
Always up-to-date visualizations of important business trends

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

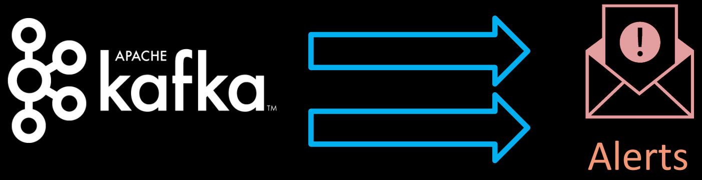
Latency: ~100 ms average

```
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

Alerts



E.g. Alert when Spark cluster load > threshold

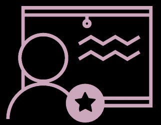
Latency: ~100 ms

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

notify PagerDuty

Ad-hoc Analysis

Databricks Delta
+
Parquet



Ad-hoc
Analysis

Trouble shoot problems as they occur with latest information

Latency: ~1 minute

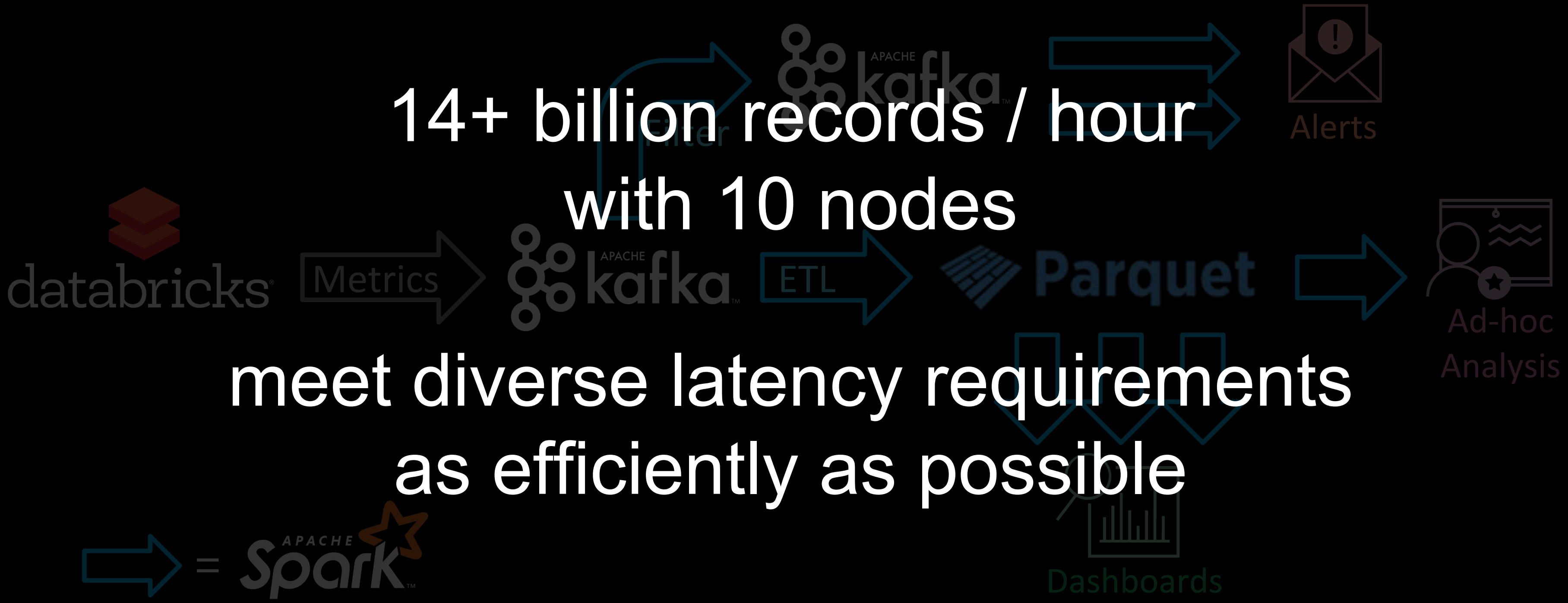
```
SELECT *
FROM delta.`/data/metrics`
WHERE level IN ('WARN', 'ERROR')
  AND customer = "..."
  AND timestamp < now() - INTERVAL 1 HOUR
```

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



Structured Streaming @ databricks

100s of customer streaming apps in production on Databricks

Largest app process 10s of trillions of records per month

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

https://databricks.com/blog/2017/10/11/benchmarking-structured-streaming-on-databricks-runtime-against-state-of-the-art-streaming-systems.html

https://databricks.com/blog/2017/10/17/arbitrary-stateful-processing-in-apache-sparks-structured-streaming.html

https://databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-spark-2-3.html

https://databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3-0.html

and more to come, stay tuned!!

Thank You

Q&A