Data Analytics using

APACHE Spark™

Unified engine for large-scale data analytics

Spark Streaming

Abhay Mane
Big Data Analytics & Machine Learning Team
C-DAC Bengaluru
abhaym@cdac.in

Introduction

- Extension of the core Spark API
- Enables **stream processing of live data streams**.
- Data can be ingested from: Kafka, Flume, Twitter, or **TCP sockets**
- Finally, processed data can be pushed out to filesystems, databases, and live dashboards.
- you can apply Spark's machine learning and graph processing algorithms on data streams.

How does it work?

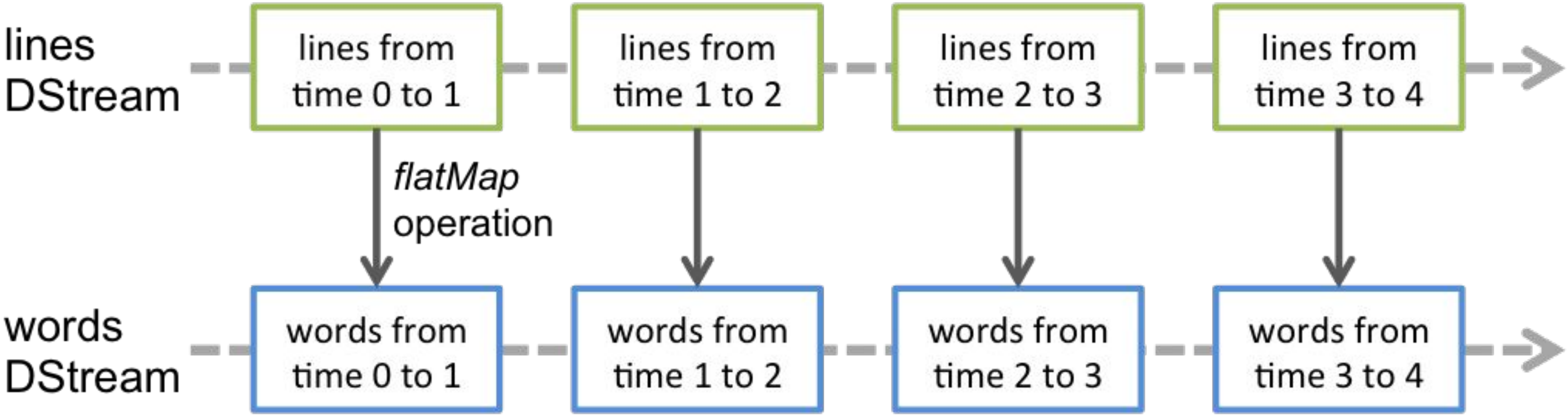
- Chop up **data streams into batches** of few secs
- Spark treats **each batch of data as RDDs** and processes them using RDD operations
- Processed results are pushed out in batches

Discretized Streams (DStreams)

- Basic abstraction provided by Spark streaming
- Represents a continuous stream data
- Internally **DStream is represented as a continuous series of RDDs**
- DStream API is very similar to RDD API

Discretized Streams (DStreams)

- Any operation applied on a DStream translates to operations on the underlying RDDs.
- **Converting a stream of lines to words,** the flatMap operation is applied on each RDD in the lines DStream to generate the RDDs of the words DStream.



Structured Streaming

Structured Streaming

- Spark Structured Streaming was introduced in Spark 2.0 (and became stable in 2.2) as an extension built on top of Spark SQL
- It takes advantage of Spark SQL code and memory optimizations.

Structured Streaming

stream processing on Spark SQL engine

fast, scalable, fault-tolerant

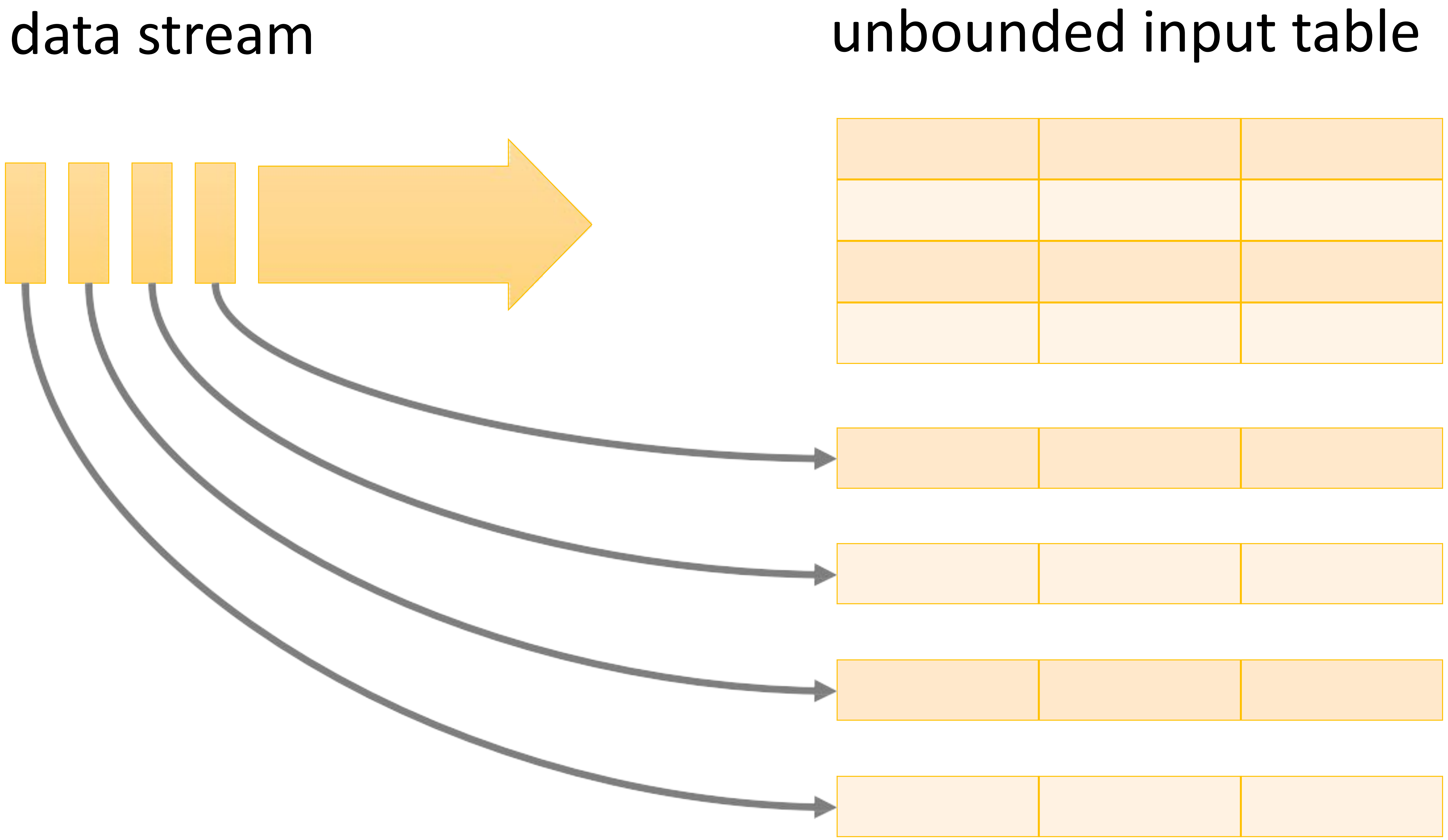
rich, unified, high level APIs

with *complex data* and *complex workloads*

rich ecosystem of data sources

integrate with many *storage systems*

Treat Streams as Unbounded Tables (Programming Model)



new data in the data stream

=

new rows appended to a unbounded table

Treat Streams as Unbounded Tables (Programming Model)

The "Output" is defined as what gets written out to the external storage. The output can be defined in a different mode:

- ***Complete Mode*** - The entire updated Result Table will be written to the external storage. It is up to the storage connector to decide how to handle writing of the entire table.
- ***Append Mode*** - Only the new rows appended in the Result Table since the last trigger will be written to the external storage. This is applicable only on the queries where existing rows in the Result Table are not expected to change.
- ***Update Mode*** - Only the rows that were updated in the Result Table since the last trigger will be written to the external storage (available since Spark 2.1.1). Note that this is different from the Complete Mode in that this mode only outputs the rows that have changed since the last trigger. If the query doesn't contain aggregations, it will be equivalent to Append mode.

Input Streaming Sources

In Spark 3.0, there are a few built-in sources.

File source - Reads files written in a directory as a stream of data. Supported file formats are text, csv, json, parquet. Note that the files must be atomically placed in the given directory, which in most file systems, can be achieved by file move operations.

Kafka source - Poll data from Kafka. It's compatible with Kafka broker versions 0.10.0 or higher. See the Kafka Integration Guide for more details.

Socket source (for testing) - Reads UTF8 text data from a socket connection. The listening server socket is at the driver. Note that this should be used only for testing as this does not provide end-to-end fault-tolerance guarantees.

Input Streaming Sources

```
spark = SparkSession. ...

# Read text from socket
socketDF = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

socketDF.isStreaming()    # Returns True for DataFrames that have streaming sources

socketDF.printSchema()

# Read all the csv files written atomically in a directory
userSchema = StructType().add("name", "string").add("age", "integer")
csvDF = spark \
    .readStream \
    .option("sep", ";") \
    .schema(userSchema) \
    .csv("/path/to/directory")  # Equivalent to format("csv").load("/path/to/directory")
```

Anatomy of Streaming Query

```
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Split the lines into words
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)

# Generate running word count
wordCounts = words.groupBy("word").count()
```

Source

Specify where to read data from

Built-in support for Files / Kafka

Can include multiple sources of different types using `join()` / `union()`

returns a Spark DataFrame

(common API for batch & streaming data)

Anatomy of Streaming Query

```
# Start running the query that prints the running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

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

Scala Java **Python** R

```
# TERMINAL 1:
# Running Netca
t

$ nc -lk 9999
apache spark
apache hadoop



...
```

```
# TERMINAL 2: RUNNING structured_network_wordcount.py

$ ./bin/spark-submit examples/src/main/python/sql/streaming/structured_network_wordcount.py localhost
 9999

-------------------------------------------
Batch: 0
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    1|
| spark|    1|
+------+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    2|
| spark|    1|
|hadoop|    1|
+------+-----+
```

Anatomy of Streaming Query

```
spark.readStream.format("kafka")
.option("kafka.boostrap.servers",...)
.option("subscribe", "topic")
.load()
.selectExpr("cast (value as string) as json")
.select(from_json("json", schema).as("data"))
```

Transformations

Cast bytes from Kafka records to a string, parse it as a json, and generate nested columns

100s of built-in, optimized SQL functions like `from_json`

user-defined functions, lambdas, function literals with map, flatMap…

Basic Operations – Selection, Aggregation…

```
# streaming DataFrame with IOT device data with schema { device: string, deviceType: string, signal: double, time: DateType }
df = ...
# Select the devices which have signal more than 10
df.select("device").where("signal > 10")

# Running count of the number of updates for each device type
df.groupBy("deviceType").count()
```

Transformations

100s of built-in, optimized SQL functions like `from_json`

user-defined functions, lambdas, function literals with map, flatMap…

Output Sinks

Output Sinks

There are a few types of built-in output sinks.

- **File sink** - Stores the output to a directory.

```
writeStream
    .format("parquet")        // can be "orc", "json", "csv", etc.
    .option("path", "path/to/destination/dir")
    .start()
```

Sink

Write transformed output to external storage systems

Built-in support for Files / Kafka

Use `foreach` to execute arbitrary code with the output data

Some sinks are transactional and exactly once (e.g. files)

Output Sinks

- **Console sink (for debugging) -** Prints the output to the console/stdout every time there is a trigger. Both, Append and Complete output modes, are supported. This should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory after every trigger.

```
writeStream
    .format("console")
    .start()
```

Output Sinks

- **Memory sink (for debugging)** - The output is stored in memory as an in-memory table. Both, Append and Complete output modes, are supported. This should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory. Hence, use it with caution.

```
writeStream
    .format("memory")
    .queryName("tableName")
    .start()
```

Recovering from failures…checkpointing

```
aggDF \
    .writeStream \
    .outputMode("complete") \
    .option("checkpointLocation",
"path/to/HDFS/dir") \
    .format("memory") \
    .start()
```

Processing Details

Trigger: when to process data

-Fixed interval micro-batches

-As fast as possible micro-batches

-Continuously (new in Spark 2.3)

Checkpoint location: for tracking the progress of the query

Fault-tolerance with Checkpointing

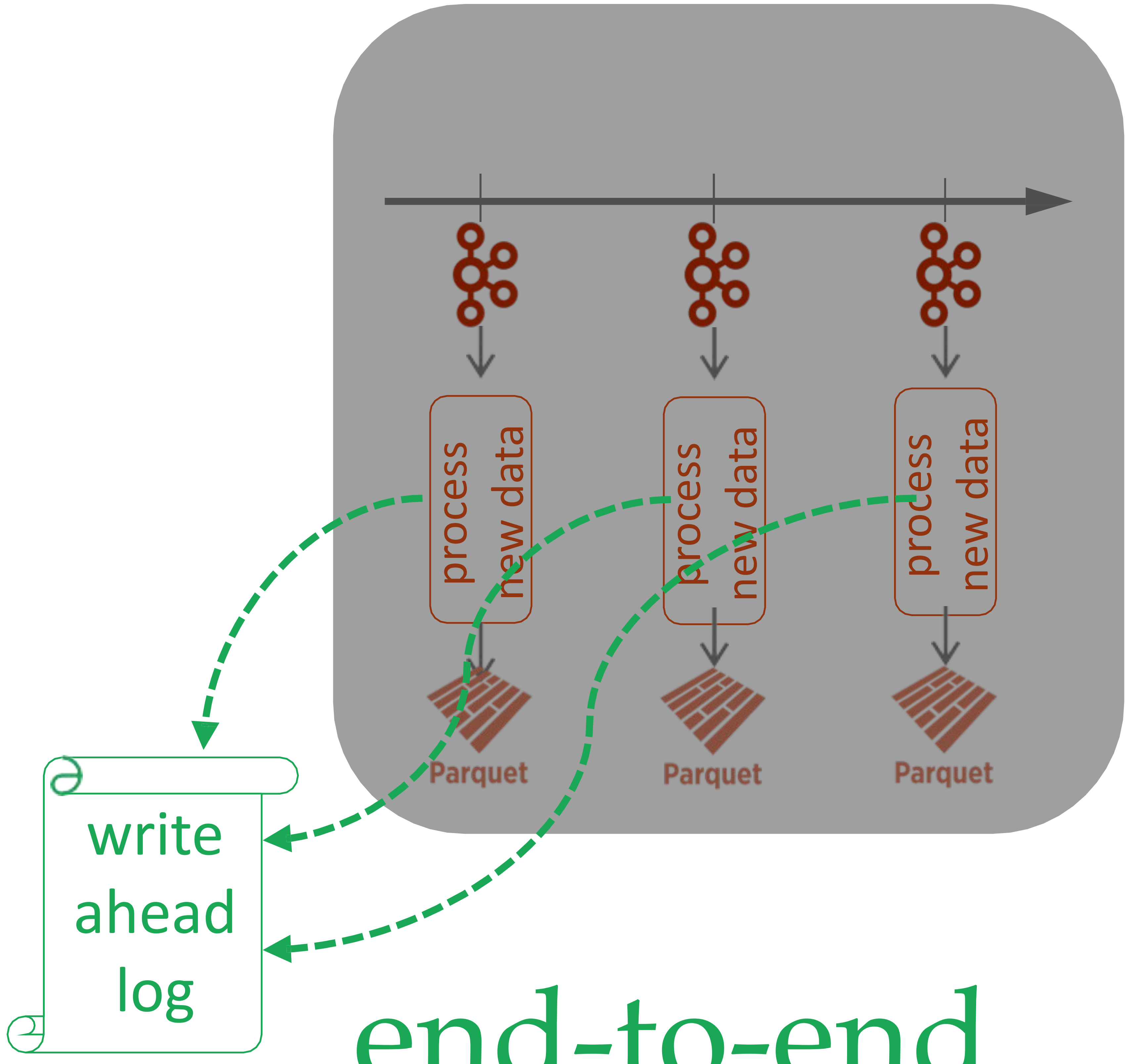
Checkpointing

Saves processed offset info to stable storage

Saved as JSON for forward-compatibility

Allows recovery from any failure

Can resume after limited changes to your streaming transformations (e.g. adding new filters to drop corrupted data, etc.)



end-to-end exactly-once guarantees

Thank You