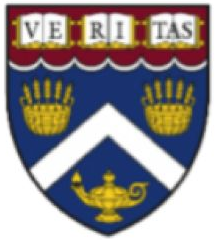
CSCI E-88 Principles Of Big Data Processing

Harvard University Extension, Fall 2019

Marina Popova

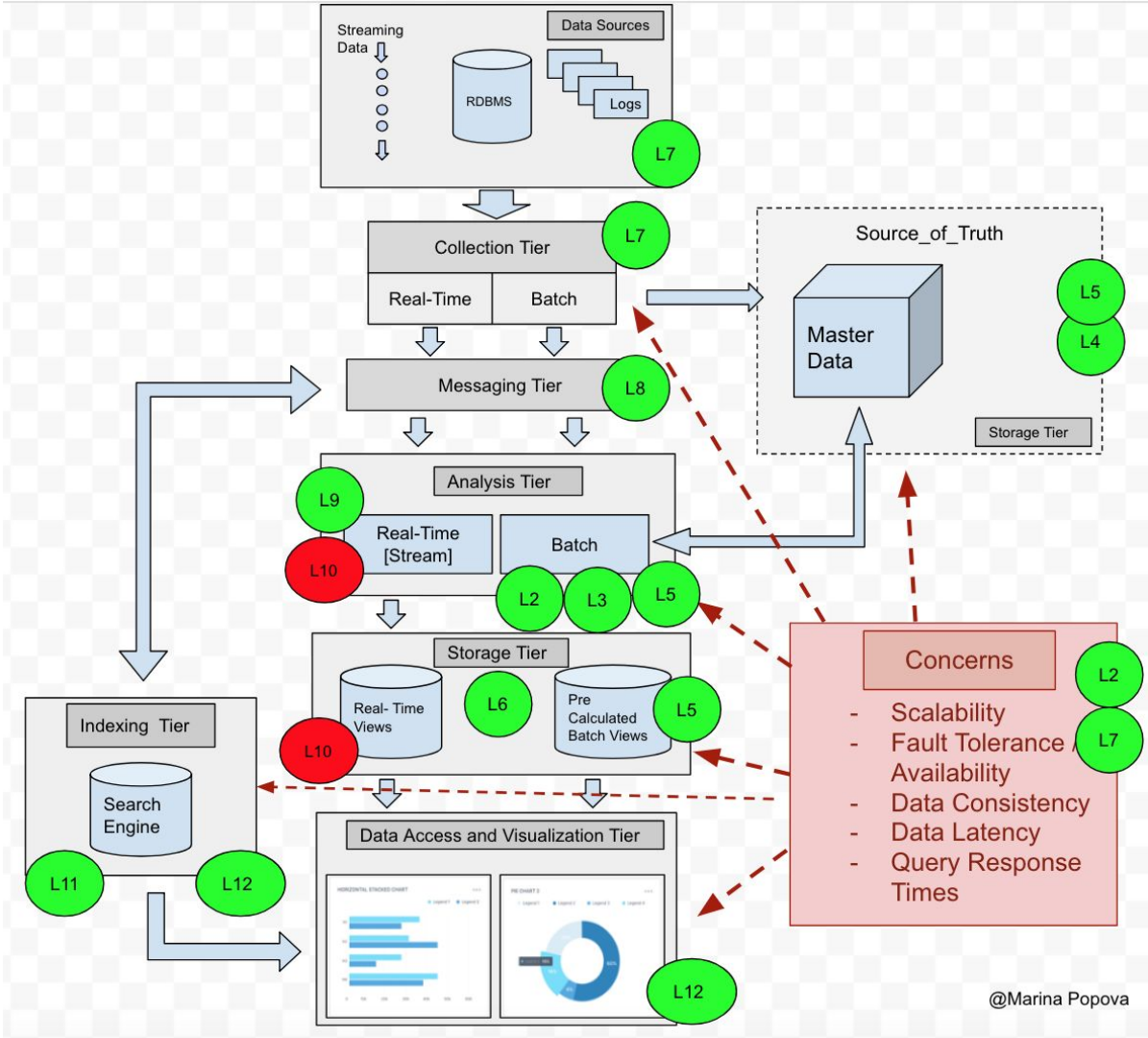


Lecture 10 Stream Processing and State

Agenda

- Mid-Term Quiz results - review in the Lab
- Streaming processing - out-of-order use case
- Other options for Stream Processing - Kafka Streams and KSQL
- State management as a Generic Problem
- Example architectures

Where Are We?


Streaming Data
Data Sources
RDBMS
Logs
L7
Collection Tier
L7
Real-Time
Batch
Source_of_Truth
Master Data
L5
L4
Storage Tier
Messaging Tier
L8
Analysis Tier
L9
L10
Real-Time [Stream]
Batch
L2
L3
L5
Storage Tier
Real- Time Views
L6
Pre Calculated Batch Views
L5
L10
Indexing Tier
Search Engine
L11
L12
Data Access and Visualization Tier
L12
Concerns
- Scalability
- Fault Tolerance / Availability
- Data Consistency
- Data Latency
- Query Response Times
L2
L7
@Marina Popova

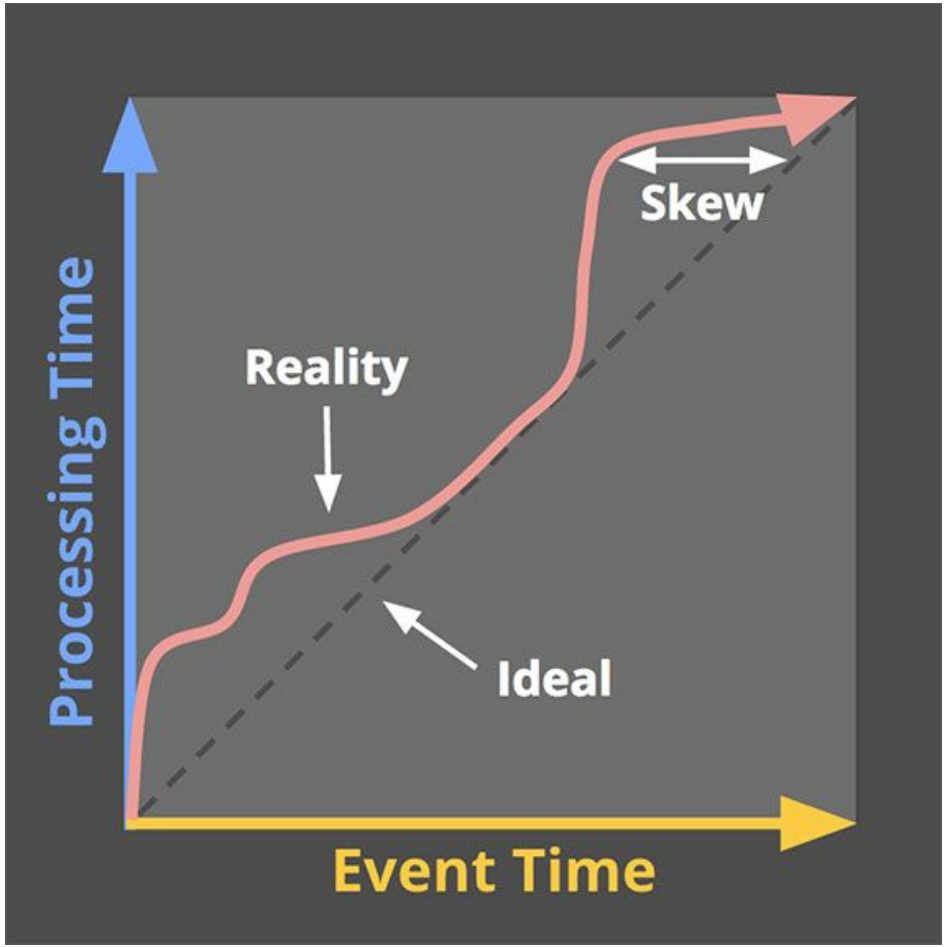
Stream Processing

- Event time: when an event took place
- Stream time: when the event entered the streaming system

"time skew" - difference between event and stream time

Ingestion time: a hybrid of processing and event time. It assigns wall clock timestamps to records as soon as they arrive in the system (at the source) and continues processing with event time semantics based on the attached timestamps

Ref: https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101



Stream Processing

Out-of-order even handling

With event-time based stream processing - events can arrive out of order, for different reasons
We do want to still process them - but up to what point?

Cannot keep all windows forever!

At some point, a window has to be considered "done" and garbage collected.
This is handled by a mechanism called **watermarks**.

Watermarks define completeness of events relative to the event-processing time

A watermark=X specifies that we assume that all events before X have been observed.

Stream Processing

How are watermarks defined?

- Precise watermarks
  - We know exactly how old the events can be
  - Set as a configuration parameter
- Heuristic watermarks
  - Probabilistic determination of completeness
  - Estimates based on all available information (data size, event number, etc.)

Stream Processing - Window Actions

Triggers: when should window content be evaluated by the windowing function ?

Regular windows:

- Processing-time - based (tumbling and sliding windows)
- Event count OR processing-time - based

"Old" windows - within the watermark but older than processing time window:

- On each event arrival
- Event-processing time-based - when watermarks end of window is reached
- Periodically on a predefined schedule
- Many other possibilities

"Old" windows - outside of the watermark - what action to take?

- Discard
- Re-evaluate corresponding window from historical data
- Store in a separate storage - for later re-integrate into full results

Spark Structured Streaming

Spark Structured Streaming:

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

"Event-time" based aggregations can be done in Spark Structured Streaming using the special grouping with window() function:

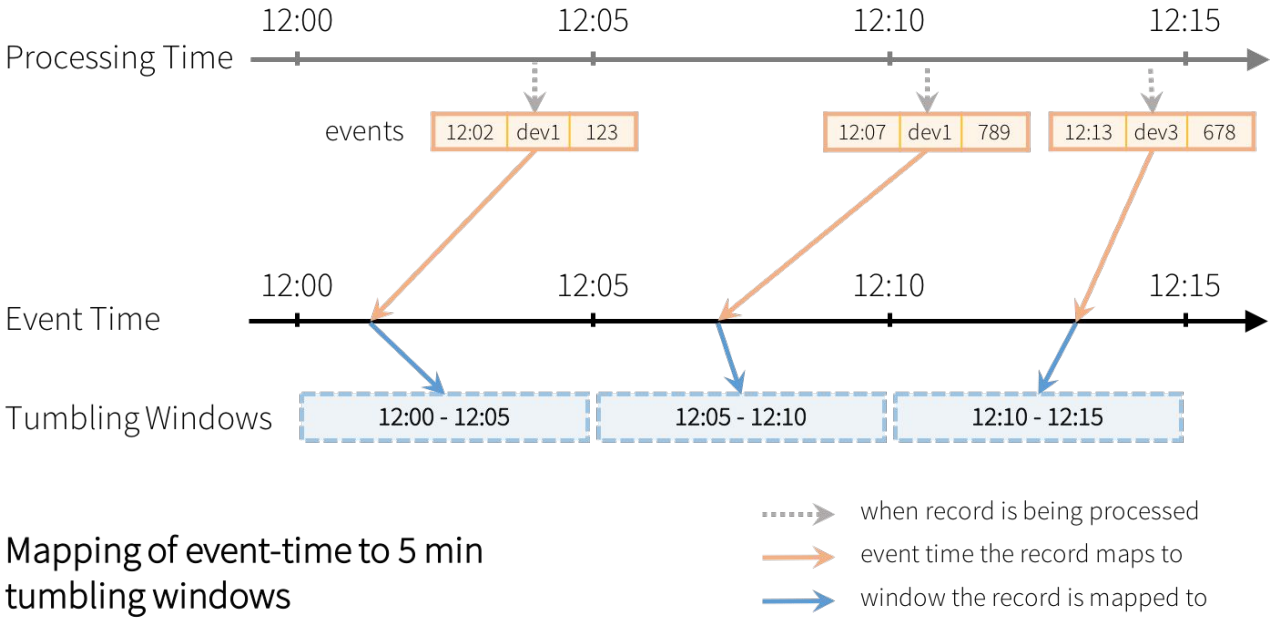
Event format:
{"eventTime":"10:25 11/01/2017",
"deviceID":"abcd111"
}

every record is going to be assigned to a 5 minute tumbling window

```
from pyspark.sql.functions import *

windowedAvgSignalDF = \
  eventsDF \
    .groupBy(window("eventTime", "5 minute")) \
    .count()
```

Stream Structured Streaming

Each window is a group for which running counts are calculated:



Mapping of event-time to 5 min tumbling windows

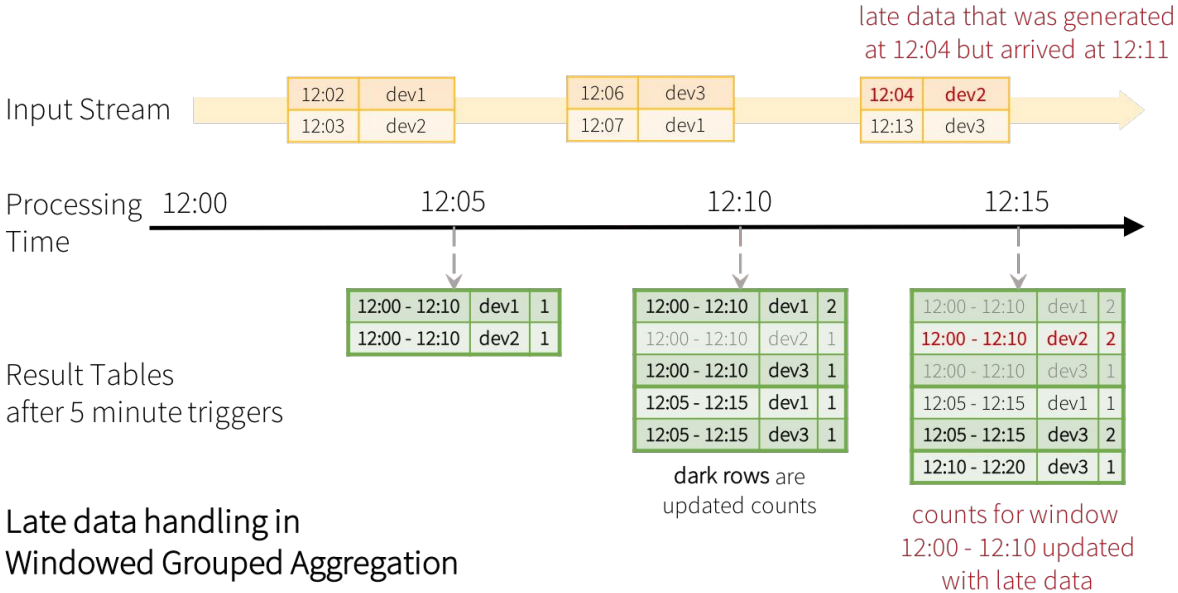
@Marina Popova

Out-of-order events

- can also use sliding windows

```
windowedCountsDF = \
  eventsDF \
    .groupBy(
      "deviceId",
      window("eventTime", "10 minutes", "5 minutes")) \
    .count()
```

- the **late event** would just update older window groups instead of the latest ones



Late data handling in Windowed Grouped Aggregation

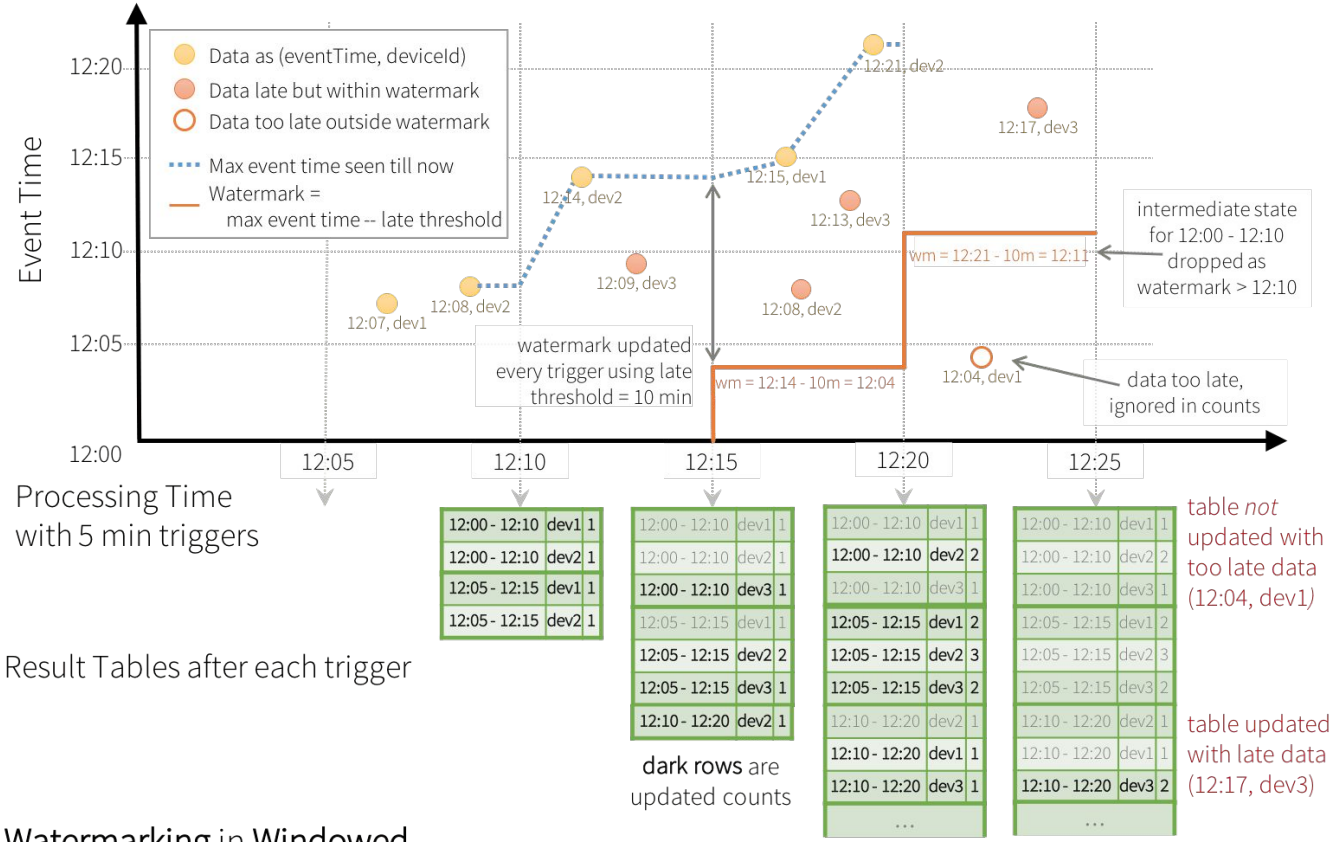
@Marina Popova

Watermarking in Spark

- *Watermarking*: enables automatic dropping of old state data
- is a moving threshold in event-time that trails behind the maximum event-time seen by the query in the processed data. The trailing gap defines how long we will wait for late data to arrive

Example with watermarking
Spark SQL will automatically keep track of the maximum observed value of the eventTime column, update the watermark and clear old state.

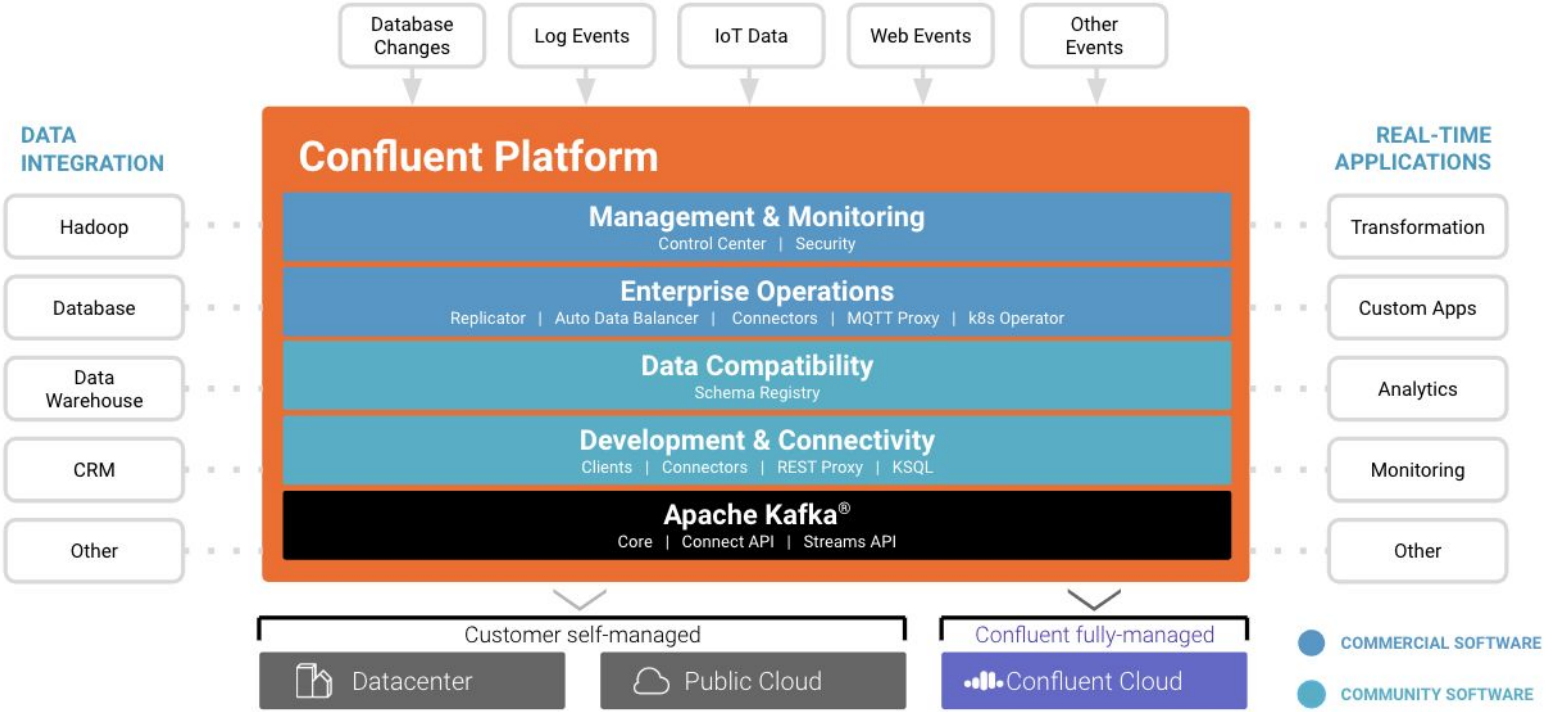
```
windowedCountsDF = \
  eventsDF \
    .withWatermark("eventTime", "10 minutes") \
    .groupBy(
      "deviceId",
      window("eventTime", "10 minutes", "5 minutes")) \
    .count()
```



Watermarking in **Windowed Grouped Aggregation**

@Marina Popova

Kafka - again! - as part of Confluent Platform



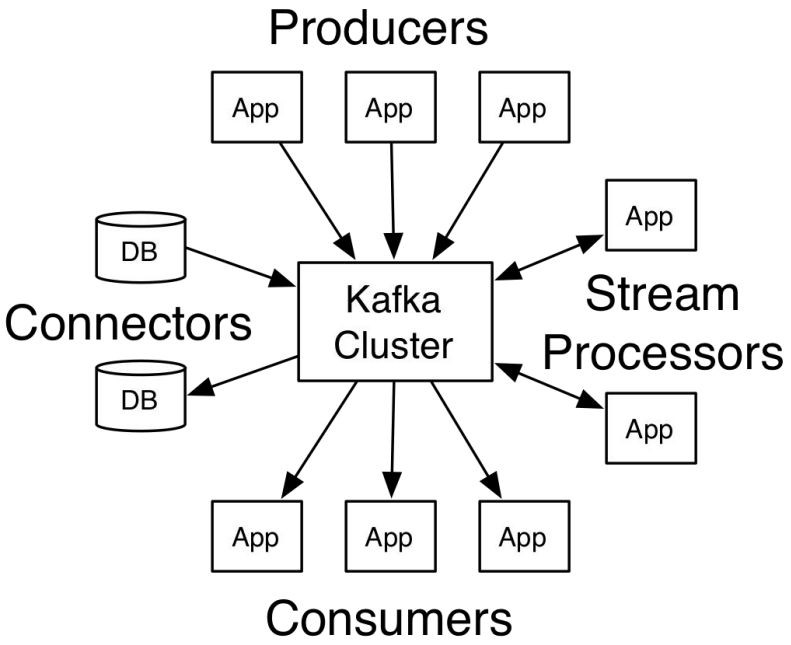
Ref: https://docs.confluent.io/current/platform.html

@Marina Popova

Kafka - again!

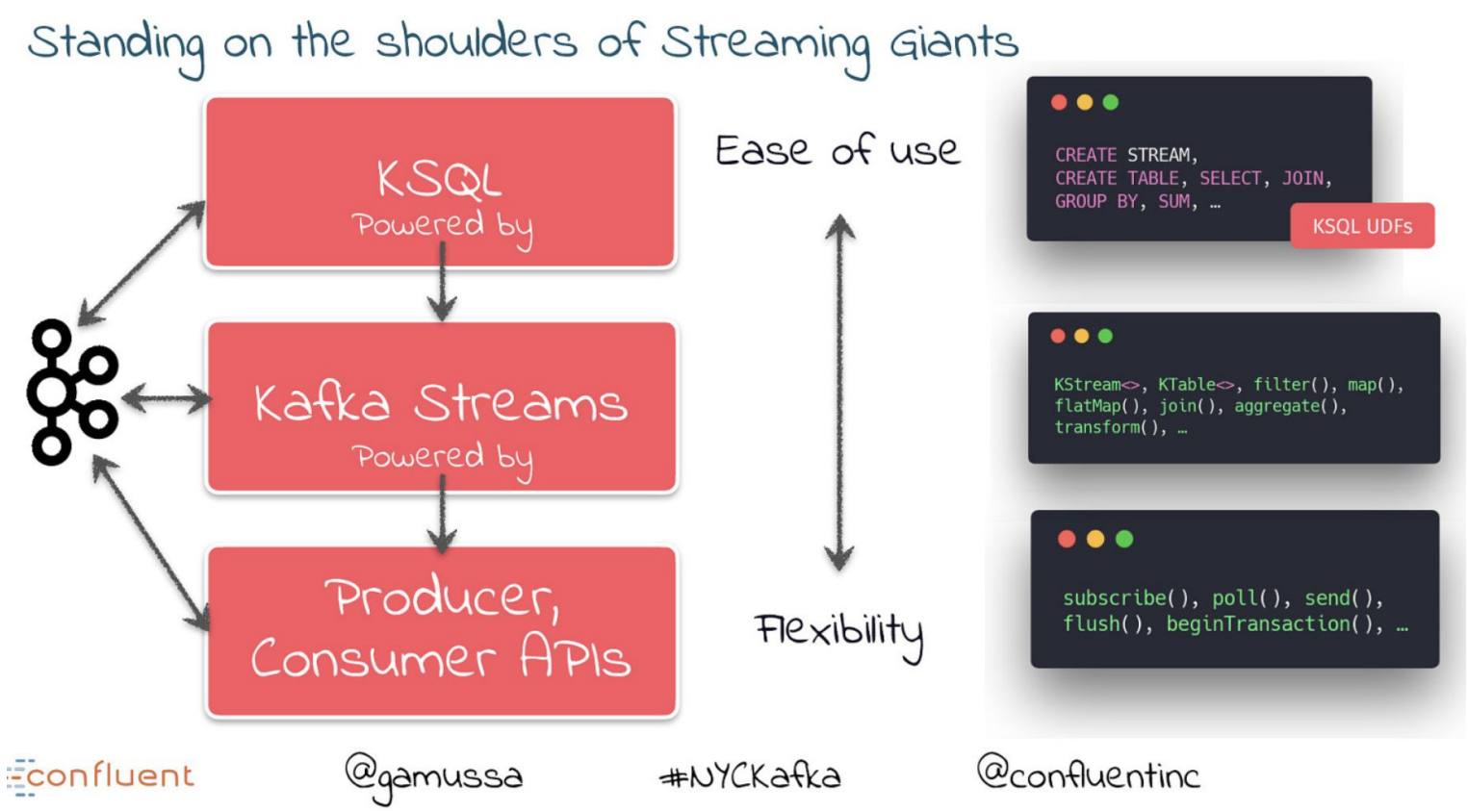
Kafka has four core APIs:

- The Producer API allows an application to publish a stream of records to one or more Kafka topics.
- The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them.
- The Streams API allows an application to act as a *stream processor*, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.
- The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems.



Ref: https://kafka.apache.org/documentation/#introduction

Kafka -> Kafka Streams -> KSQL



https://softwaremill.com/kafka-streams-how-does-it-fit-stream-landscape/

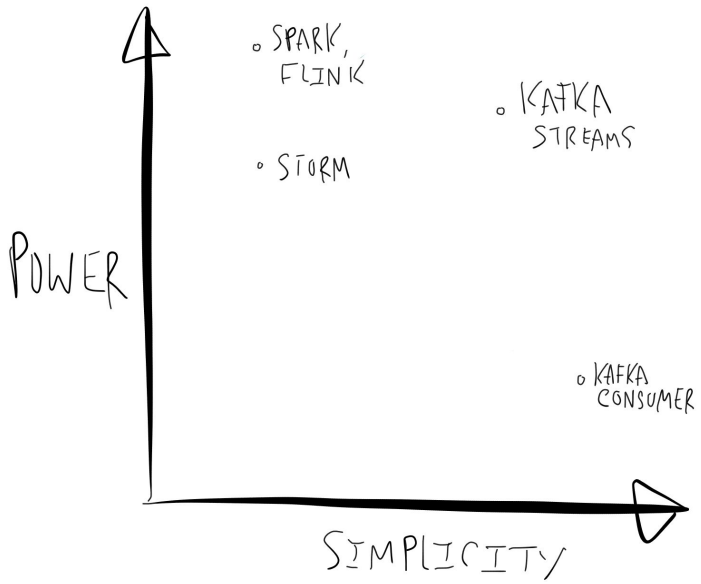
@Marina Popova

Kafka Streams

The goal is to simplify stream processing:

1. Kafka Streams is a fully embedded library with no stream processing cluster—just Kafka and your application.
2. Fully integrating the idea of tables of state with streams of events

Kafka Streams simplifies application development by building on the Kafka producer and consumer libraries and leveraging the native capabilities of Kafka to offer data parallelism, distributed coordination, fault tolerance, and operational simplicity.



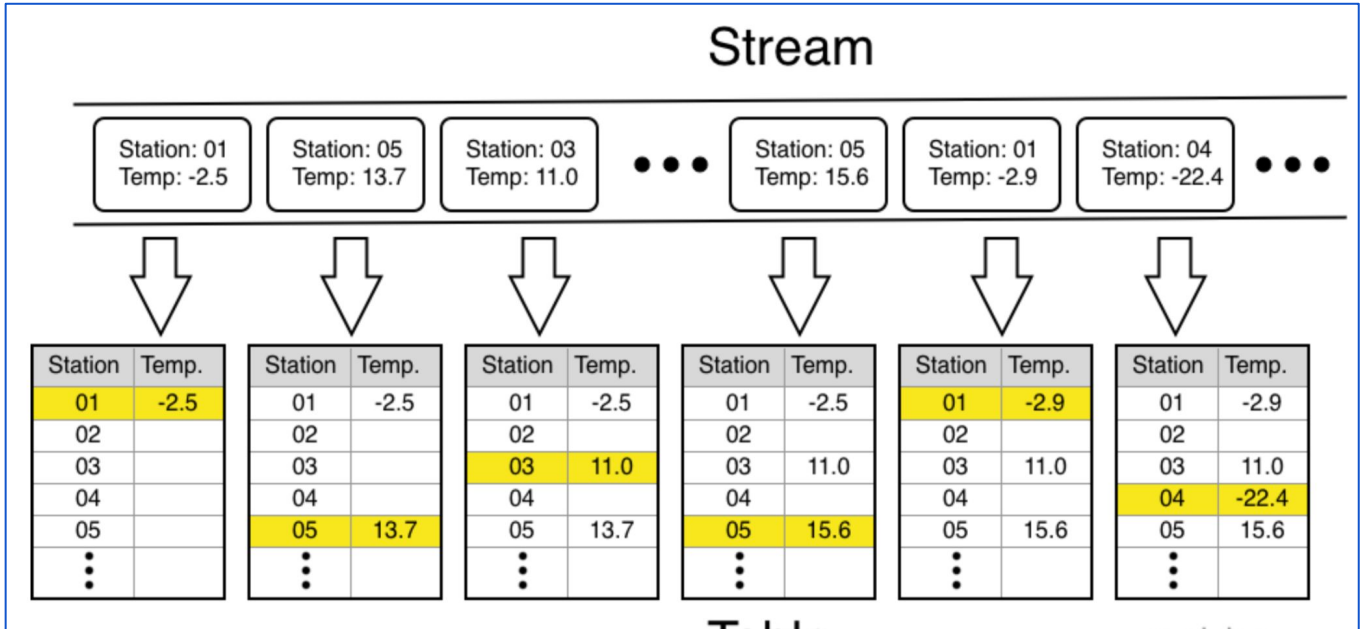
https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/

Kafka Streams Data Abstractions: KStream and KTable

Kafka Streams defined two basic abstractions: KStream and KTable.
The distinction comes from how the key-value pairs are interpreted.

KStream: each key-value is an independent piece of information

KTtable is a changelog. If the table contains a key-value pair for the same key twice, the latter overwrites the mapping.

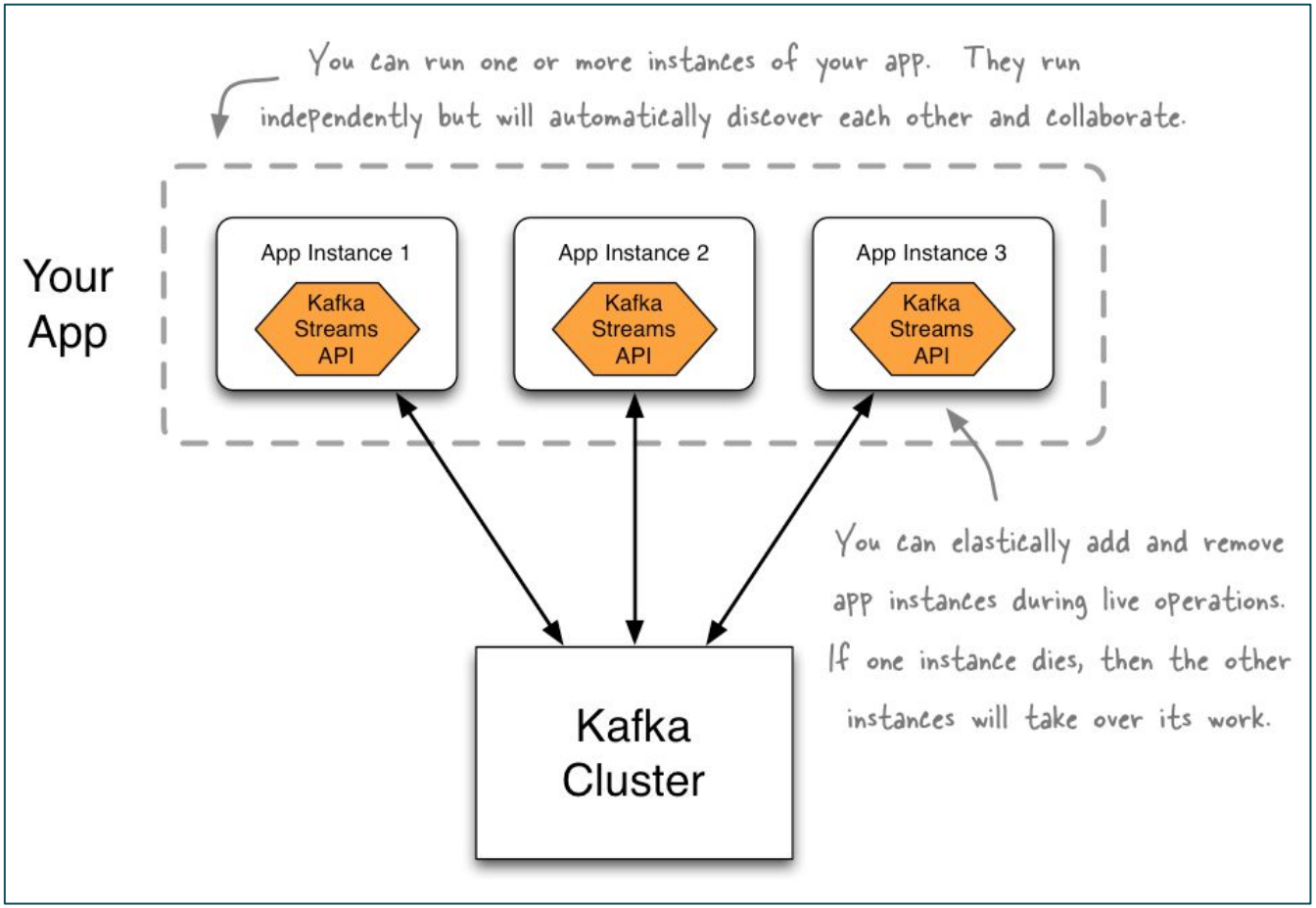


@Marina Popova

Kafka Streams - what does it look like?

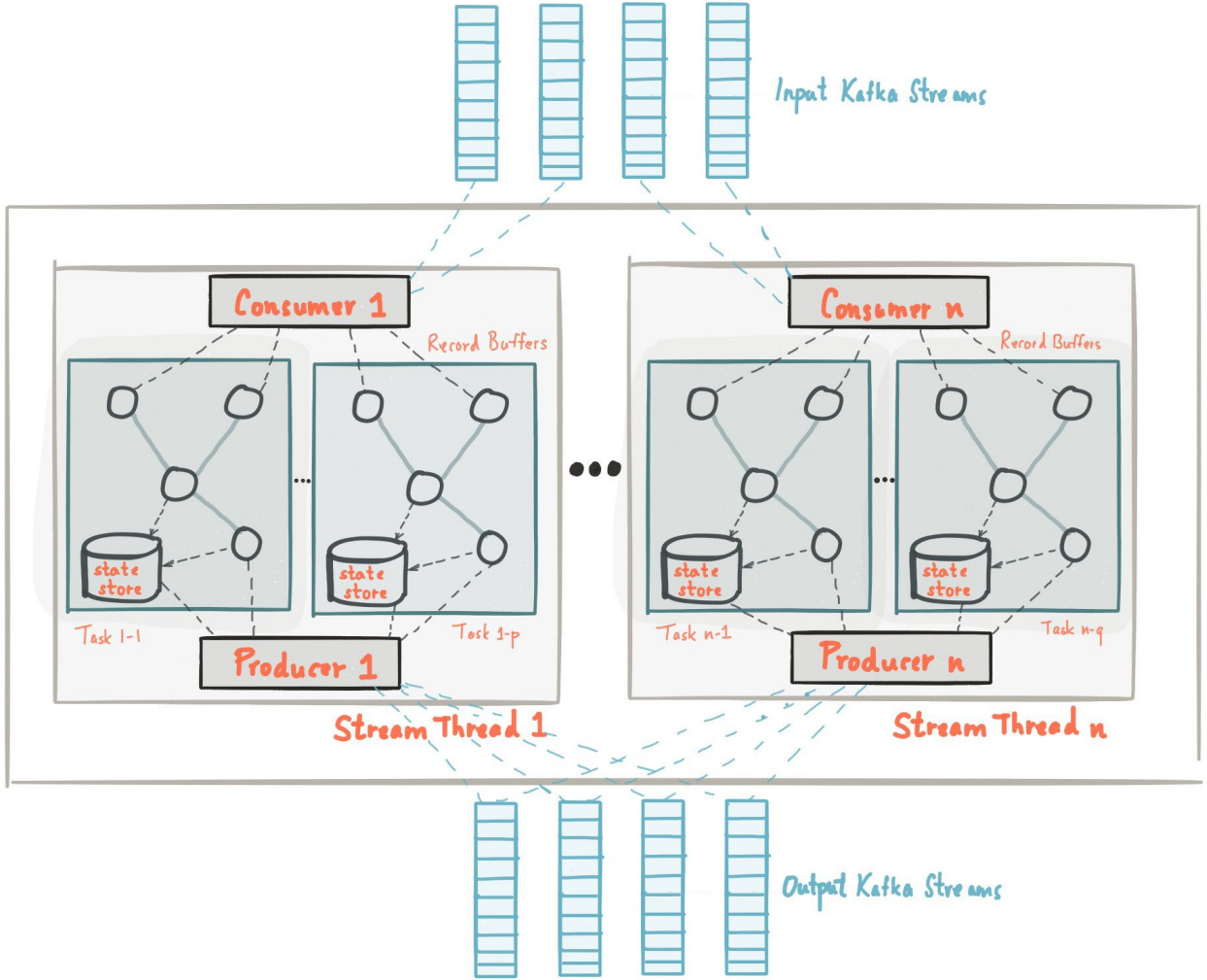
```
public class WordCountApplication {

    public static void main(final String[] args) throws Exception {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-application");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-broker1:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());

        StreamsBuilder builder = new StreamsBuilder();
        KStream<String, String> textLines = builder.stream("TextLinesTopic");
        KTable<String, Long> wordCounts = textLines
            .flatMapValues(textLine -> Arrays.asList(textLine.toLowerCase().split("\\W+")))
            .groupBy((key, word) -> word)
            .count(Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>as("counts-store"));
        wordCounts.toStream().to("WordsWithCountsTopic", Produced.with(Serdes.String(), Serdes.Long()));

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();
    }

}
```

Kafka Streams Architecture



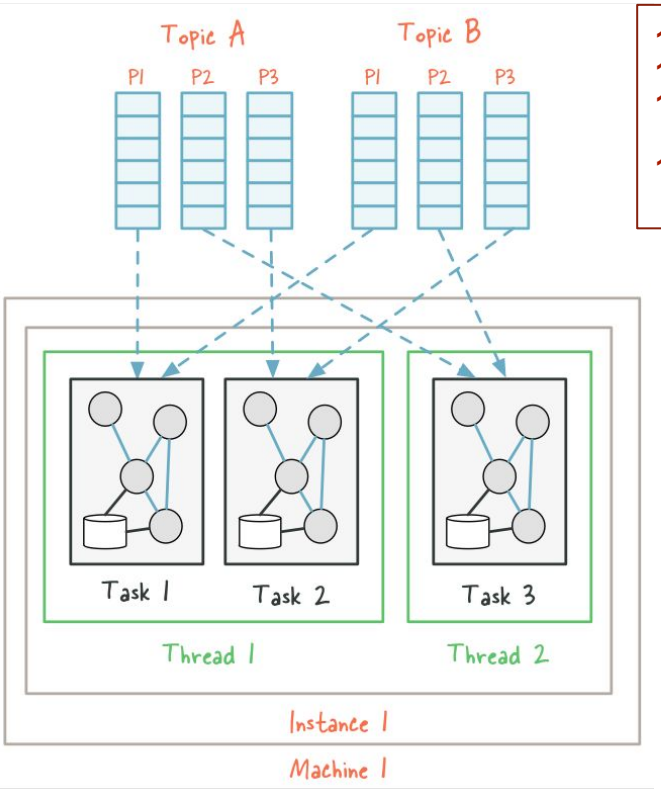
- Kafka Streams app is a normal Java application
- it just uses the Kafka Streams API (lib)
- can be packaged, deployed and monitored as any Java app

Kafka Streams Architecture



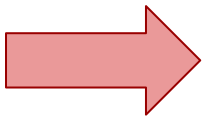
@Marina Popova

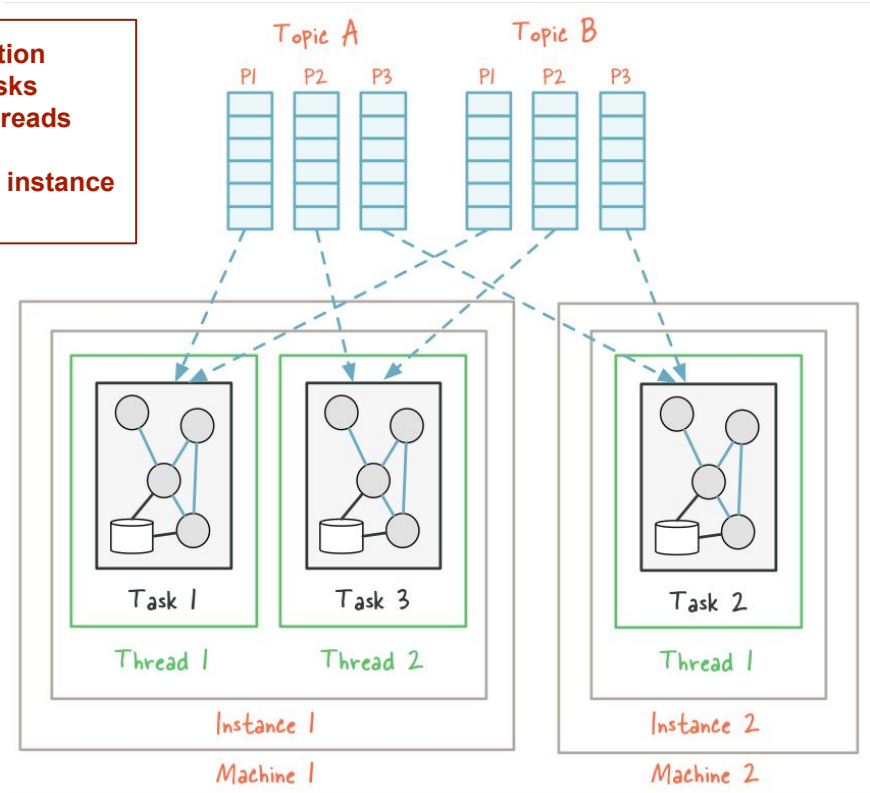
Kafka Streams/KSQL Parallelization



1 Task --> 1 partition
1 Thread --> 1-n Tasks
1 App Instance --> 1-n Threads

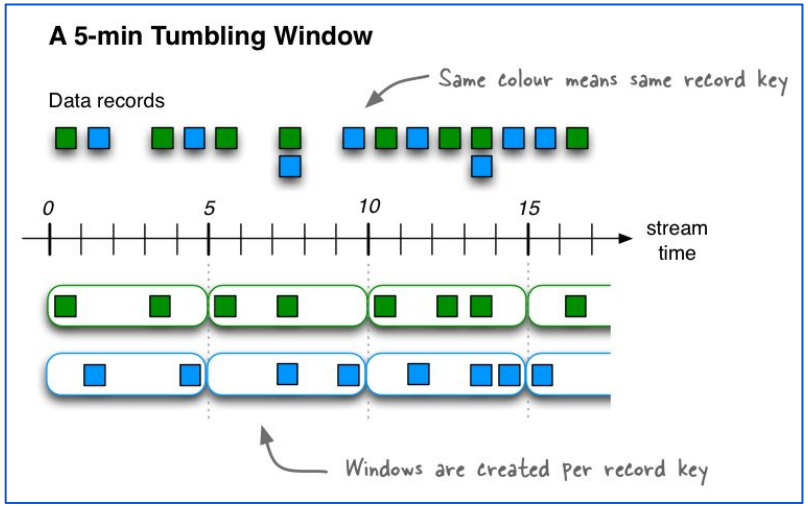
1 Thread <--> 1 RocksDB instance

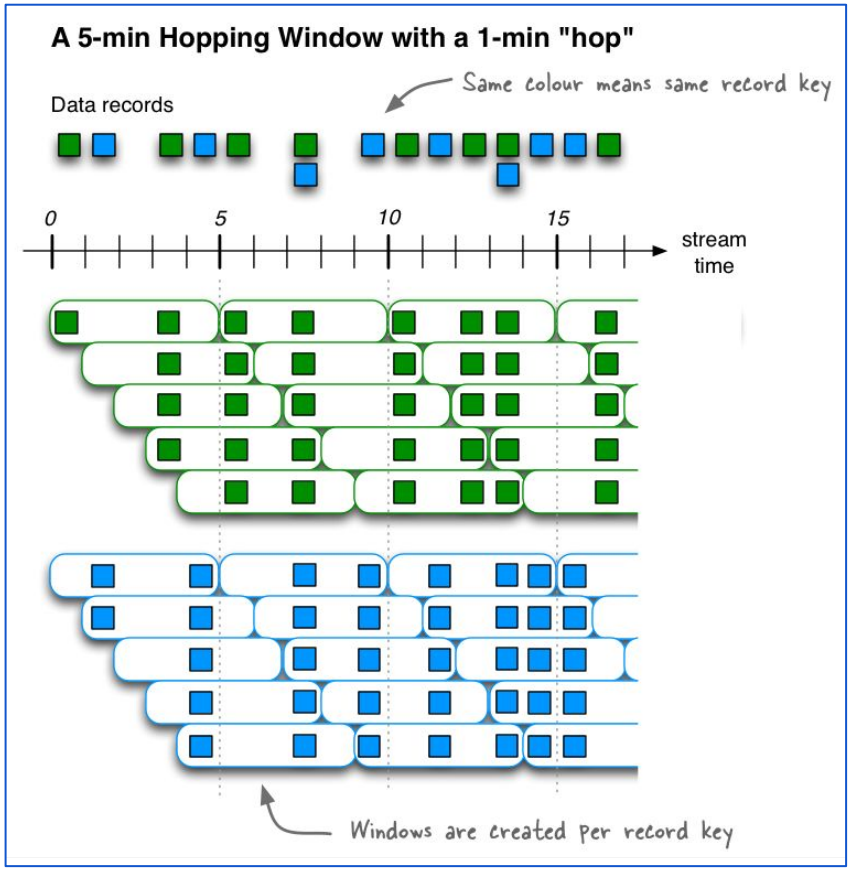




Kafka Streams: Window Operations

- applied to "group" operations such as aggregations and joins
- windows are tracked per record key !
- supported windows:
  - tumbling - non-overlapping
  - hopping - fixed duration, overlapping
  - sliding - fixed, overlapping, based on TS deltas
  - session
- out-of-order/watermark handling:
  - specify "retention" period via `Materialized.withRetention(...)`

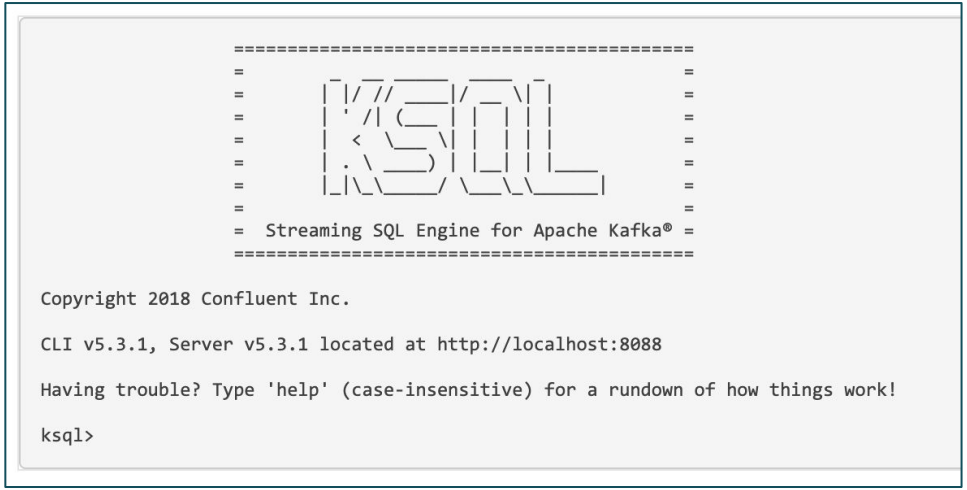




@Marina Popova

what is KSQL ?

- Streaming SQL engine for Kafka
- interactive SQL interface to Kafka streams
- no programming - SQL commands
- a layer on top of Kafka Streams, and , as such:
- scalable and fault-tolerant



The DDL statements include:

- CREATE STREAM
- CREATE TABLE
- DROP STREAM
- DROP TABLE
- CREATE STREAM AS SELECT (CSAS)
- CREATE TABLE AS SELECT (CTAS)

The DML statements include:

- SELECT
- INSERT INTO
- CREATE STREAM AS SELECT (CSAS)
- CREATE TABLE AS SELECT (CTAS)

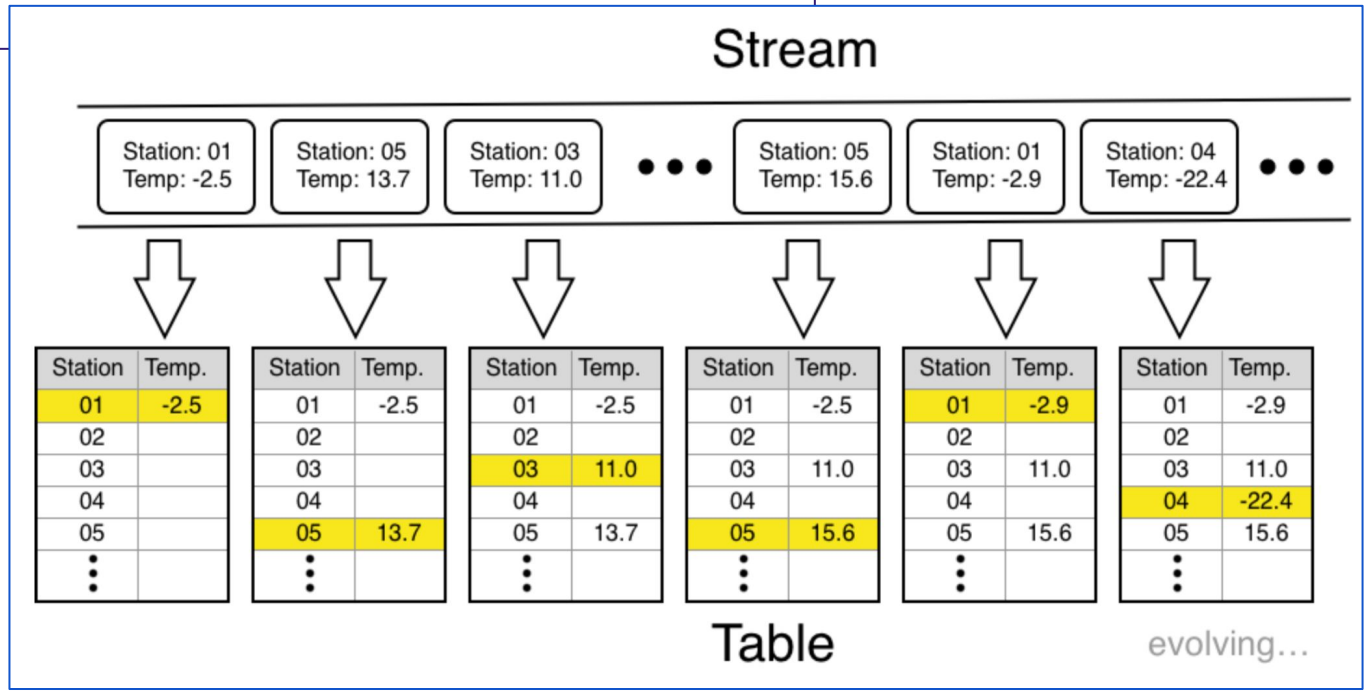
@Marina Popova

KSQL: Data Abstractions: STREAM and TABLE

KSQL reads Kafka topic data into a Stream or a Table:

a STREAM is an unbounded sequence of events (uses KStreams underneath)

a TABLE is a materialized view of events with only the latest values for each key (uses KTable underneath)



KSQL examples

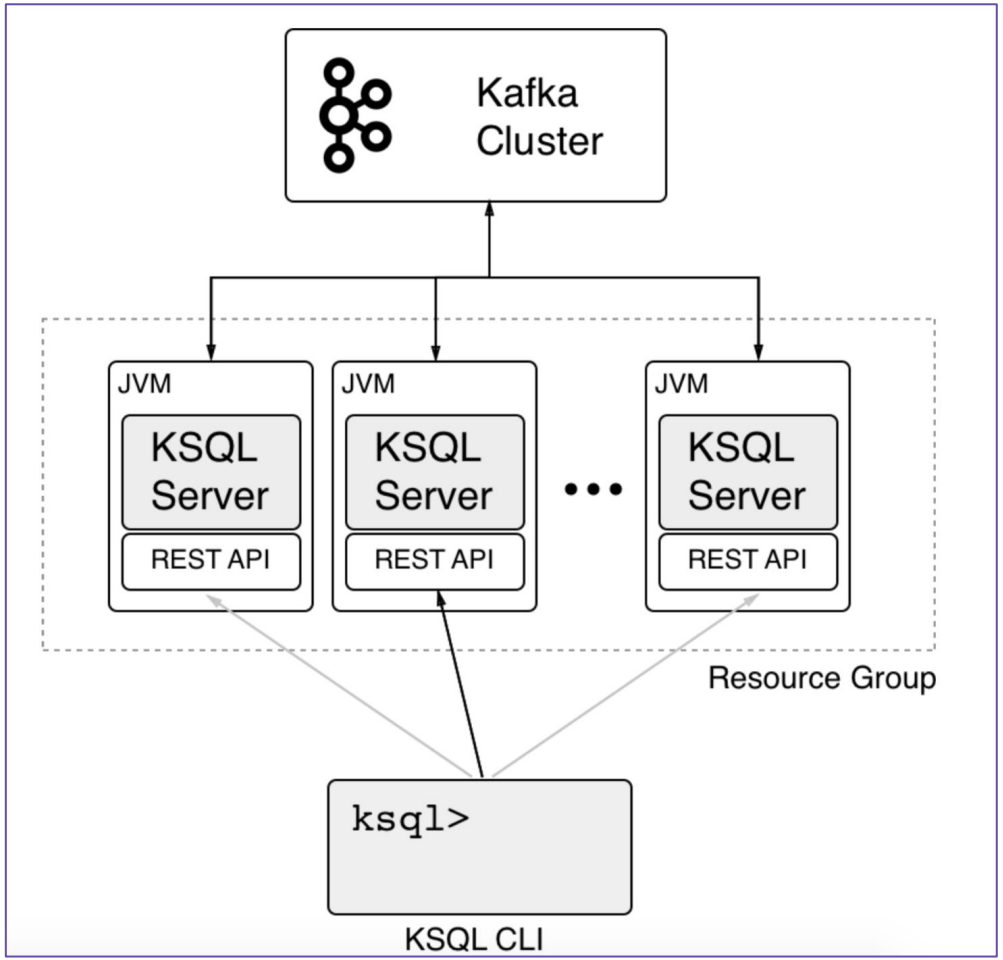
```
ksql> CREATE STREAM persons (firstName string, lastName string, birthDate string) WITH (kafka_topic='persons'
```

```
ksql> DESCRIBE persons;

Name                 : PERSONS
 Field     | Type
----------------------------------------
 ROWTIME   | BIGINT           (system)
 ROWKEY    | VARCHAR(STRING)  (system)
 FIRSTNAME | VARCHAR(STRING)
 LASTNAME  | VARCHAR(STRING)
 BIRTHDATE | VARCHAR(STRING)
----------------------------------------
For runtime statistics and query details r
```

```
ksql> SELECT * FROM persons;
1534874843511 | null | Trinity | Beatty | 1958-06-16T13:10:49.824+0000
1534874844526 | null | Danyka | Hodkiewicz | 1961-04-05T20:51:32.358+0000
1534874845529 | null | Florida | Gorczany | 1992-09-01T22:44:33.114+0000
...
```

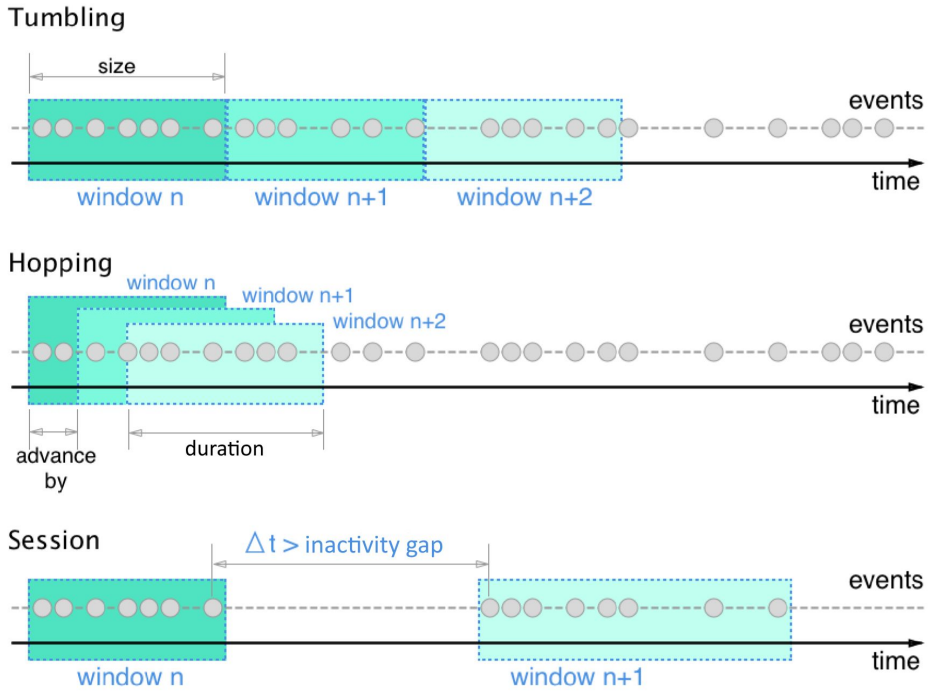
Ref: https://aseigneurin.github.io/2018/08/22/kafka-tutorial-10-ksql.html

KSQL Architecture



KSQL: Windowed Operations

Windowed Aggregation



@Marina Popova

Types of Operations: Streams and KSQL

Persistent vs Non-Persistent Operations

```
SELECT pageid FROM pageviews_original LIMIT 3;
```

Non-persistent: Interactive Queries

- KSQL: based on: SELECT ... FROM ... WHERE ...
- KStreams:

```
static void createToUpperCaseStream(final StreamsBuilder builder) {
    final KStream<String, String> textLines = builder.stream(inputTopic);
    final KStream<String, String> upperCaseValues = textLines
            .mapValues(value -> value.toUpperCase());
```

Persistent:

- KSQL: CREATE TABLE AS SELECT ....
- KSQL: CREATE STREAM AS SELECT ....
- Kstreams:
  - KStream.to(<new_topic>)
  - KStream.through(<new_topic>).map()...

```
CREATE STREAM pageviews_enriched AS
SELECT users_original.userid AS userid, pageid, regionid, gender
FROM pageviews_original
LEFT JOIN users_original
  ON pageviews_original.userid = users_original.userid;
```

```
static void createToUpperCaseStream(final StreamsBuilder builder) {
    final KStream<String, String> textLines = builder.stream(inputTopic);
    final KStream<String, String> upperCaseValues = textLines
            .mapValues(value -> value.toUpperCase());

    upperCaseValues.to(outputTopic);
}
```

Types of Operations: Streams and KSQL

Stateless vs Statefull

Stateless:

- do not require state for each record processing
- KStreams: mapValues(), peek(), filter()
- KSQL: SELECT ... WHERE

```
static void createToUpperCaseStream(final StreamsBuilder builder) {
    final KStream<String, String> textLines = builder.stream(inputTopic);
    final KStream<String, String> upperCaseValues = textLines
            .mapValues(value -> value.toUpperCase());

    upperCaseValues.to(outputTopic);
}
```

Statefull: aggregations, window operations, joins, map() with different key :

- require a state store per thread!
- groupByKey()
- windowBy()
- count()
- map()

```
static void createAnomalyDetectionStream(final StreamsBuilder builder) {
    // detect users as anomalous if they have appeared more
    // than 3 times in the click stream during one minute
    final KStream<String, String> views = builder.stream("UserClicks");
    final KTable<Windowed<String>, Long> anomalousUsers = views
      // map the user name as key, because the subsequent
      // counting is performed based on the key
      .map((ignoredKey, username) -> new KeyValue<>(username, username))
      // count users, using one-minute tumbling windows;
      .groupByKey()
      .windowedBy(TimeWindows.of(Duration.ofMinutes(1)))
      .count()
      // get users whose one-minute count is >= 3
      .filter((windowedUserId, count) -> count >= 3);
}
```

Kafka Streams/KSQL: Stateful Operations

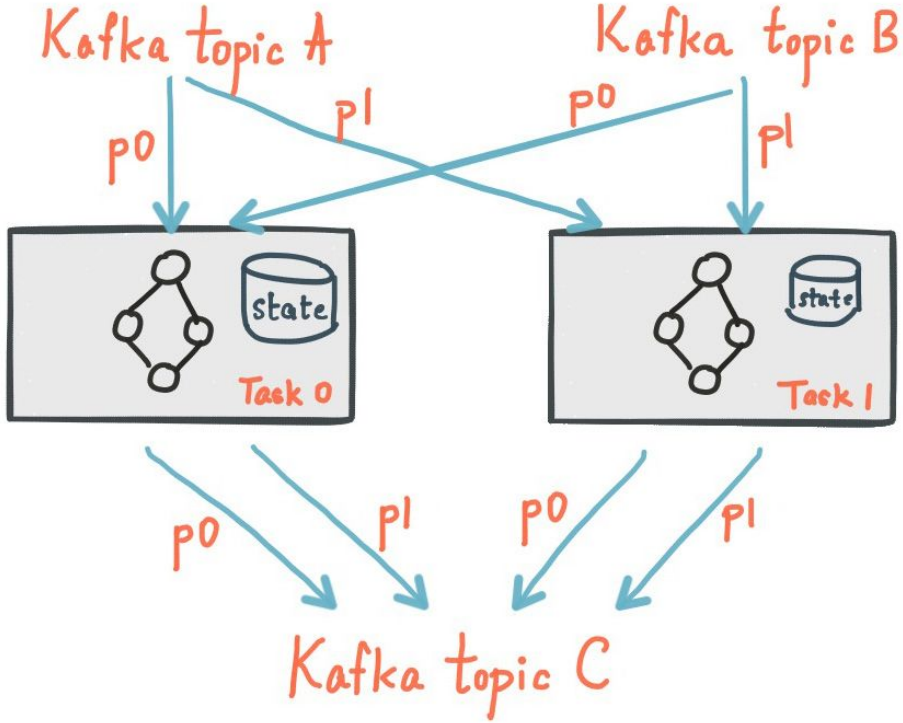
For stateful operations Kafka Streams provides **state stores**, which can be used by stream processing applications to store and query data.

They are required and used when implementing any of the following stateful operations:

- Joins
- Aggregations
- Window operations

State stores are fully fault-tolerance and automatically recoverable by the Kafka Streams

The following diagram shows two stream tasks with their dedicated local state stores.

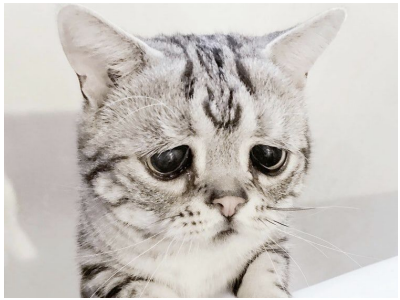


Stateful Operations and Re-partitioning

Stateful operations may require re-partitioning!

If input stream keys are the same as the operation keys - operation can be performed using existing partitions!

If input stream keys are not the same as the operation keys: an intermediate topic is created with all messages re-sent with the new keys!

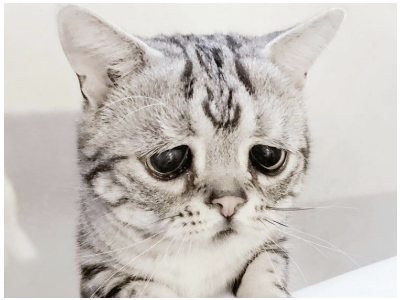


Ref: https://kafka.apache.org/20/documentation/streams/developer-guide/dsl-api.html#stateless-transformations

Stateful Operations and Re-partitioning

There are optimized operations to avoid unnecessary re-partitioning

Key-Changing Operation	Value-Only Operation
map	mapValues
flatMap	flatMapValues
transform	transformValues



How does it work??

Ref: https://www.confluent.io/blog/optimizing-kafka-streams-applications

@Marina Popova

KSQL: State Management

https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Internal+Data+Management

All stateful aggregate operations like SUM(), COUNT(), TOPK(), JOIN WITH, TOPKDISTINCT require internal state

KSQL and Kafka Streams manage this by using a **state store**

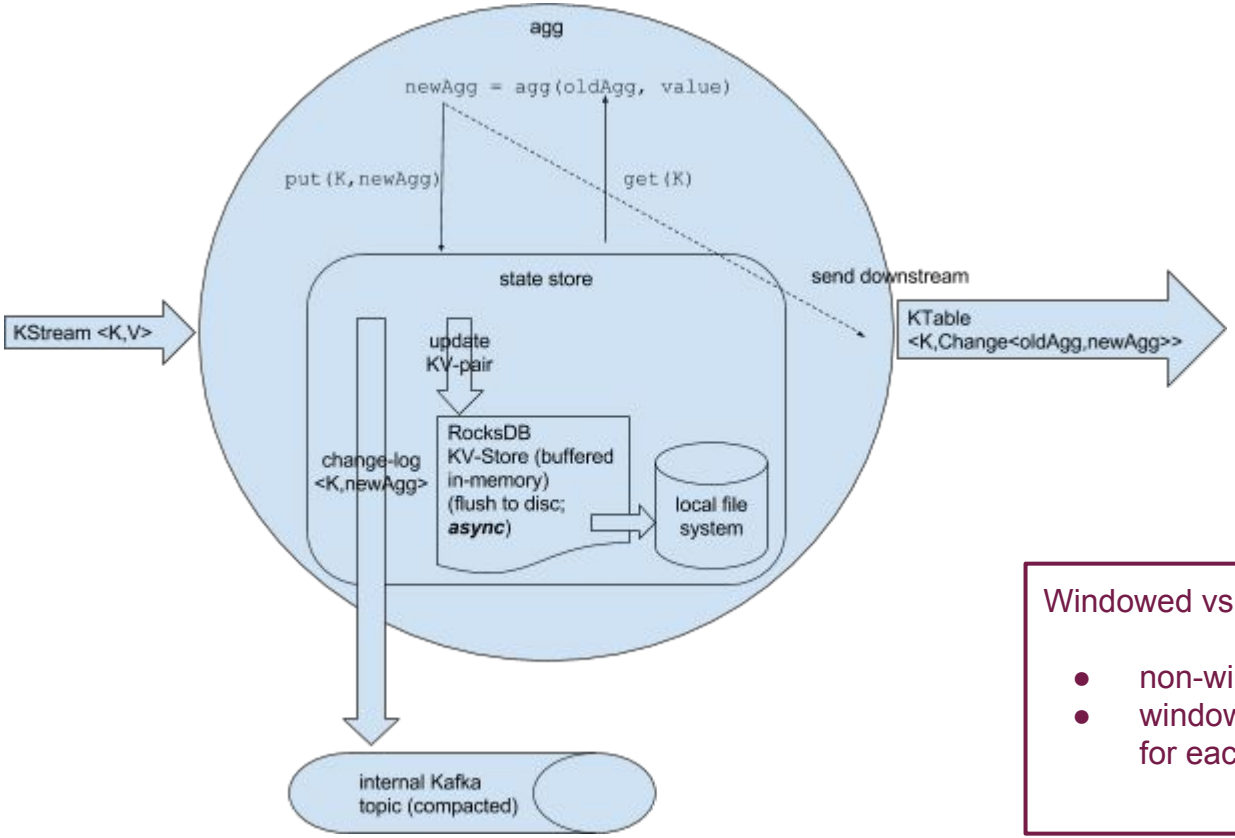
A state store can be ephemeral (lost on failure) or fault-tolerant (restored after the failure).

The default implementation used by Kafka Streams DSL is a fault-tolerant state store using:

1. **an internally created and compacted changelog topic (for fault-tolerance) and**
2. **one (or multiple) RocksDB instances (for cached key-value lookups)**

The changelog topic is the source of truth for the state (= the log of the state), while RocksDB is used as (non-fault tolerant) cache.

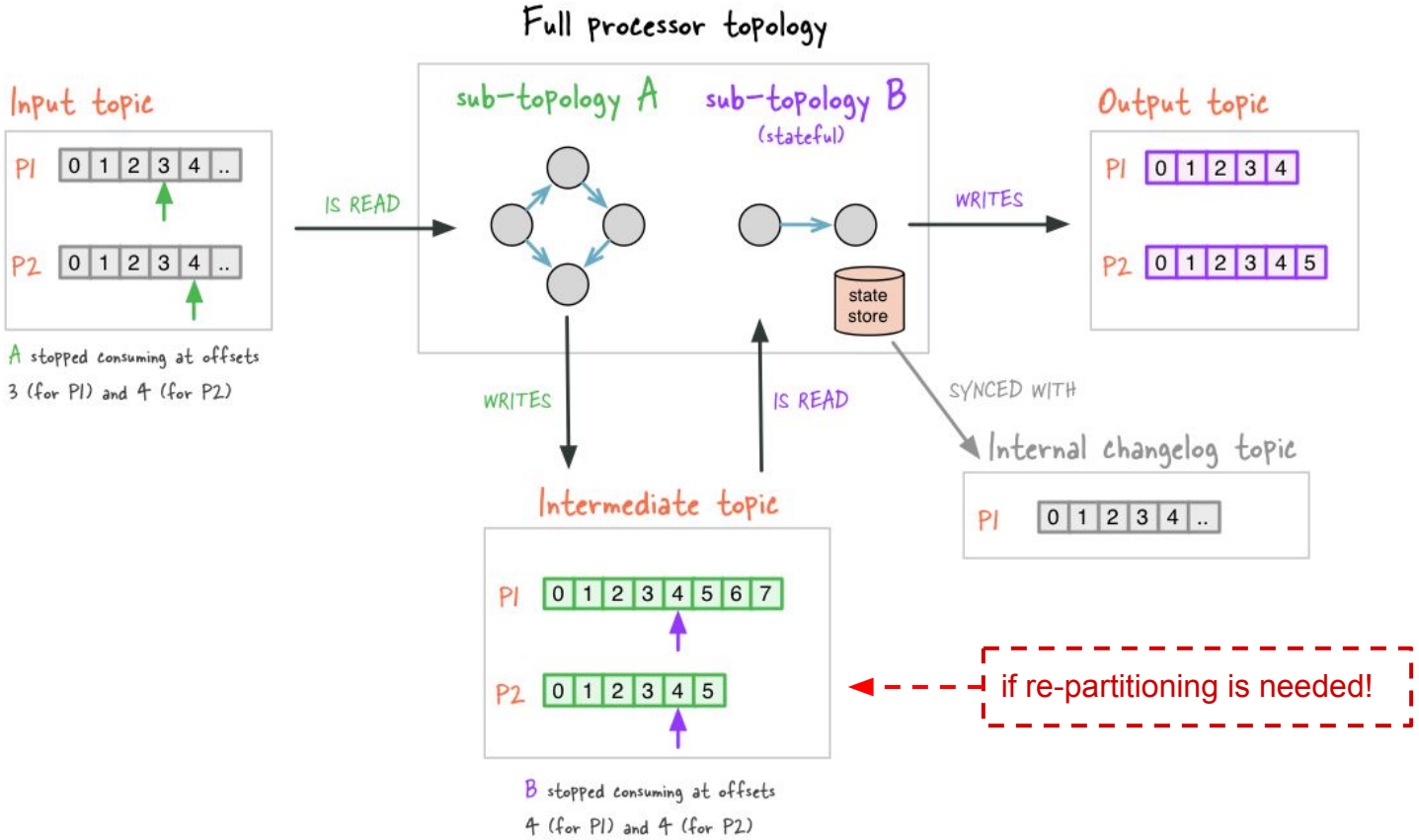
KStreams and KSQL: State Management



Windowed vs non-windowed state:

- non-windowed aggregation key: K
- window aggregation key: <K,W> - i.e., for each window a new key is used

KSQL: State management



@Marina Popova

KSQL - Under the Hood: Operations and Resources

Operation / Query type	Kafka resource usage	notes
INFO: EXPLAIN query … SHOW streams ...	no interaction with Kafka - read meta info from KSQL servers	
Non-persistent DDL: CREATE STREAM WITH <topic> ... CREATE TABLE WITH <topic> ...	write metadata to Kafka - command topic	
Non-persistent **stateless** queries: SELECT … FROM STREAM/TABLE … WHERE ...	read from Kafka source topics	
Non-persistent **stateful** queries: SELECT … COUNT() FROM … JOIN …	read and write to Kafka: ● repartition topic [if required] ● changelog topic ● RocksDB state (in-memory and/or disc) **All state/topic data is deleted once query is stopped (Cntrl+C)**	Resource usage varies by query type: ● Joins: 2x resources of stateless queries ● Aggregations: SUM, COUNT, TOPK ... : 4x resources of stateless queries ● **non-windowed**: memory: ~ number of distinct keys - may spill to disk ● **windowed**: same as above , but PER window --> will require (4*N)x resources of stateless queries, where N is the number of windows @Marina Popova
Persistent DDL --> queries: CREATE TABLE AS SELECT ... CREATE STREAM AS SELECT ...	read and write to Kafka: ● output topic ● repartition topic [if required] ● changelog topic ● RocksDB state (in-memory and/or disc)	

Generalization of the Problem ...

What are we trying to do, really?

- ingest/stream large quantities of data
- do analytics on this data - including stateful operations like aggregations and AdHoc queries by arbitrary keys
- hopefully, store this data for historical queries
- hopefully, use both historical and real-time data for the same analytics
- do all this using Distributed Systems!

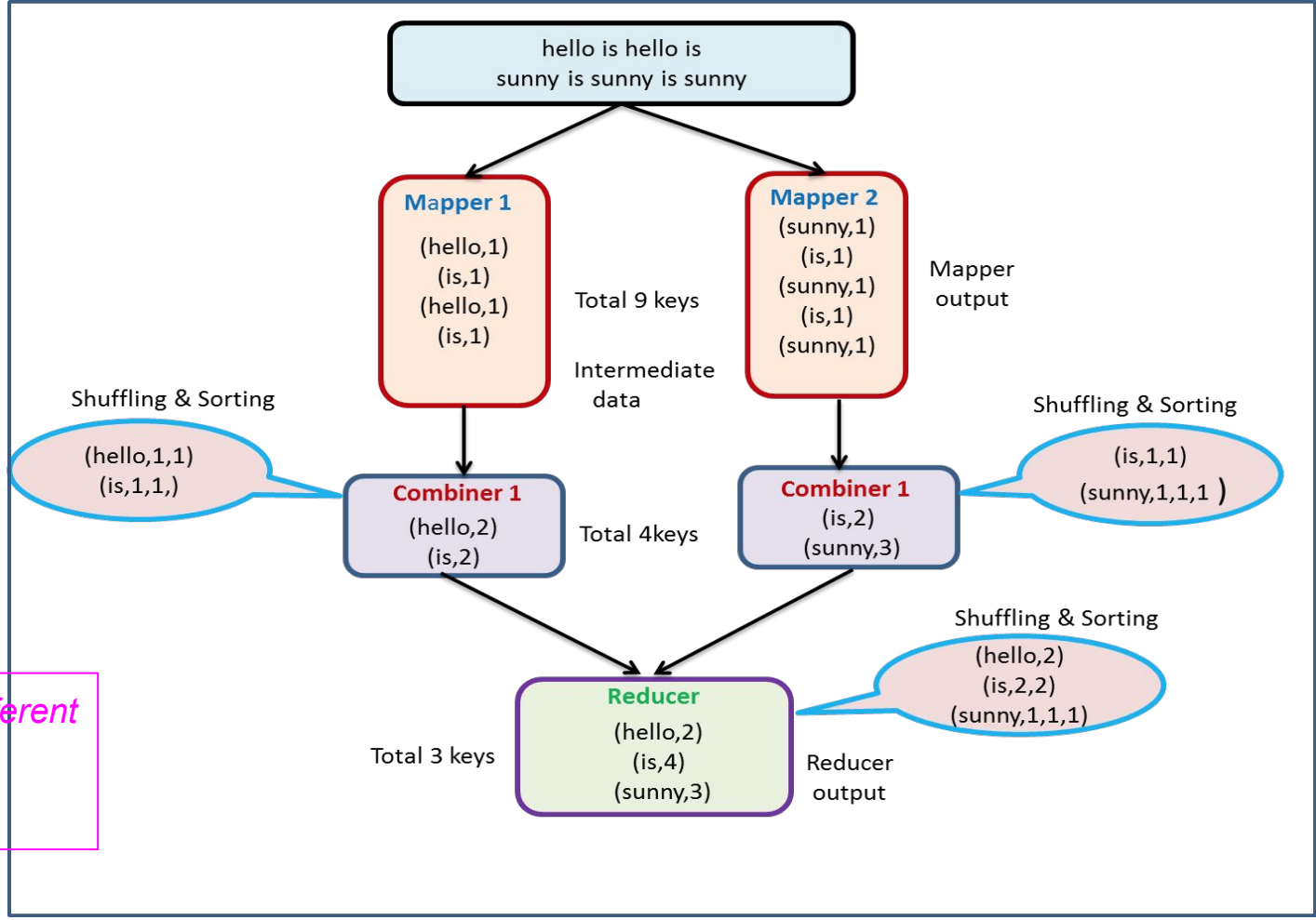
What's the best way to do this??

Yes, MR!

Back to Map-Reduce! Yeah!!

MR Execution Flow:

1. Input splitting
2. Task allocation
3. Map phase
4. Combiner phase
5. Partition phase
6. Shuffle / Sort phase
7. Reduce phase

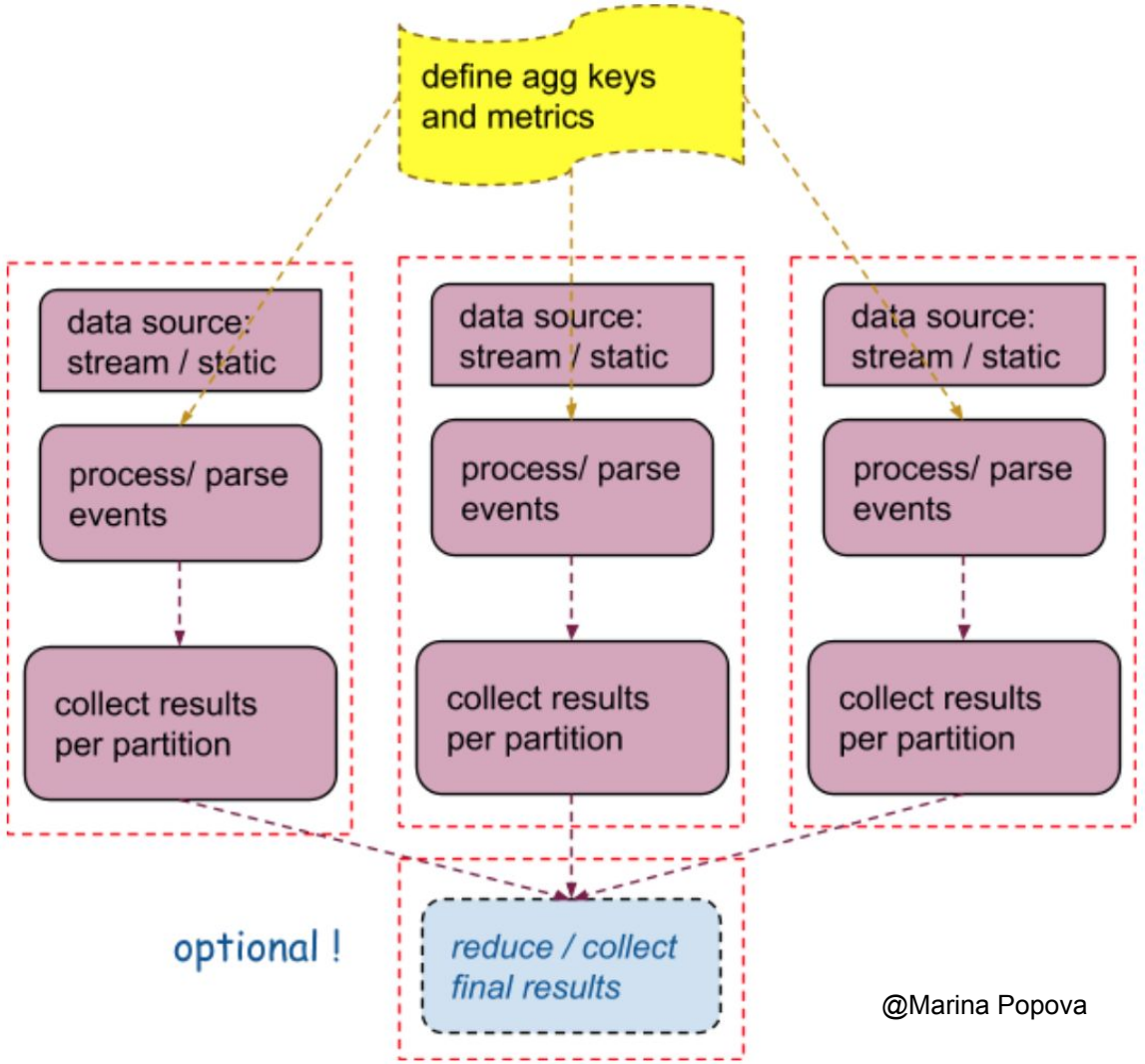


Processing topology is different for Stateless vs Stateful operations !

References: http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/

Stateless operations

- filter
- transform
- select ...
- uniques (approx - HLL)

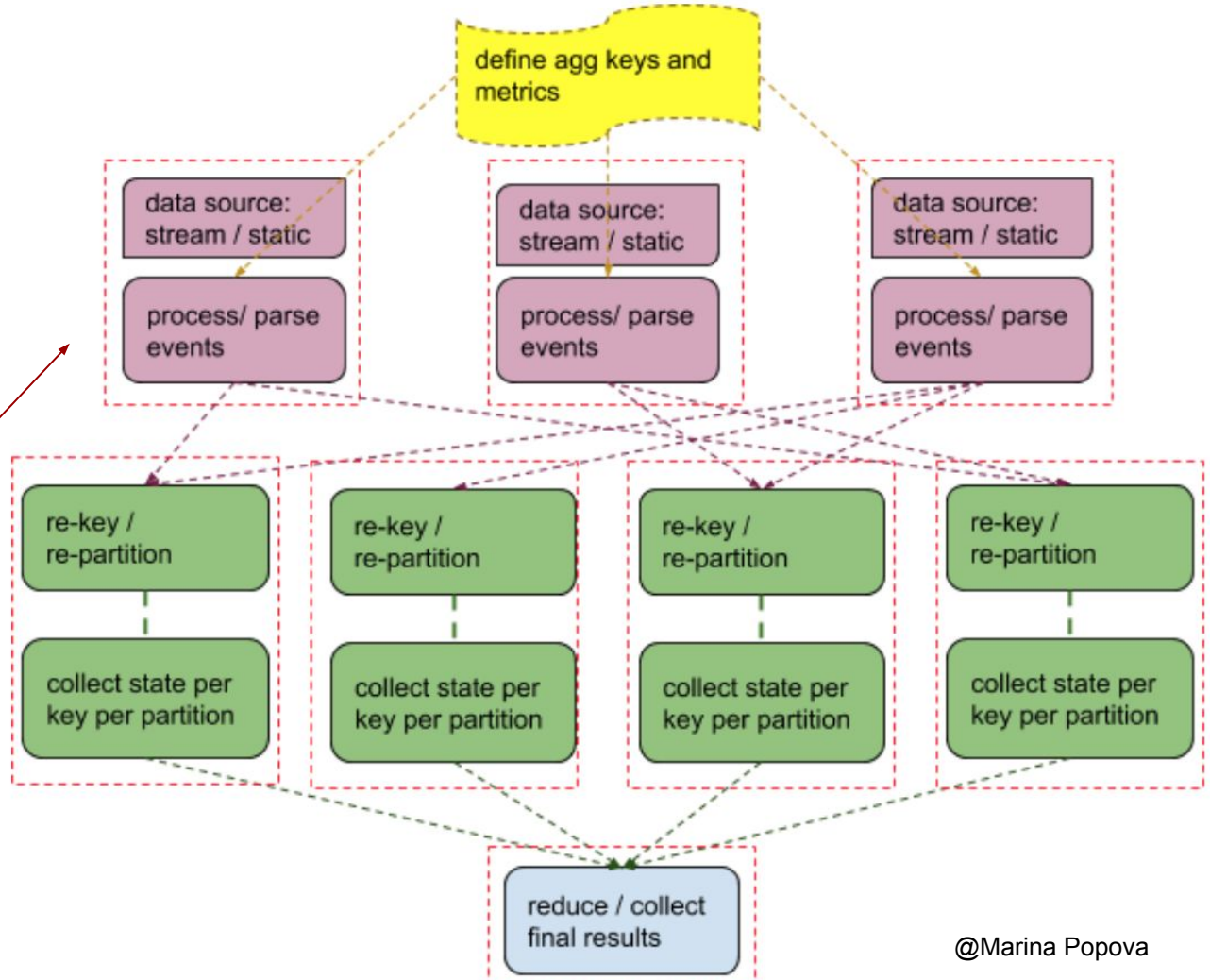


@Marina Popova

Stateful operations

- aggregates:
  - MIN(), MAX()
  - TOPK
  - CountBy
- uniques (exact)

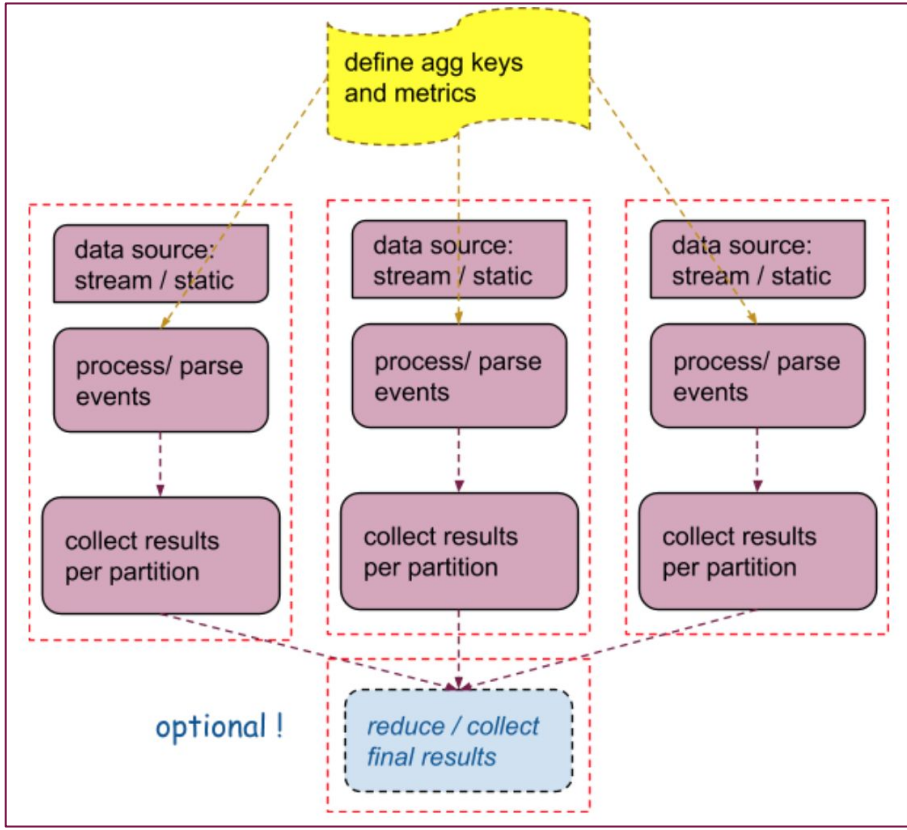
different physical servers!



@Marina Popova

KSQL: stateless operations

SELECT ... FROM ... WHERE ...
data: Kafka topic - partitions
processing: KSQL server (consumer group)
results: output Kafka topic - partitions [might be co-partitioned or not ....]
results: console/CLI



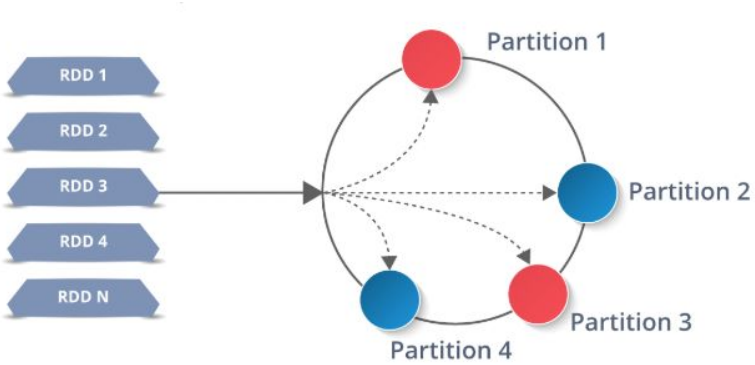
@Marina Popova

KSQL: stateful operations

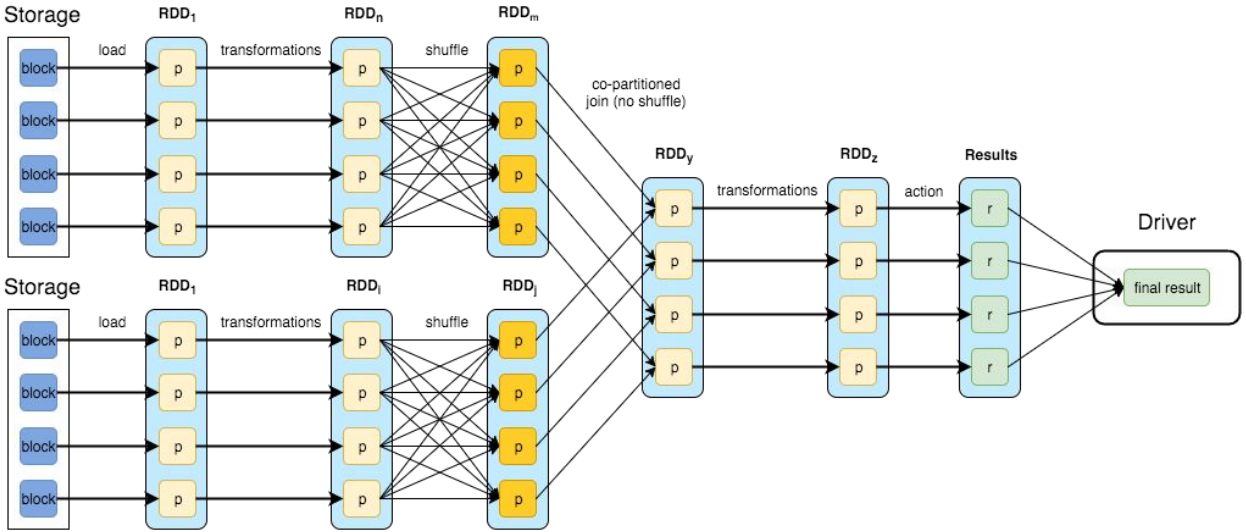
KSQL: SELECT ... FROM ... WHERE ...
data: Kafka topic - partitions
processing: KSQL server (consumer group)
internal re-partitioning topic
state: RocksDB (per partition)
partial results: output Kafka topic - partitions [might be co-partitioned or not ....]
results: console/CLI

Back to Spark: RDDs - recap:

- Spark is based on distributed data structures called Resilient Distributed Datasets (RDDs) which can be thought of as **immutable parallel data structures**
- A JavaRDD<String> is essentially just a List<String> dispersed amongst each node in our cluster, with each node getting several different chunks of our List.
- RDDs work by splitting up their data into a series of partitions to be stored on each executor node. Each node will then perform its work only on its own partitions.
- There are two ways to create RDDs: parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or **any data source offering a Hadoop InputFormat.**



Spark: operations on RDDs

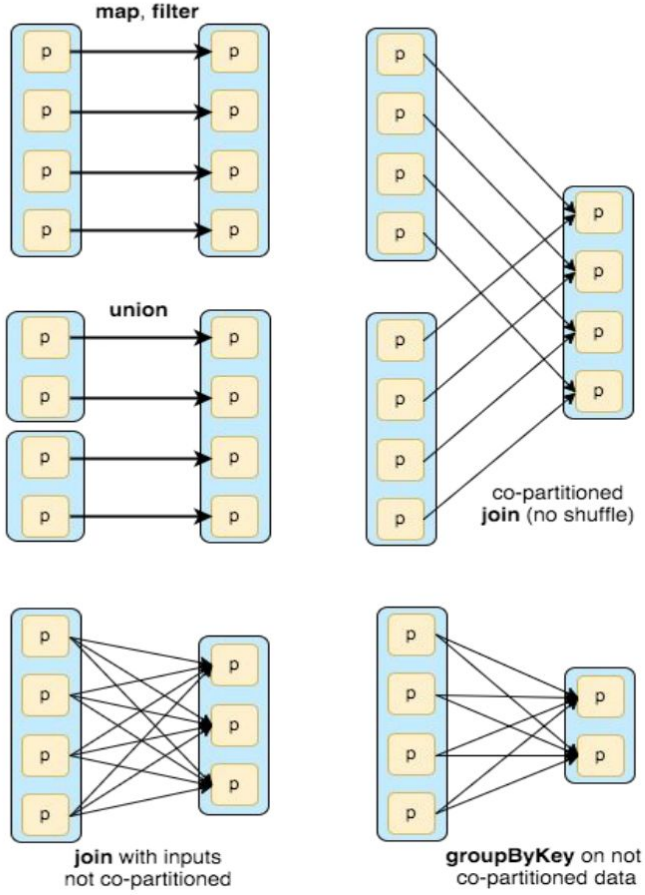


Transformations create dependencies between RDDs

There are two types of dependencies, **"narrow"** and **"wide"**

Ref: http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

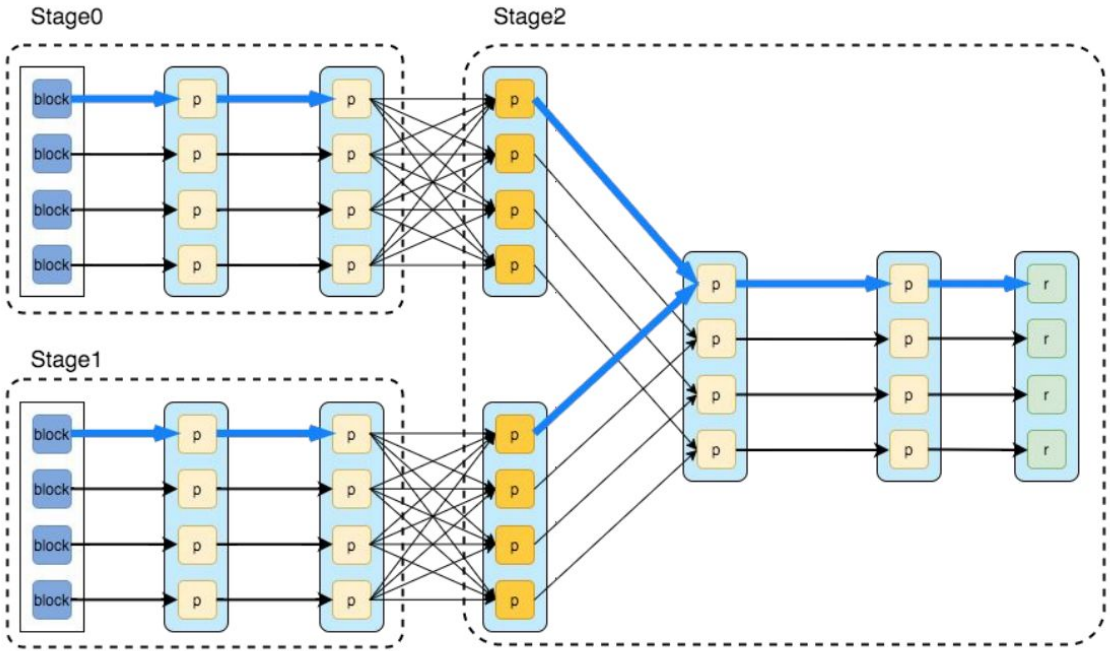
Dependency types



- Narrow (pipelineable)
  - each partition of the parent RDD is used by at most one partition of the child RDD
  - allow for pipelined execution on one cluster node
  - failure recovery is more efficient as only lost parent partitions need to be recomputed

- Wide (shuffle)
  - multiple child partitions may depend on one parent partition
  - require data from all parent partitions to be available and to be shuffled across the nodes
  - if some partition is lost from all the ancestors a complete recomputation is needed

Stages and Tasks



- Stages breakdown strategy
  - check backwards from final RDD
  - add each “narrow” dependency to the current stage
  - create new stage when there’s a shuffle dependency
- Tasks
  - *ShuffleMapTask* partitions its input for shuffle
  - *ResultTask* sends its output to the driver

Spark: stateful + shuffle operations

```
sparkContext.textFile("hdfs://...")
 .flatMap(line => line.split(" "))
 .map(word => (word,
1)).reduceByKey(_ + _)

.saveAsTextFile("hdfs://...")
```

OR with SparkSQL:

```
Dataset<Row> sqlDF =
spark.sql("SELECT * FROM people");
```

data: HDFS blocks wrapped into RDDs

processing: Executor Tasks

shuffle over network

state: in-memory or local disk when spilling

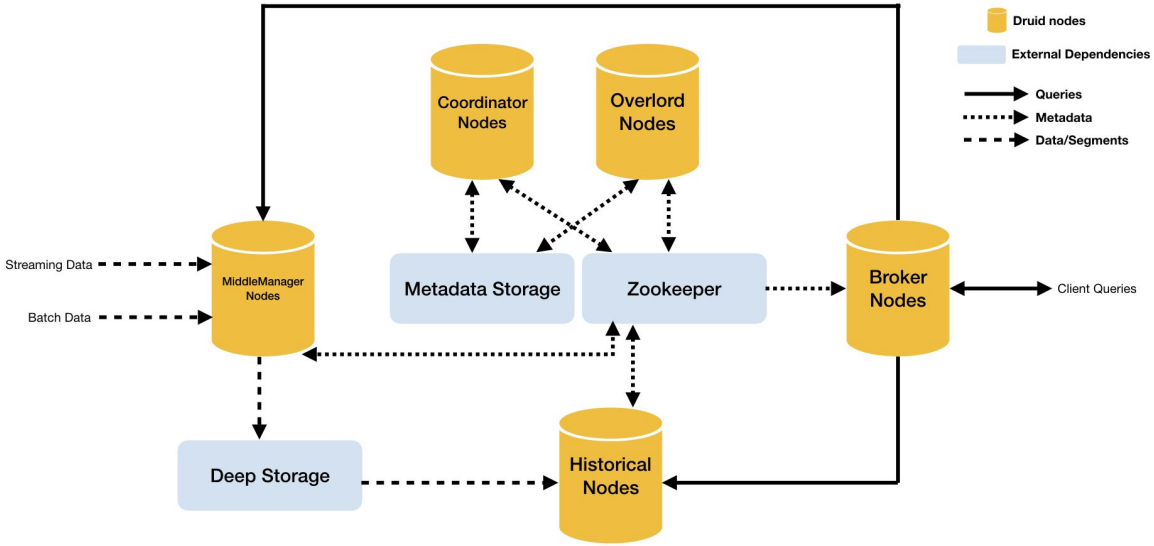
results: in-memory / HDFS blocks - sent to Driver

results: Driver app

@Marina Popova

Druid: Architecture Overview

http://druid.io/docs/latest/design/index.html#architecture

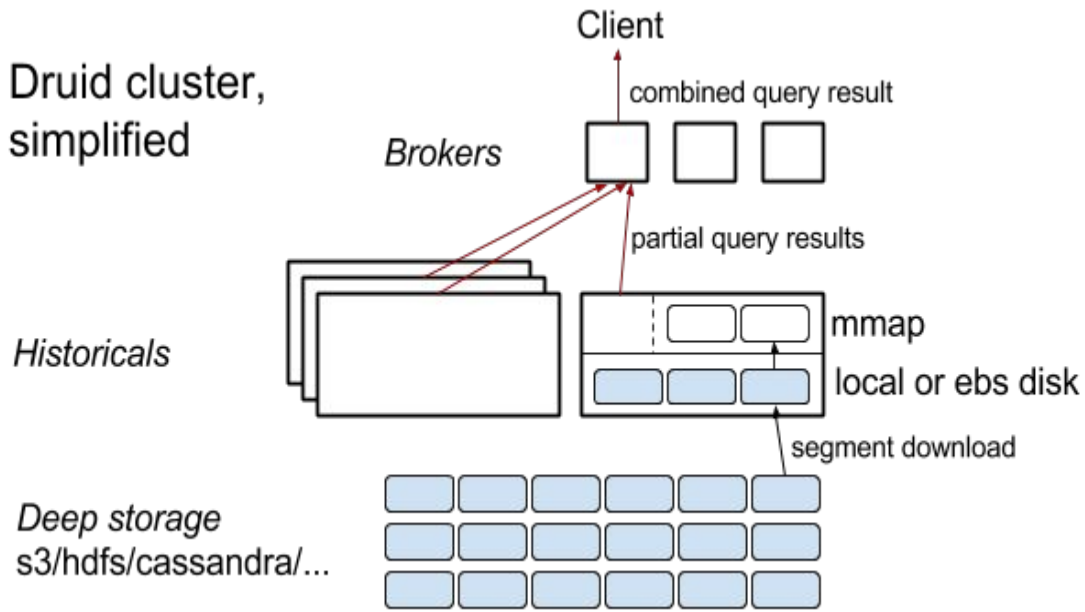


Ingestion/Indexing Details

- incoming data is parsed and stored into Deep Storage (S3 or HDFS) in segments
- uses proprietary storage format
- columnar storage
- compression - optimized for columns
- segments are indexed (using Druid-specific indexing tecniques)
- meta-data info about each index is stored in the Metadata Storage (location of the segment, time range, etc.)

Druid: Architecture Overview - Query Path

https://medium.com/@leventov/the-problems-with-druid-at-large-scale-and-high-load-part-1-714d475e84c9



Querying Details

- Historical nodes load available segments into the local storage
- then they update Zookeeper with this info
- Brokers use ZK to know which historicals serve which nodes
- Client query comes to a randomly chosen Broker node first
- this Broker determines which Brokers have all required segments to serve the query - and issues sub-queries to all of them
- this Broker also aggregates the partial results from the sub-queries

@Marina Popova

Druid: Architecture Overview

https://medium.com/@leventov/the-problems-with-druid-at-large-scale-and-high-load-part-1-714d475e84c9

Where does Druid excel?

- aggregation on ingest, with predefined dimensions and metrics
- scalable ingestion into deep storage (S3 or HDFS)
- sub-second queries on rolled up data - including historical rollups

Where can it cause issues?

- no fault-tolerance on the query execution path
  - a single query could end up being processed on hundreds of historical nodes. When a broker sends sub-queries to historicals, it needs all of those sub-queries to complete successfully, before it could return anything to the client. If any of the sub-queries fail, the whole query fails.
- TopK queries are approximate
- Brokers don't handle straggling sub-queries to the historicals: so the whole query query is always slower than the slowest historical, used in this query - and this can happen due to potential unequal distribution of load/data in historicals
- Huge variance in performance of historical nodes due to possibilities for uneven load distribution among the historicals.

Druid: aggregation/ Ad-Hoc queries operations

```
    "metricsSpec" : [],
    "granularitySpec" : {
      "type" : "uniform",
      "segmentGranularity" : "day",
      "queryGranularity" : "none",
      "intervals" : ["2015-09-12/2015-09-13"],
      "rollup" : false
    }
  },
  "ioConfig" : {
    "type" : "index",
    "firehose" : {
      "type" : "local",
      "baseDir" : "quickstar
      "filter" : "wikiticker
    },
    "appendToExisting" : fal
  },
```

JSON Spec file

data: streaming (Kafka) or HDFS/S3

processing: Middle Managers/ Historicals

shuffle over network

partial state: Brokers: in-memory or local disk

partial results: Brokers: in-memory

results: aggregated on the coordinating Broker

@Marina Popova

Druid - architecture type?

Druid can be used as an all-in-one Big Data processing system - what type of system is it?

??????

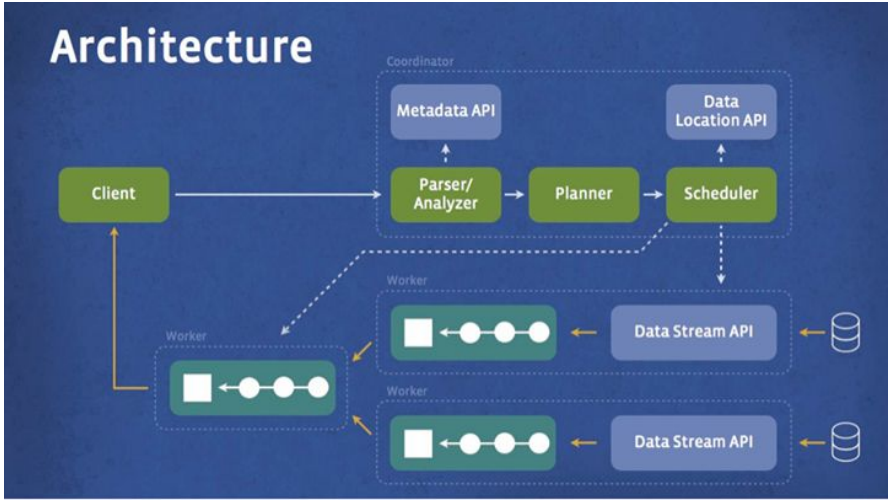
Answer:
Pure Type 4 System - Lambda Architecture !

Athena: Architecture

Amazon Athena is an interactive query service that makes it easy to analyze data directly in Amazon S3 using standard SQL

it is based on Presto - distributed SQL engine developed by FB

- Presto is a distributed system that runs on Hadoop, and uses an architecture similar to a classic massively parallel processing (MPP) database management system
- It has one coordinator node working in synch with multiple worker nodes
- Users submit their SQL query to the coordinator which uses a custom query and execution engine to parse, plan, and schedule a distributed query plan across the worker nodes.
- After the query is compiled, Presto processes the request into multiple stages across the worker nodes. All processing is in-memory, and pipelined across the network between stages, to avoid any unnecessary I/O overhead.



Athena: Architecture

Performance and Sizing Implications:

- ORDER BY: all rows of data are sent to ONE worker - and sorted in memory. If data does not fit in memory - query may fail
- GROUP BY - same

- Data partitioning - extremely important, to avoid long-running queries and high cost:
  - Partitions are hierarchical (eg Year -> Month -> Day). The objective with partitions is to "skip over" files that don't need to be read. Therefore, they will only provide a benefit if the WHERE clause uses the partition hierarchy.
  - For example, using SELECT ... WHERE year=2018 will use the partition and skip all other years.

- data format: should use Parquet for better support of select-column-based queries

Athena: Ad-Hoc queries over S3

```
Hive-SQL query:

SELECT * FROM "varnish_events"
WHERE "responsecode" != 200
```

data: S3

processing: Presto MR (in-memory) jobs

re-key/shuffle over network

partial state: Presto Workers

partial results: Presto Workers

results: S3, Console

@Marina Popova