Scalable realtime data processing using Kafka Streams

Trayan Iliev

IPT – IT Education Evolved, http://iproduct.org/

Copyright © 2003-2023 IPT – Intellectual Products & Technologies Ltd. All rights reserved. Licensed under CC BY-NC-SA 4.0

Следете актуалните обяви за **Java**

DEV.BG

Machine Learning + Big Data in Real Time + Cloud Technologies

=> The Future of Intelligent Systems

Trayan Iliev

– CEO of IPT – Intellectual Products & Technologies – IT Education Evolved

– Oracle® certified programmer 15+ Y

– 12+ years IT trainer

– End-to-end reactive full-stack apps with Go, Python, Java, Kotlin, TypeScript, React, React Native, Angular, and Vue.js

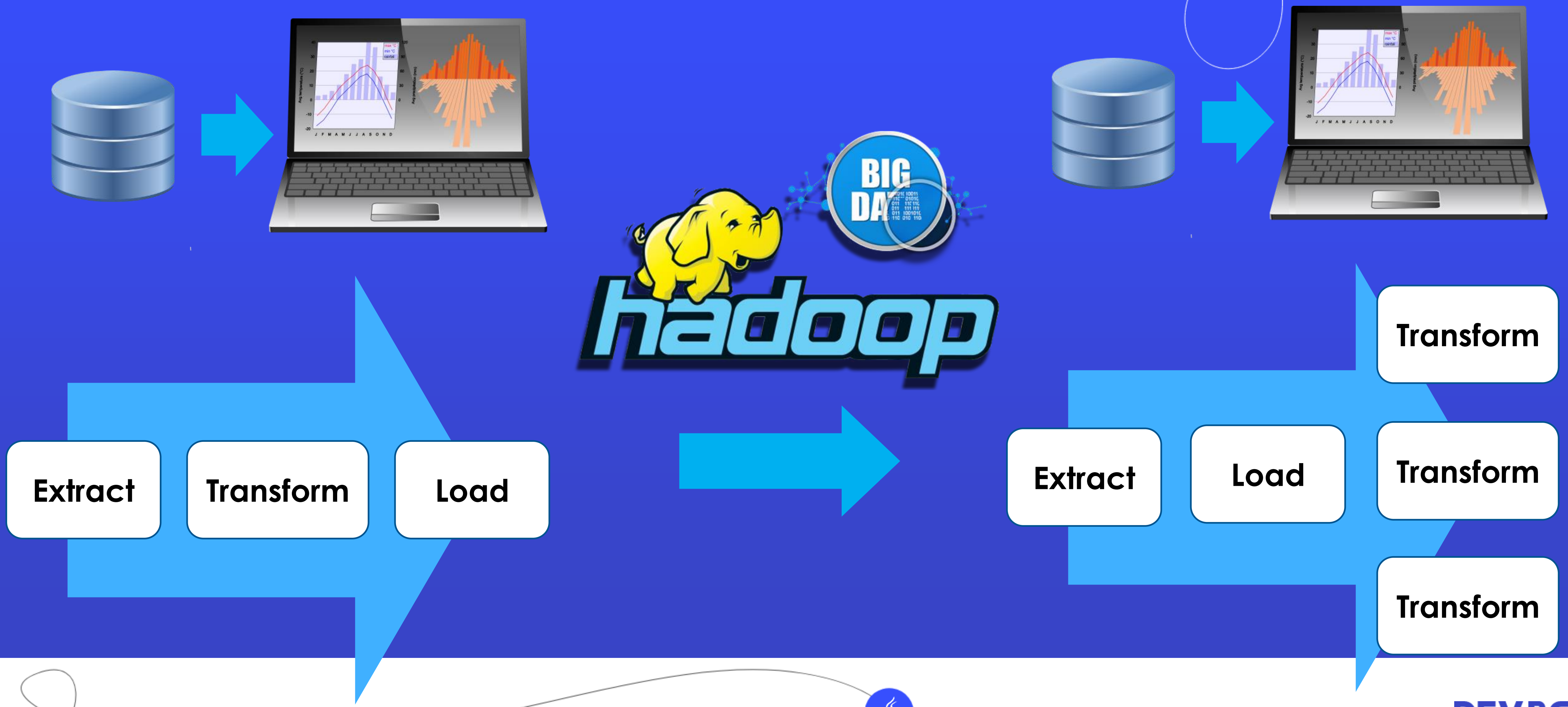
– Machine Learning (ML) and Deep Learning (DL) with Python, Neuro-Symbolic learning, big data/event processing in realtime (Spark, Kafka, etc.), IoT, уеб и мобилни технологии.

– Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Lecturer @ Sofia University – courses: Fullstack React, Angular & TypeScript, Spring 5 Reactive, Distributed Machine Learning, Practical Robotics & IoT

– Robotics / smart-things/ IoT enthusiast, RoboLearn hackathons organizer

Batch Processing



Data / Event / Message Streams

"Conceptually, a stream is a (potentially never-ending) flow of data records, and a transformation is an operation that takes one or more streams as input, and produces one or more output streams as a result."

Apache Flink: Dataflow Programming Model

Следете актуалните обяви за **Java**

DEV.BG

Data Stream Programming

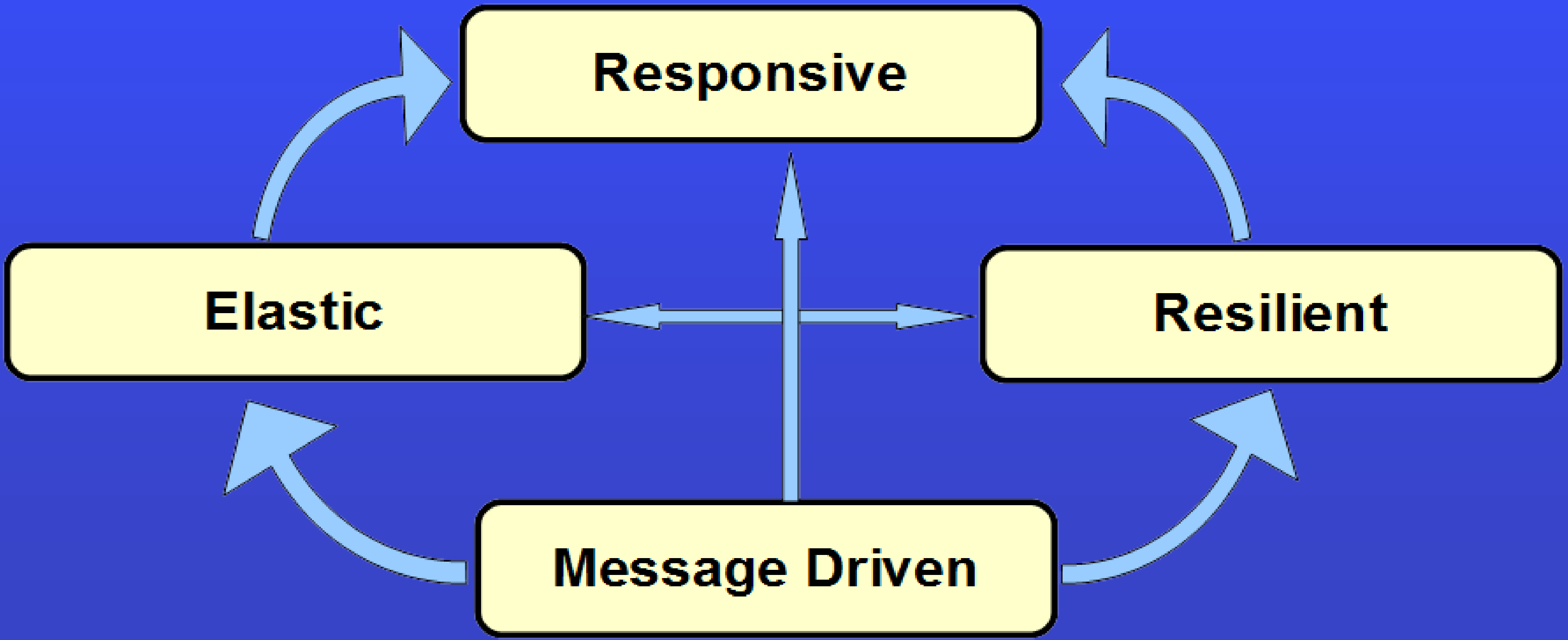
The idea of abstracting logic from execution is hardly new -- it was the dream of SOA. And the recent emergence of microservices and containers shows that the dream still lives on.

For developers, the question is whether they want to learn yet one more layer of abstraction to their coding. On one hand, there's the elusive promise of a common API to streaming engines that in theory should let you mix and match, or swap in and swap out.

Tony Baer (Ovum) @ ZDNet - Apache Beam and Spark: New competition for squashing the Lambda Architecture?

Reactive Manifesto

http://www.reactivemanifesto.org



Scalable, Massively Concurrent

- Message Driven – asynchronous message-passing allows to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages [Reactive Manifesto].

- The main idea is to separate concurrent producer and consumer workers by using message queues.

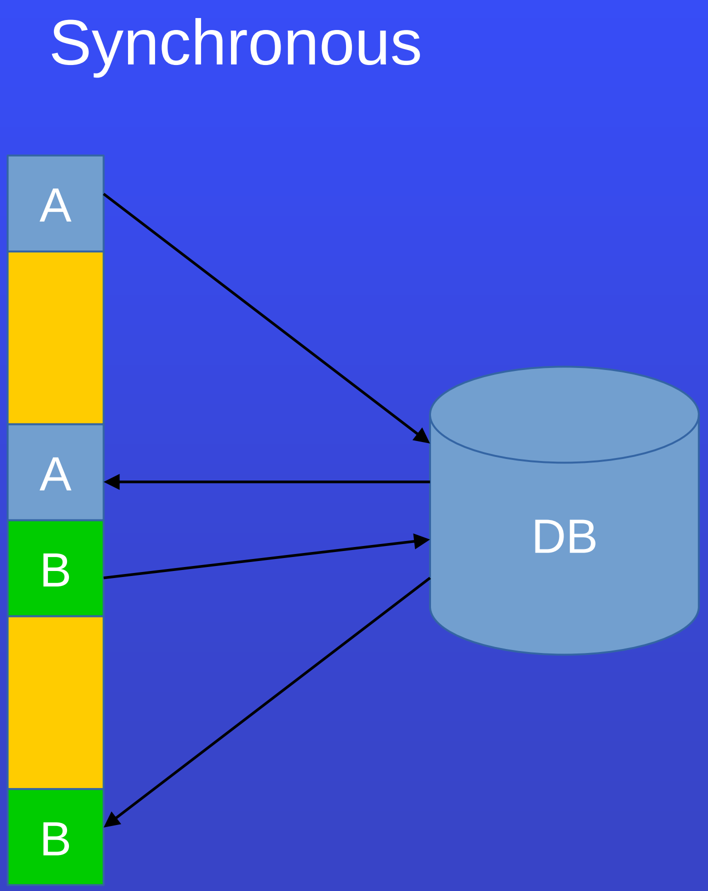
- Message queues can be unbounded or bounded (limited max number of messages)

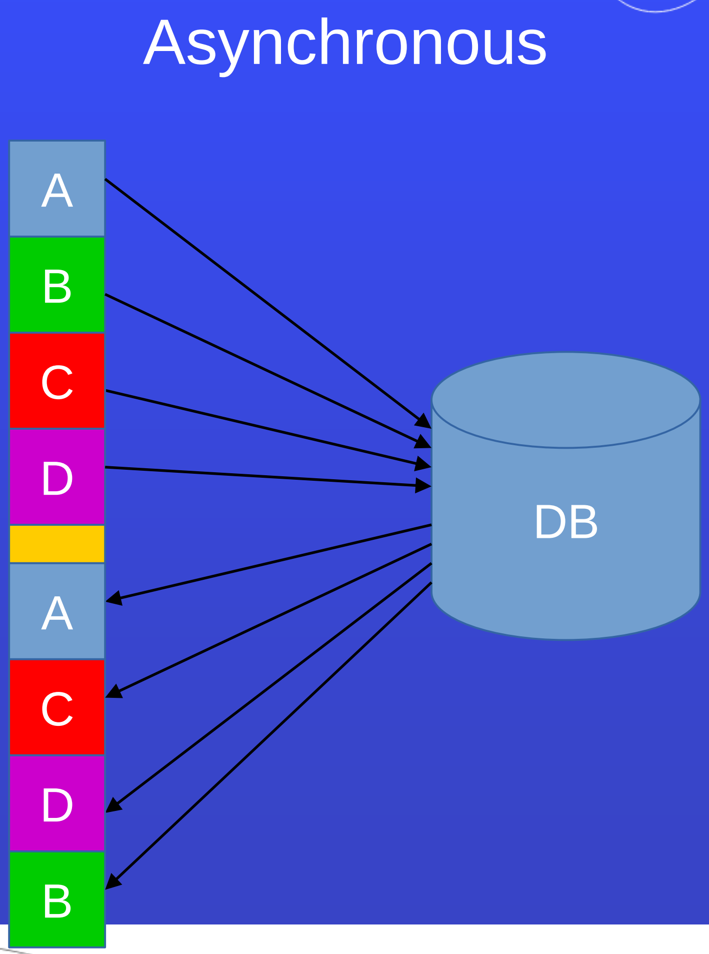
- Unbounded message queues can present memory allocation problem in case the producers outrun the consumers for a long period → OutOfMemoryError

Следете актуалните обяви за **Java**

DEV.BG

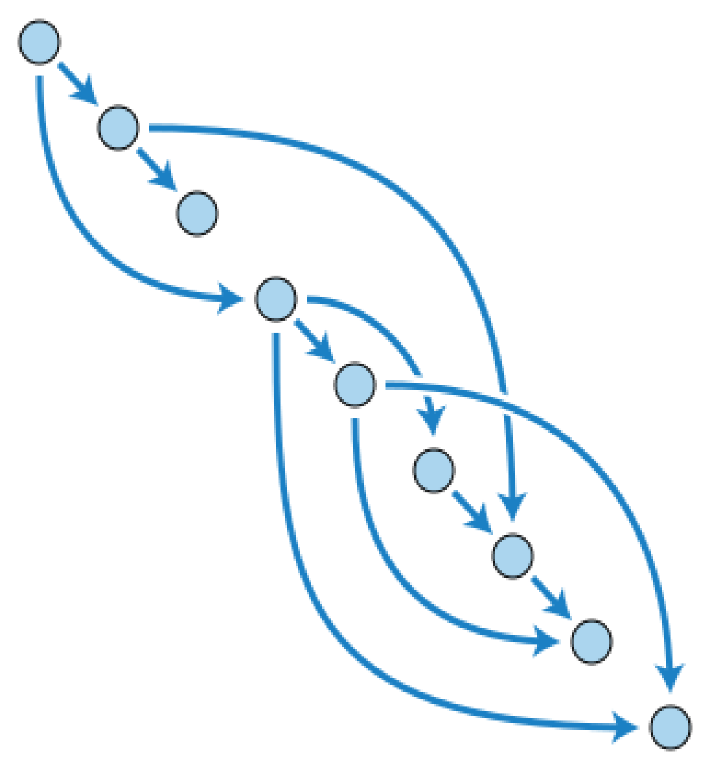
Synchronous vs. Asynchronous IO

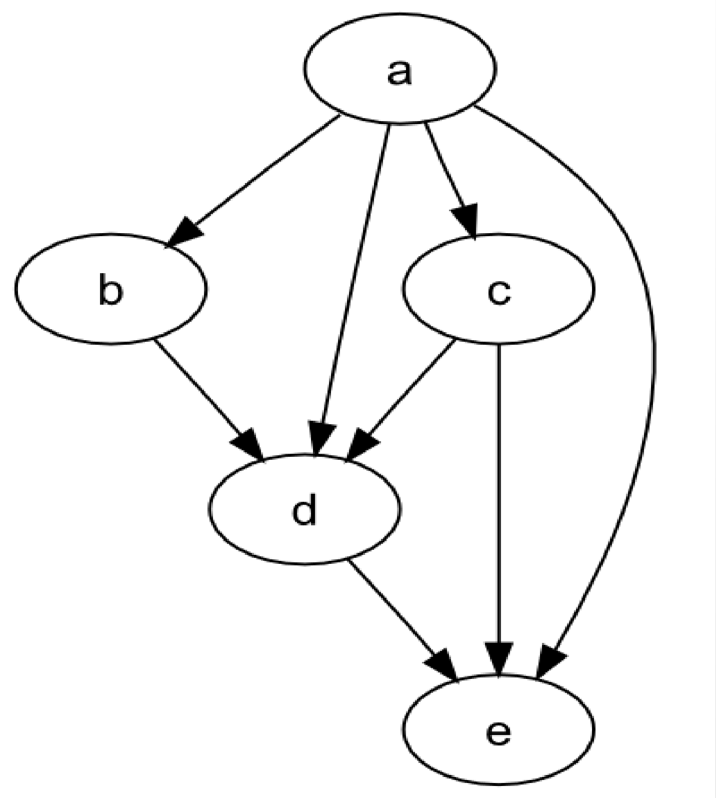


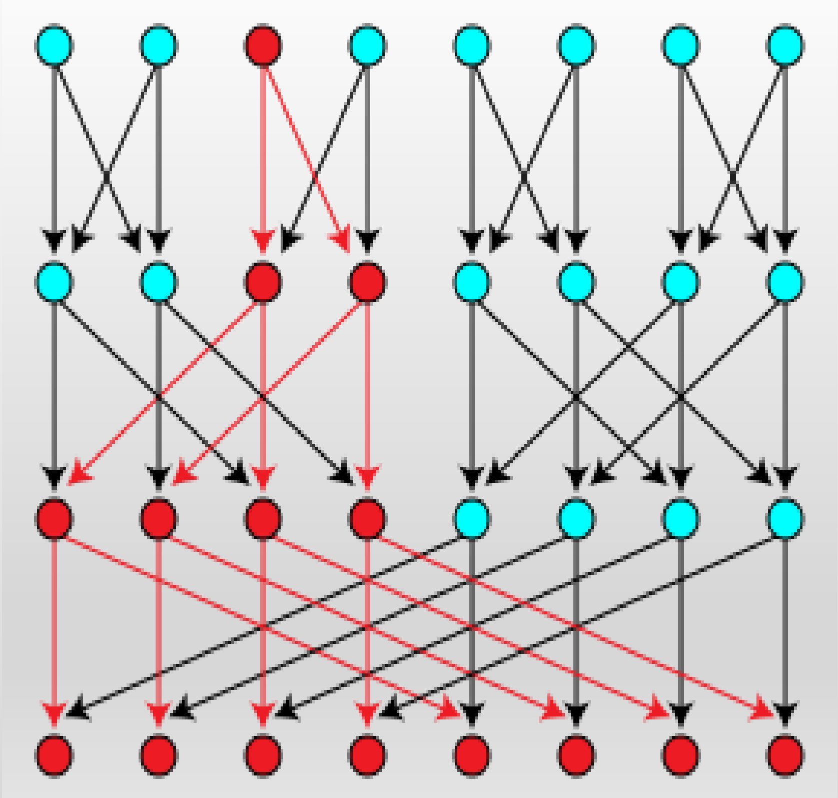


Следете актуалните обяви за **Java**
DEV.BG

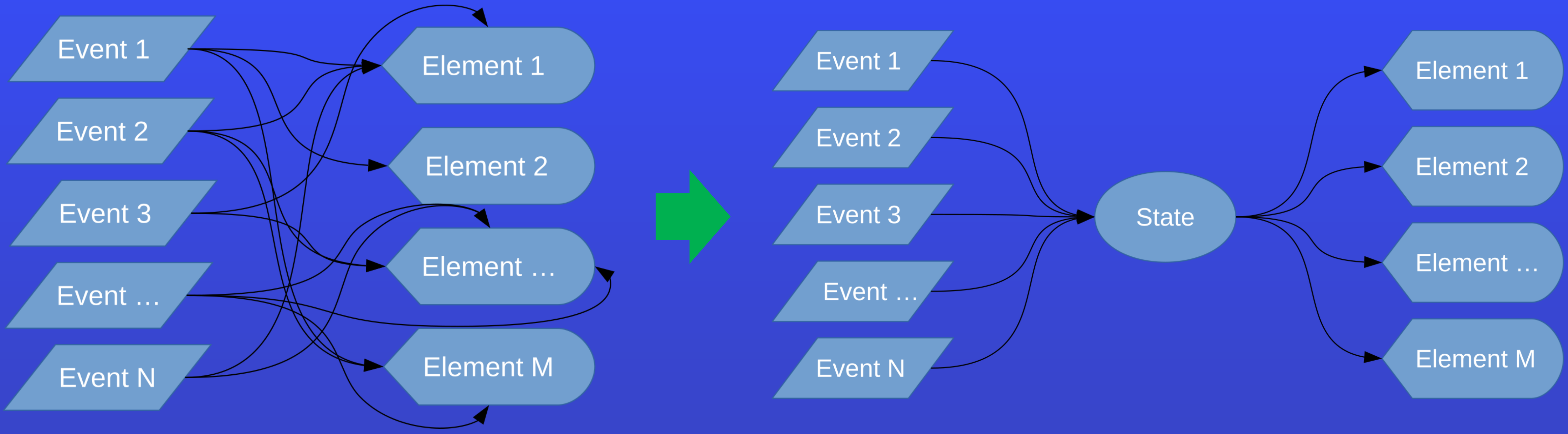
Stream Topology => Direct Acyclic Graph





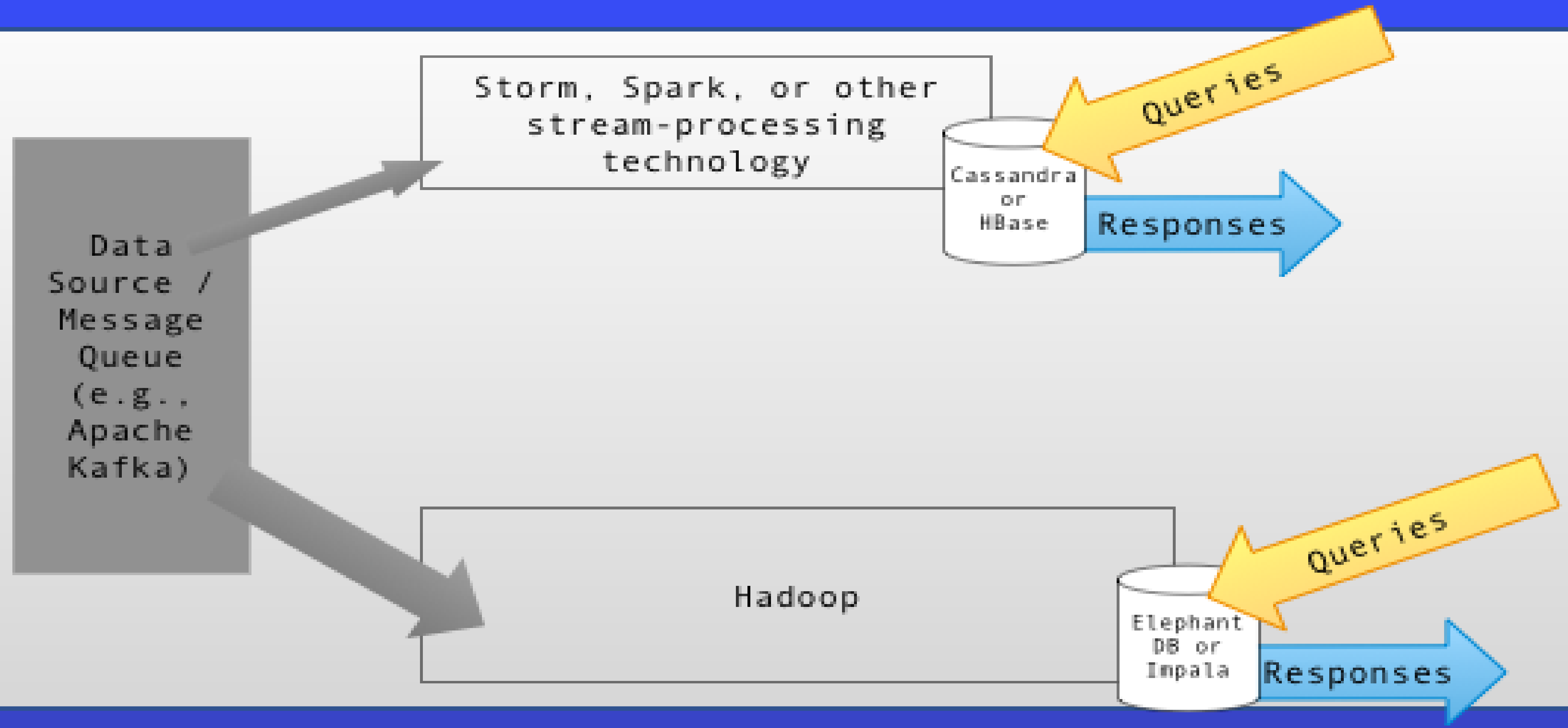


Event Sourcing – Events vs. Sate (Snapshots)



Lambda Architecture - I

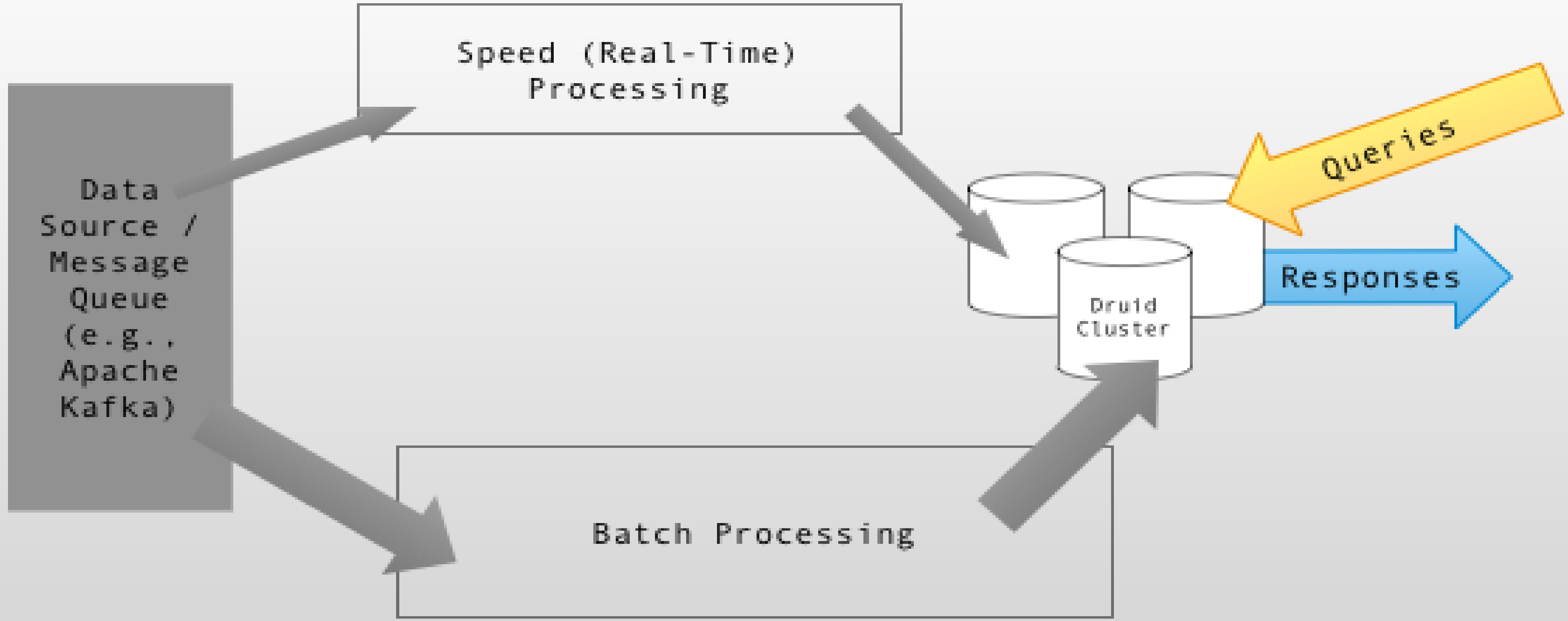
Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)



https://commons.wikimedia.org/w/index.php?curid=34963987, By Textractor - Own work, CC BY-SA 4

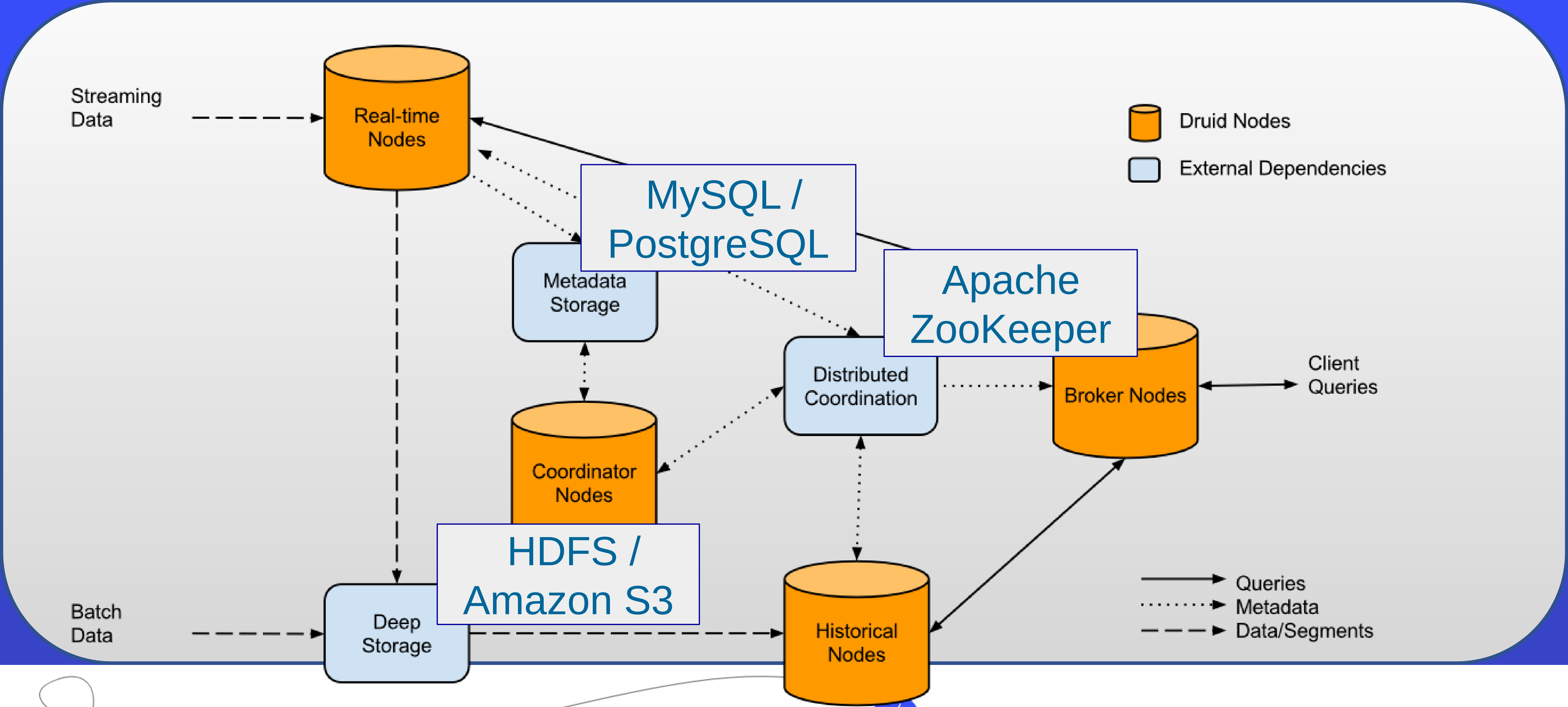
Lambda Architecture - II

Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)



https://commons.wikimedia.org/w/index.php?curid=34963986, By Textractor - Own work, CC BY-SA 4

Lambda Architecture - Druid Distributed Data Store



https://commons.wikimedia.org/w/index.php?curid=33899448 By Fangjin Yang, GFDL

Kappa Architecture - I

Query = K (New Data) = K (Live streaming data)

λ vs $\varkappa$

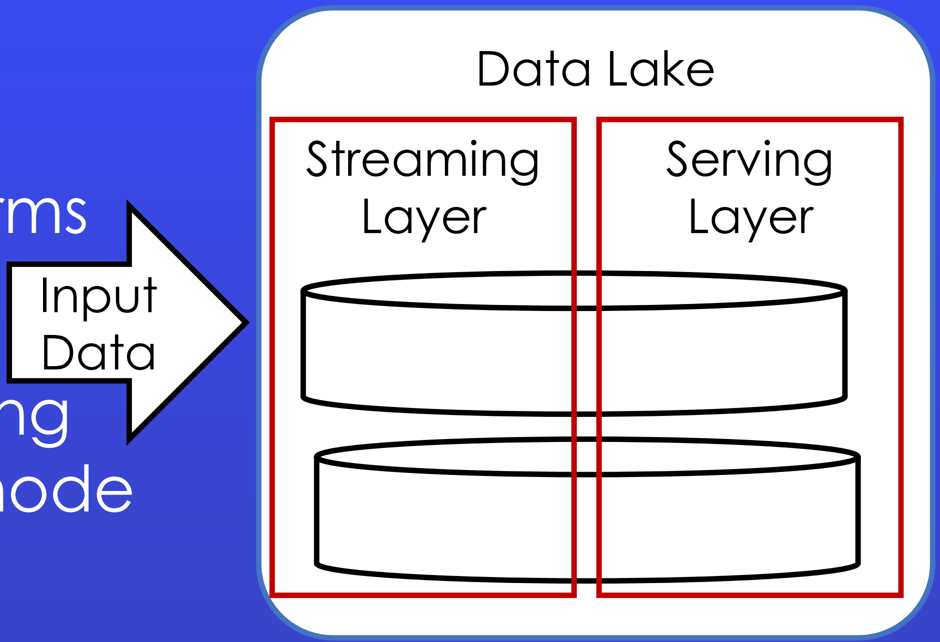
- Proposed by Jay Kreps in 2014
- Real-time processing of distinct events
- Drawbacks of Lambda architecture:
  - It can result in coding overhead due to comprehensive processing
  - Re-processes every batch cycle which may not be always beneficial
  - Lambda architecture modeled data can be difficult to migrate
- Canonical data store in a Kappa Architecture system is an append-only immutable log (like Kafka, Pulsar)

Следете актуалните обяви за **Java**

DEV.BG

Kappa Architecture -II

- Multiple data events or queries are logged in a queue to be catered against a distributed file system storage or history.
- The order of the events and queries is not predetermined. Stream processing platforms can interact with database at any time.
- It is resilient and highly available as handling terabytes of storage is required for each node of the system to support replication.
- Machine learning is done on the real time basis



Zeta Architecture

Main characteristics of Zeta architecture:

- file system (HDFS, S3, GoogleFS),
- realtime data storage (HBase, Spanner, BigTable),
- modular processing model and platform (MapReduce, Spark, Drill, BigQuery),
- containerization and deployment (cgroups, Docker, Kubernetes, etc.),
- Software solution architecture (serverless computing – e.g. Amazon Lambda)
- Recommender systems and machine learning

Следете актуалните обяви за **Java**

DEV.BG

Distributed Stream Processing – Apache Projects:

- Apache Spark is an open-source cluster-computing framework. Spark Streaming, Spark Mllib, Spark ML (ML pipelines using Dataframes)

- Apache Storm is a distributed stream processing – streams DAG

- Apache Samza is a distributed real-time stream processing framework.

Distributed Stream Processing – Apache Projects:

- Apache Flink - open source stream processing framework – stateful computations over data streams - Flink ML: Machine Learning library

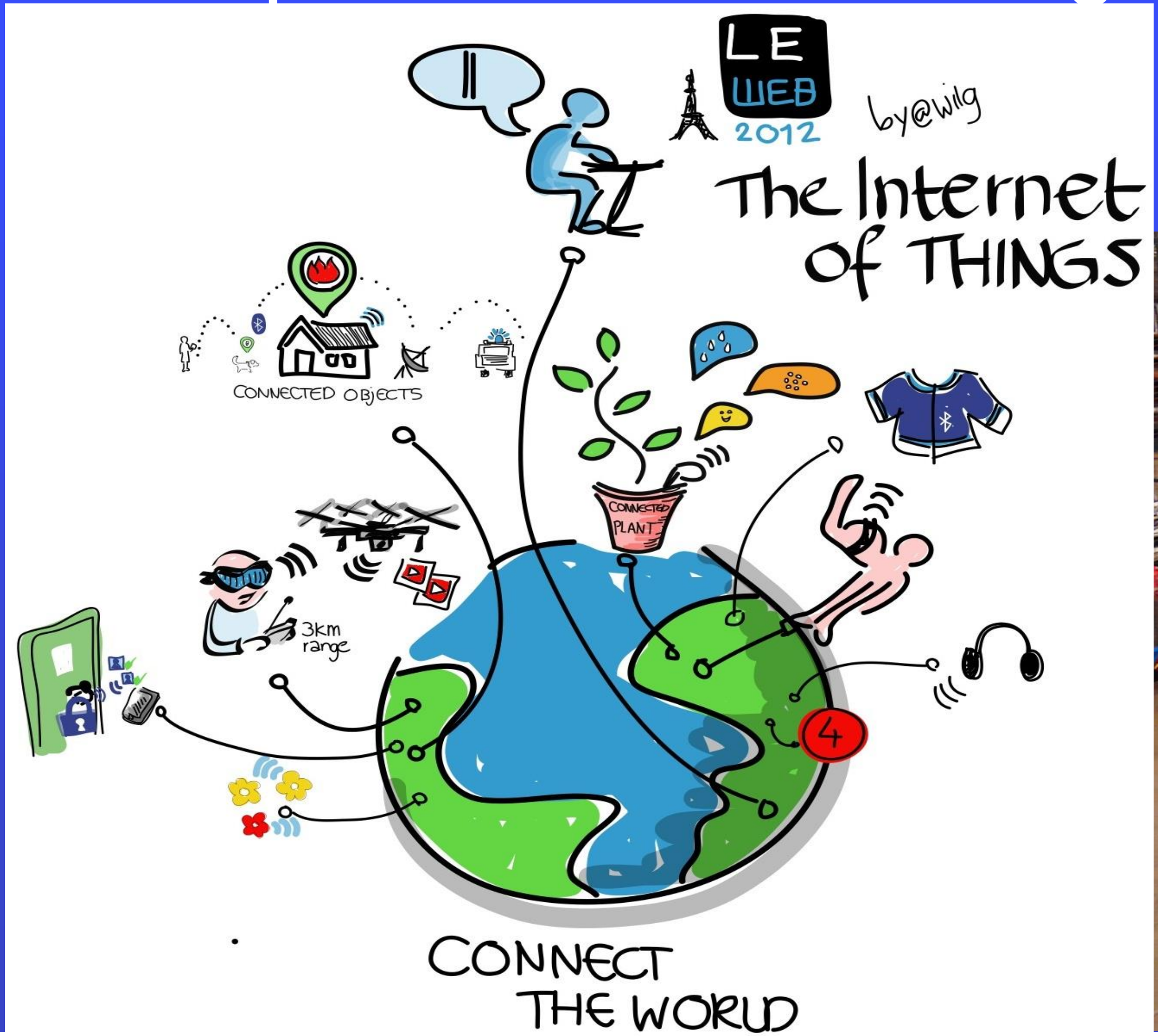
- Apache Kafka - open-source stream processing (Kafka Streams), real-time, low-latency, high-throughput, massively scalable pub/sub

- Apache Beam – unified batch and streaming, portable, extensible

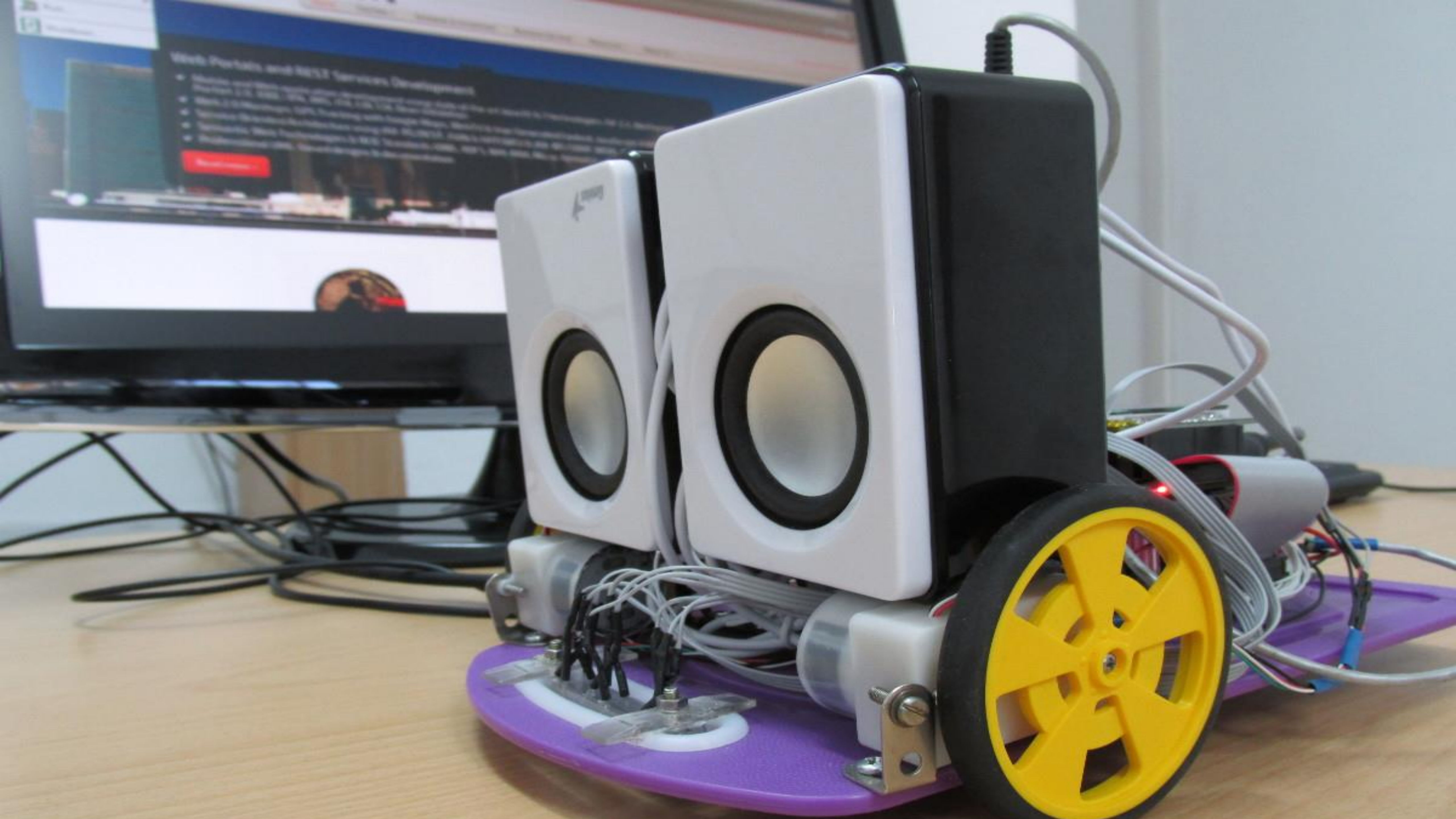
Следете актуалните обяви за **Java**

DEV.BG

Example: Internet of Things (IoT)



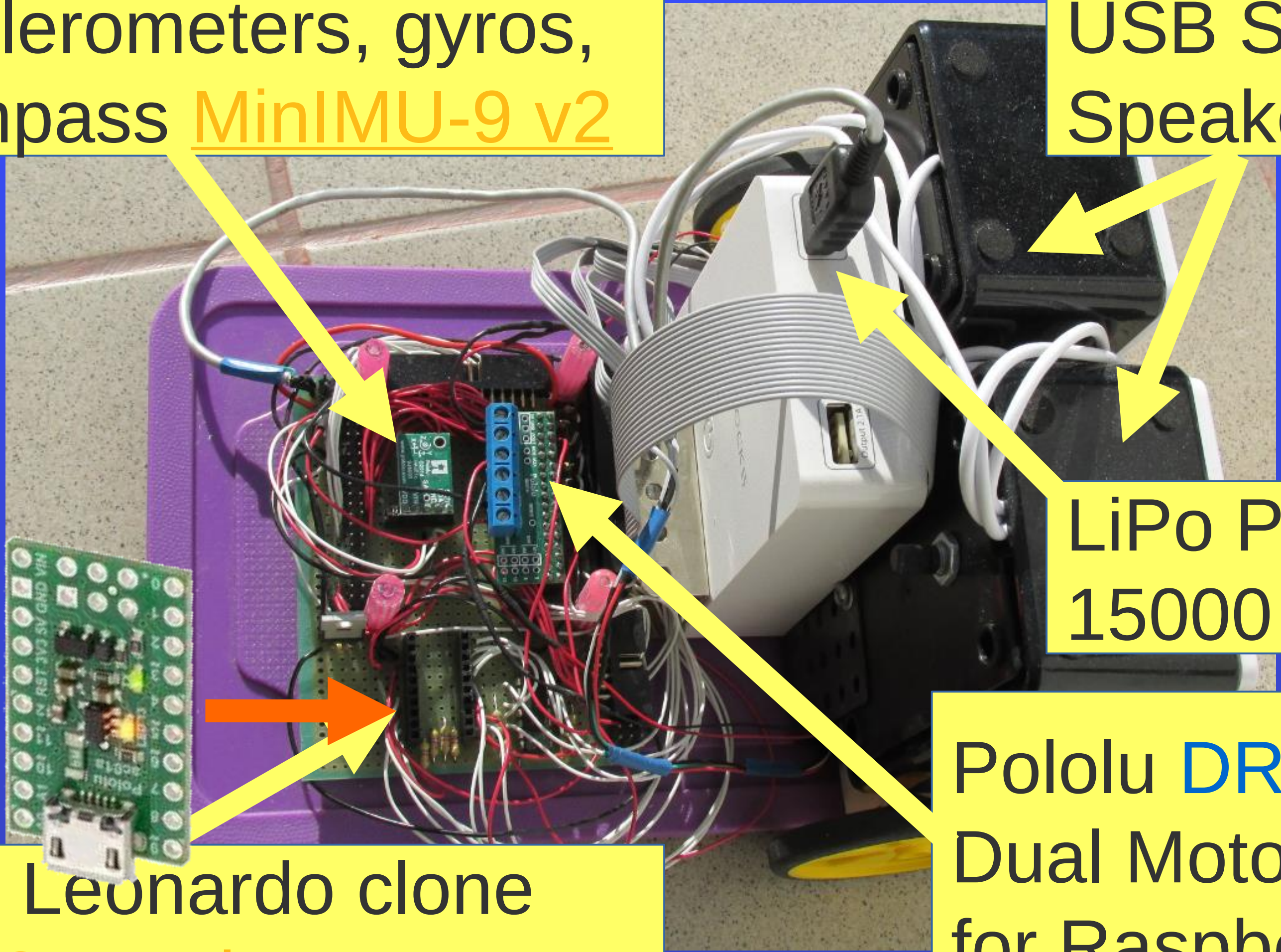
CC BY 2.0, Source: https://www.flickr.com/photos/wilgengebroed/8249565455/ ,
Radar, GPS, lidar for navigation and obstacle avoidance (2007 DARPA Urban Challenge)


Web Portals and REST Services Development

Example: Internet of Things (IoT) & Robotics

3D accelerometers, gyros, and compass MinIMU-9 v2

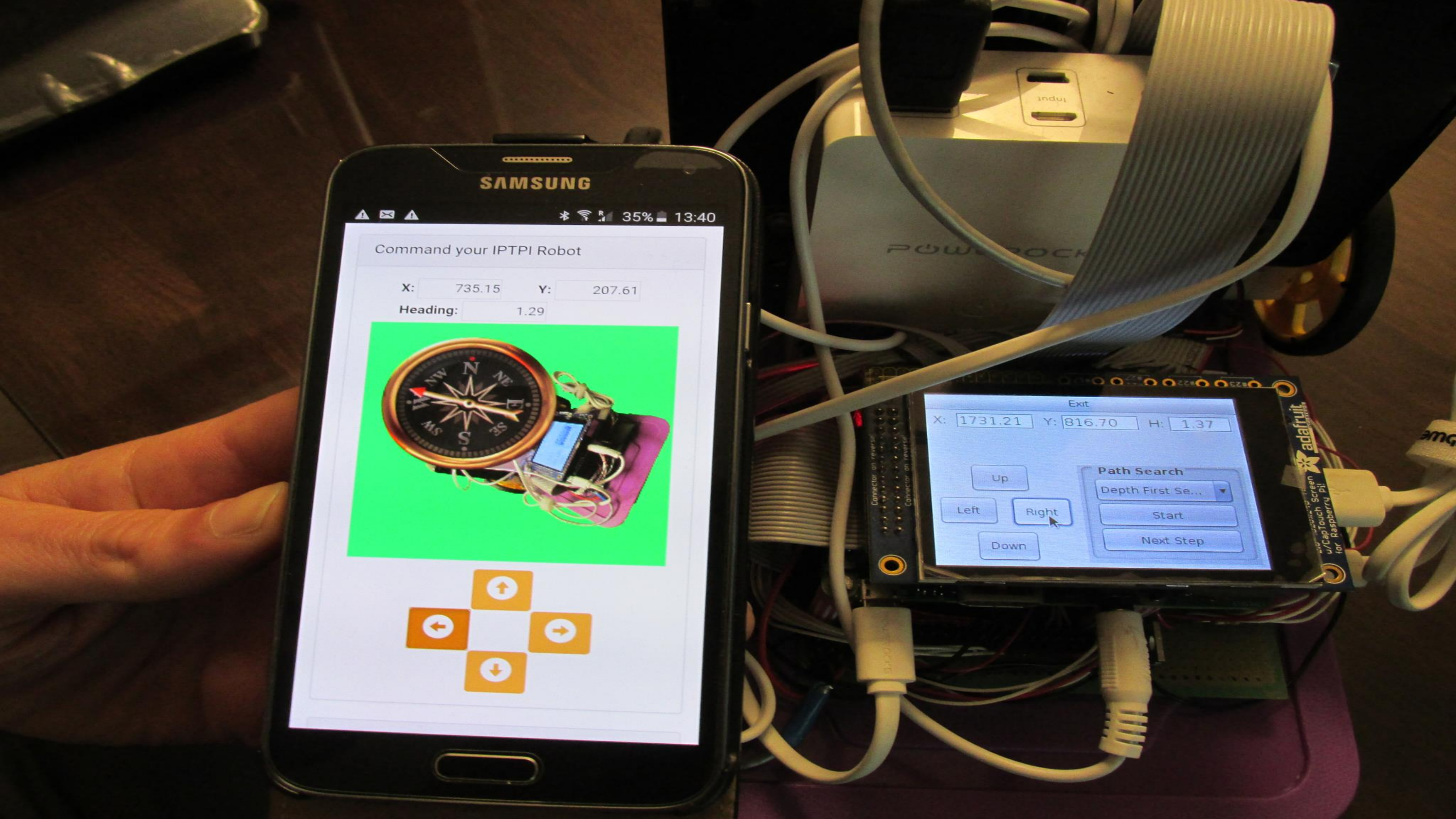
USB Stereo Speakers - 5V



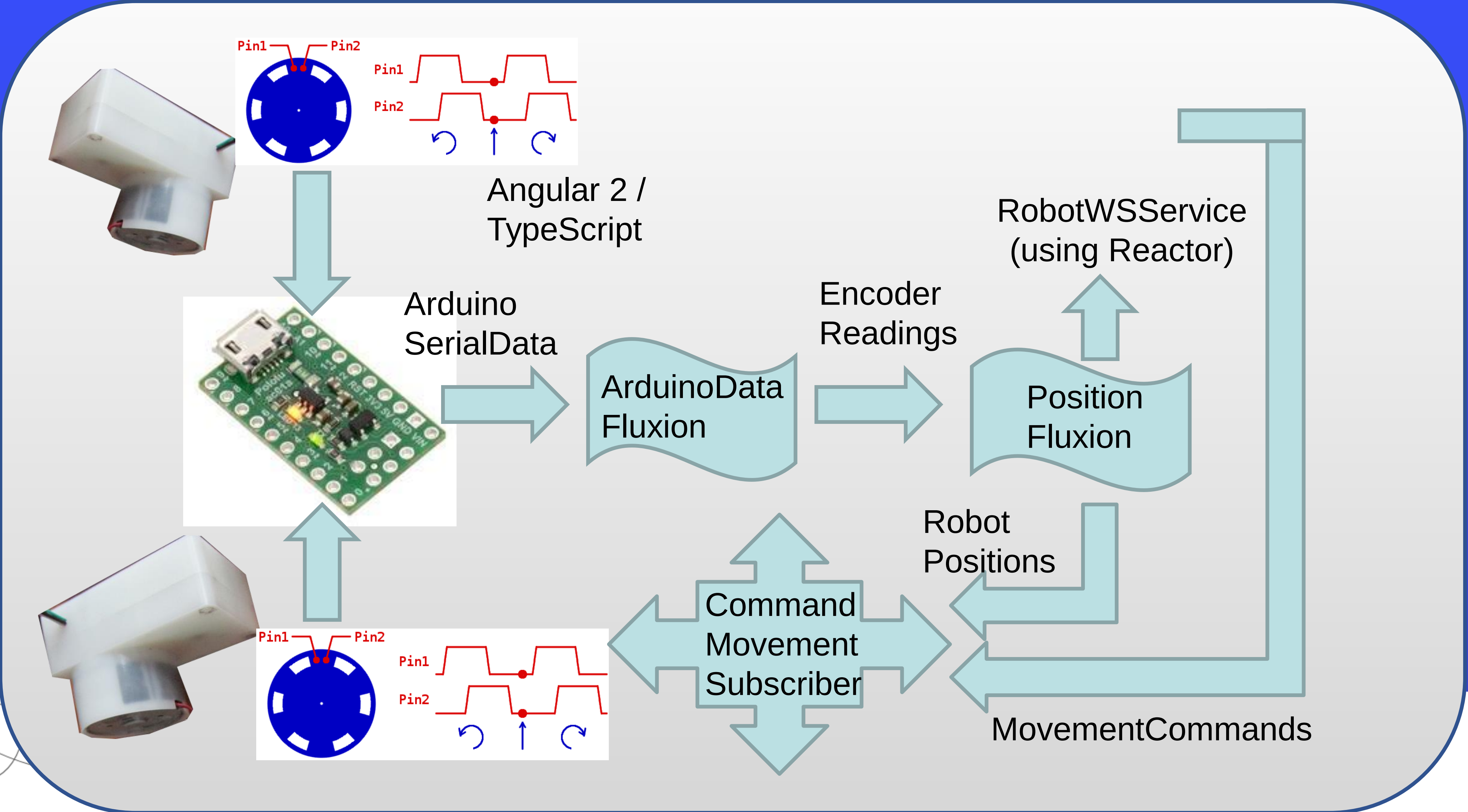
LiPo Powebank 15000 mAh

Pololu DRV8835 Dual Motor Driver for Raspberry Pi

Arduino Leonardo clone A-Star 32U4 Micro


SAMSUNG
35% 13:40
Command your IPTPI Robot
X: 735.15
Y: 207.61
Heading: 1.29
Exit
X: 1731.21
Y: 816.70
H: 1.37
Up
Left
Right
Down
Path Search
Depth First Se...
Start
Next Step
adafruit

Adaptive Robot Control using Reactive Streams



DEV.BG

Apache Kafka Distributed Streaming Platform

- Kafka achieves high-throughput, low-latency, durability, and near-limitless scalability by maintaining a distributed system based on commit logs, delegating key responsibility to clients, optimizing for batches and allowing for multiple concurrent consumers per message.
- Publish and subscribe (Pub/Sub) to streams of records – similar to a message queue or enterprise messaging system
- Store streams of records – in a fault-tolerant and durable way
- Process streams of records – as they occur (in real-time)

Two Types of Applications for Kafka

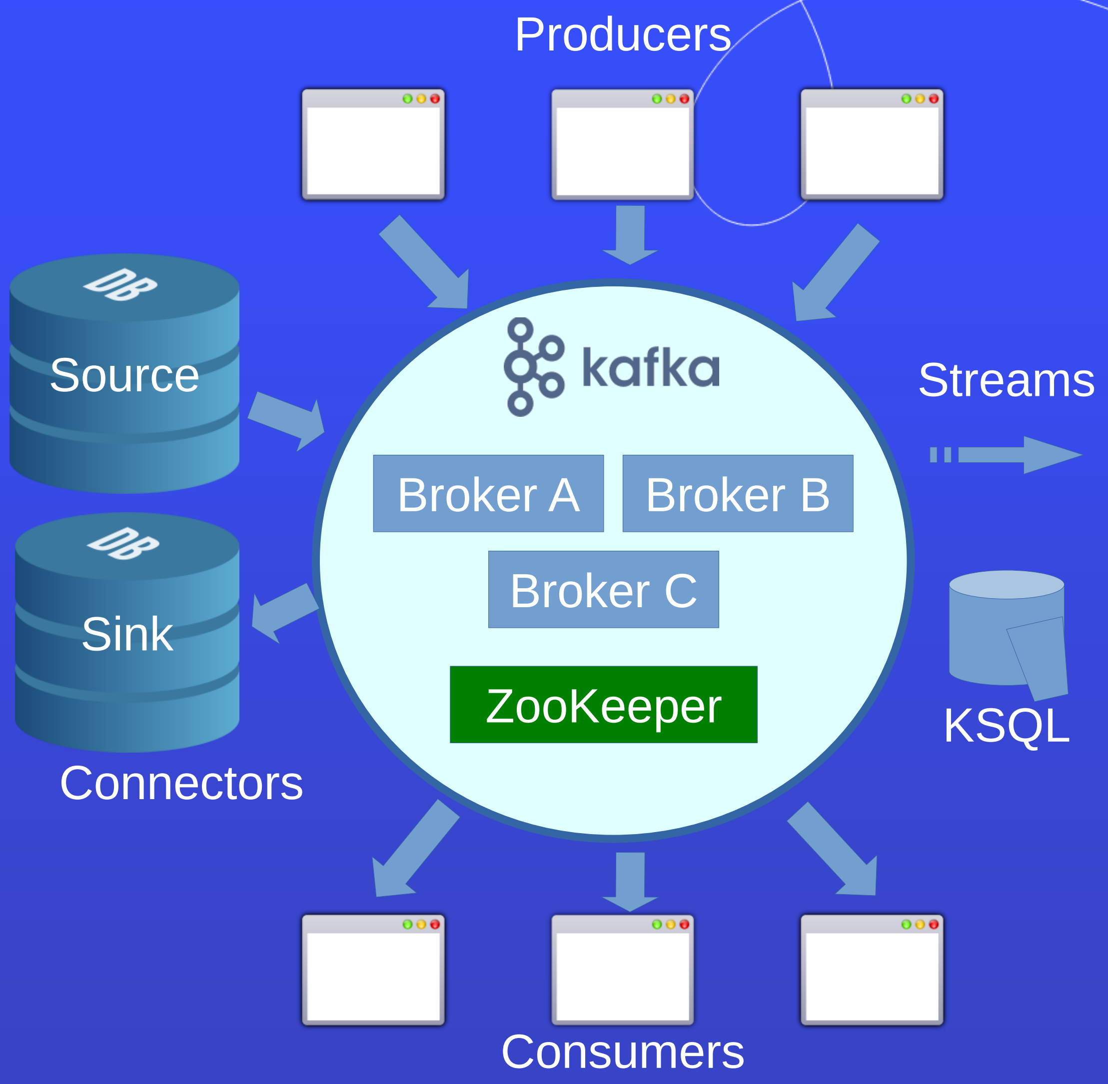
- Building real-time streaming data pipelines that reliably get data between systems or applications
- Building real-time streaming applications that transform or react to the streams of data – Kafka Streams

Apache Kafka Typical Use-Cases

- IoT, telemetry, and sensor networks
- Positional data / Logistics - supply chain and transportation alerts
- Service/process monitoring - aggregating metrics and logs from distributed servers and applications (Event-driven SOA)
- Real-time analytics, fraud detection – processing of business/customer events in real time
- Click stream analytics, real-time predictive analytics
- Stock-trading analysis

Kafka Main Concepts

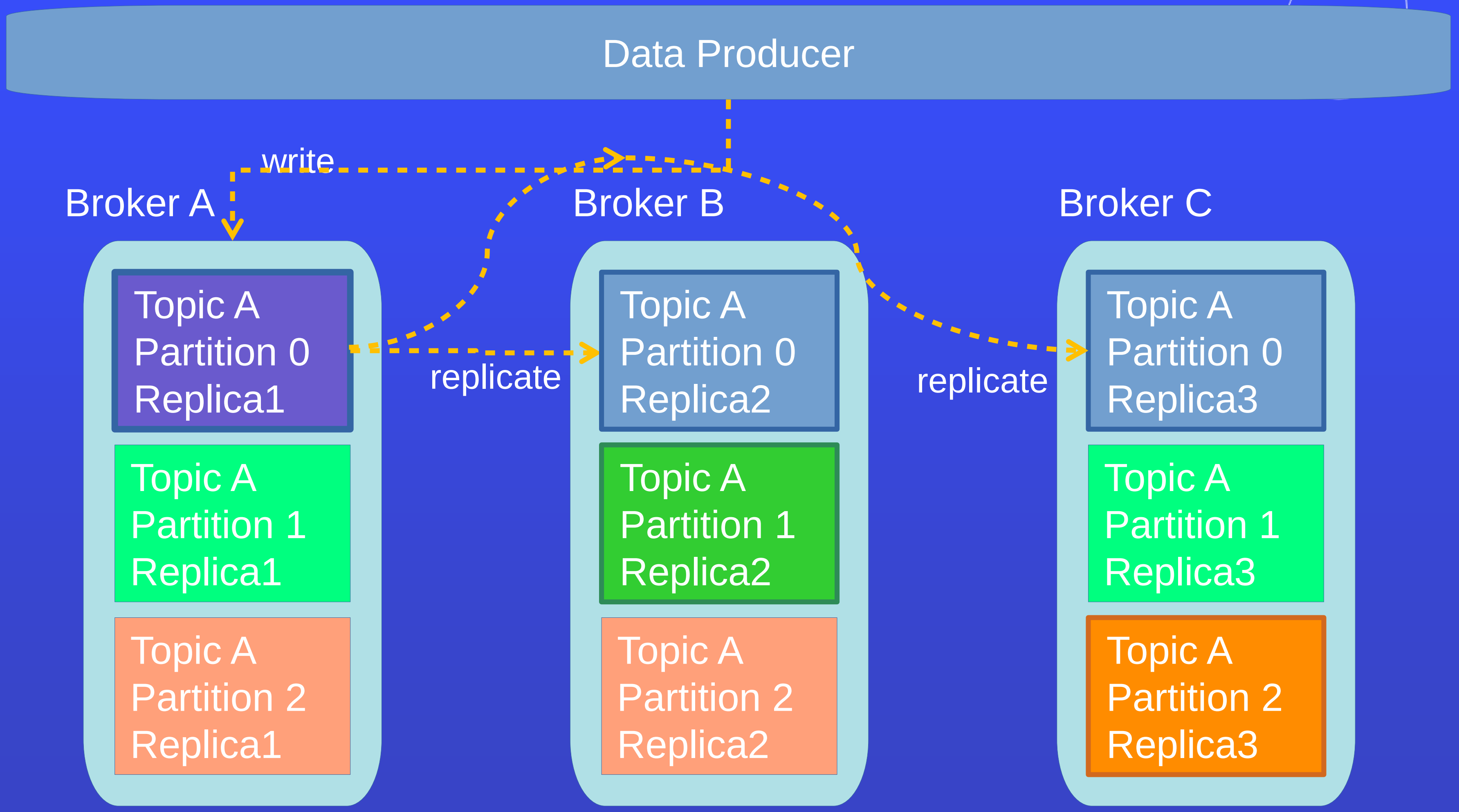
- Kafka is run as a cluster on one or more servers (brokers) that can span multiple datacenters.
- The Kafka cluster stores streams of records in categories called topics.
- Each record consists of a key, value, and timestamp.



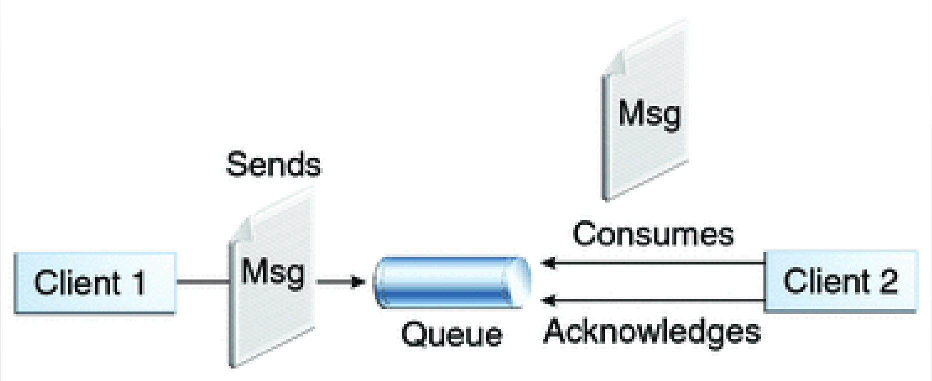
Kafka Core APIs

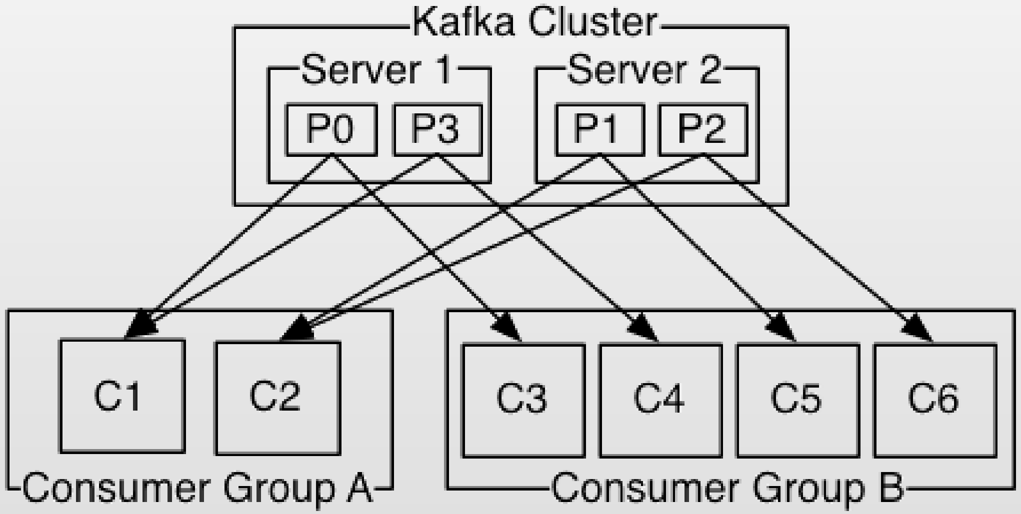
- Producer API - publish a stream of records to one or more Kafka topics.
- Consumer API - subscribe to one or more topics and process the stream of records produced to them.
- Streams API - a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.
- Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems – e.g. connector to a DB might capture every change in a table

Kafka Data Replication



Kafka as a Messaging System





https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Kafka Streams

- By combining storage and low-latency subscriptions, streaming applications can treat both past and future data the same way. That is a single application can process historical, stored data but rather than ending when it reaches the last record it can keep processing as future data arrives. This is a generalized notion of stream processing that subsumes batch processing as well as message-driven applications => Kappa architecture
- Likewise for streaming data pipelines the combination of subscription to real-time events make it possible to use Kafka for very low-latency pipelines; but the ability to store data reliably make it possible to use it for critical data where the delivery of data must be guaranteed or for integration with offline systems that load data only periodically or may go down for extended periods of time for maintenance.

Why you'll love using Kafka Streams?

- Elastic, highly scalable, fault-tolerant
- Deploy to containers, VMs, bare metal, cloud
- Equally viable for small, medium, & large use cases
- Fully integrated with Kafka security
- Write standard Java and Scala applications
- Exactly-once processing semantics
- No separate processing cluster required
- Develop on Mac, Linux, Windows

Kafka Streams Advantages - I

- Designed as a simple and lightweight client library, which can be easily embedded in any Java application and integrated with any existing packaging, deployment and operational tools that users have for their streaming applications.
- Has no external dependencies on systems other than Apache Kafka itself as the internal messaging layer; notably, it uses Kafka's partitioning model to horizontally scale processing while maintaining strong ordering guarantees.
- Supports fault-tolerant local state, which enables very fast and efficient stateful operations like windowed joins and aggregations.

Kafka Streams Advantages - II

- Supports exactly-once processing semantics to guarantee that each record will be processed once and only once even when there is a failure on either Streams clients or Kafka brokers in the middle of processing.
- Employs one-record-at-a-time processing to achieve millisecond processing latency, and supports event-time based windowing operations with out-of-order arrival of records.
- Offers necessary stream processing primitives, along with a high-level Streams DSL and a low-level Processor API.

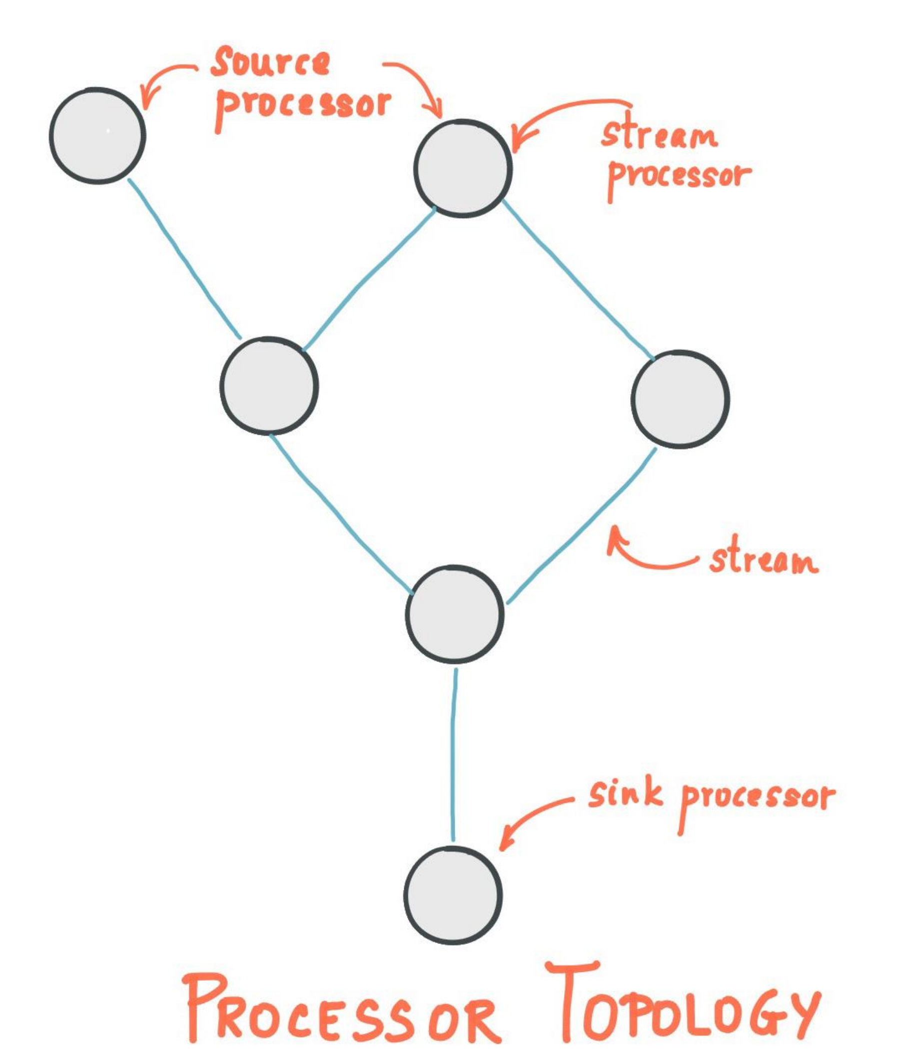
Stream Processing Topology - I

- A stream is the most important abstraction provided by Kafka Streams: it represents an unbounded, continuously updating data set. A stream is an ordered, replayable, and fault-tolerant sequence of immutable data records, where a data record is defined as a key-value pair.
- A stream processing application is any program that makes use of the Kafka Streams library. It defines its computational logic through one or more processor topologies, where a processor topology is a graph of stream processors (nodes) that are connected by streams (edges).
- A stream processor is a node in the processor topology; it represents a processing step to transform data in streams by receiving one input record at a time from its upstream processors in the topology, applying its operation to it, and may subsequently produce one or more output records to its downstream processors.

Types of Processors

- Source Processor: A source processor is a special type of stream processor that does not have any upstream processors. It produces an input stream to its topology from one or multiple Kafka topics by consuming records from these topics and forwarding them to its down-stream processors.
- Sink Processor: A sink processor is a special type of stream processor that does not have down-stream processors. It sends any received records from its up-stream processors to a specified Kafka topic.
- Note that in normal processor nodes other remote systems can also be accessed while processing the current record. Therefore the processed results can either be streamed back into Kafka or written to an external system.

Kafka Stream Processing - DAG



https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Следете актуалните обяви за **Java**

DEV.BG

Time in Kafka Streams

- A critical aspect in stream processing is the notion of time, and how it is modeled and integrated. For example, some operations such as windowing are defined based on time boundaries:
- Event time - the point in time when an event or data record occurred, i.e. was originally created "at the source".
- Processing time - the point in time when the event or data record happens to be processed by the stream processing application, i.e. when the record is being consumed.
- Ingestion time – time when an event or data record is stored in a topic partition by a Kafka broker. The difference to event time is that this ingestion timestamp is generated when the record is appended to the target topic by Kafka broker, not when the record is created at source.

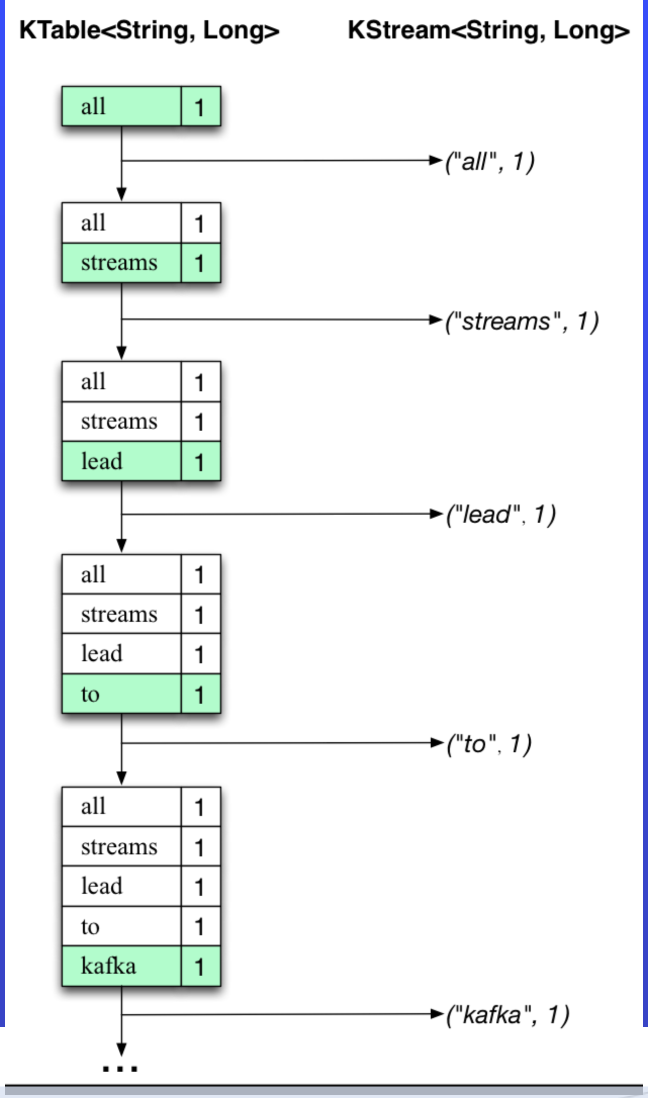
Time in Kafka Streams: Configuration

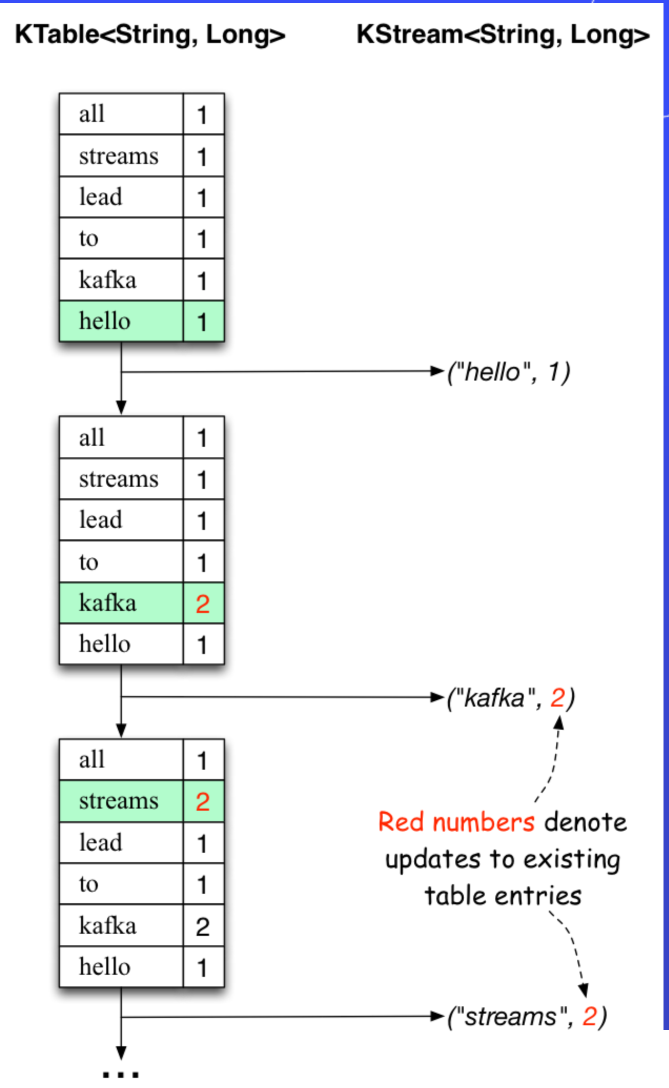
- **log.message.timestamp.type** – define whether the timestamp in the message is message create time or log append time. The value should be either `CreateTime` or `LogAppendTime`
- **log.message.timestamp.difference.max.ms** – The maximum difference allowed between the timestamp when a broker receives a message and the timestamp specified in the message. If log.message.timestamp.type=CreateTime, a message will be rejected if the difference in timestamp exceeds this threshold. This configuration is ignored if log.message.timestamp.type=LogAppendTime. The maximum timestamp difference allowed should be no greater than log.retention.ms to avoid unnecessarily frequent log rolling.

Custom TimestampExtractor

```java
@Slf4j
public class CustomTimeExtractor implements TimestampExtractor {
    @Override
    public long extract(ConsumerRecord<Object, Object> record, long partitionTime)  {
        final long timestamp = record.timestamp();

        // `TemperatureReading` is your own custom class, which we assume has a method that returns
        // the embedded timestamp (in milliseconds).
        var myReading = (TemperatureReading) record.value();
        if (myReading != null) {
            return java.sql.Timestamp.valueOf(myReading.getTimestamp()).getTime();
        }
        else {
            // Kafka allows `null` as message value.  How to handle such message values
            // depends on your use case.  In this example, we decide to fallback to
            // wall-clock time (= processing-time).
            return System.currentTimeMillis();
        }
    }
```

Kafka Stream Processing Example





https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Kafka Streams Dependencies

```
dependencies {
    implementation 'org.apache.kafka:kafka-clients:3.4.0'
    implementation 'org.apache.kafka:kafka-streams:3.4.0‘
    …
}
```

Kafka Streams Code Skeleton

```
public static void main(String[] args) {
        // Use the builders to define the actual processing topology, e.g. to specify from which input topics to read,
        // which stream operations (filter, map, etc.) should be called, and so on.

        StreamsBuilder builder = ...;  // when using the DSL
        Topology topology = builder.build();
        //
        // OR
        //
        Topology topology = ...; // when using the Processor API

        // Use the configuration to tell your application where the Kafka cluster is,
        // which Serializers/Deserializers to use by default, to specify security settings,and so on.
        Properties props = ...;
        KafkaStreams streams = new KafkaStreams(topology, props);
        // Add shutdown hook to stop the Kafka Streams threads. You can optionally provide a timeout to `close`.
        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}
```

Stream Partitions and Tasks

- Kafka messaging layer partitions data for storing and transporting it.
- Kafka Streams partitions data for processing it.
- In both cases, this partitioning is what enables data locality, elasticity, scalability, high performance, and fault tolerance. Kafka Streams uses the concepts of partitions and tasks as logical units of its parallelism model based on Kafka topic partitions.
- Each stream partition is a totally ordered sequence of data records and maps to a Kafka topic partition.
- A data record in the stream maps to a Kafka message from that topic.
- The keys of data records determine the partitioning of data in both Kafka and Kafka Streams -how data is routed to specific topic partitions.

https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Следете актуалните обяви за Java

DEV.BG

Stream Partitions and Tasks - II

- An application's processor topology is scaled by breaking it into multiple tasks.
- Kafka Streams creates a fixed number of tasks based on the input stream partitions for the application, with each task assigned a list of partitions from the input streams (i.e., Kafka topics).
- The assignment of partitions to tasks never changes so that each task is a fixed unit of parallelism of the application.
- Tasks can then instantiate their own processor topology based on the assigned partitions; they also maintain a buffer for each of its assigned partitions and process messages one-at-a-time from these record buffers.
- As a result stream tasks can be processed independently and in parallel without manual intervention.

Stream Partitions and Tasks - III

- Kafka Streams is NOT a resource manager, but a library that "runs" anywhere its stream processing application runs.
- Multiple instances of the application are executed either on the same machine, or spread across multiple machines and tasks can be distributed automatically by the library to those running application instances.
- Assignment of partitions to tasks never changes - if an application instance fails, all its assigned tasks will be automatically restarted on other instances and continue to consume from the same stream partitions.
- Topic partitions are assigned to tasks, and tasks are assigned to all threads over all instances, in a best-effort attempt to trade off load-balancing and stickiness of stateful tasks. For this assignment, Kafka Streams uses the StreamsPartitionAssignor class.

StreamsPartitionAssignor Tasks Assignment

1. Decode the subscriptions to assemble the metadata for each client and check for version probing.
2. Check all repartition source topics and use internal topic manager to make sure they have been created with the right number of partitions. Also verify and/or create any changelog topics with the correct number of partitions.
3. Use the partition grouper to generate tasks along with their assigned partitions, then use the configured TaskAssignor to construct the mapping of tasks to clients.
4. Construct the global mapping of host to partitions to enable query routing.
5. Within each client, assign tasks to consumer clients.

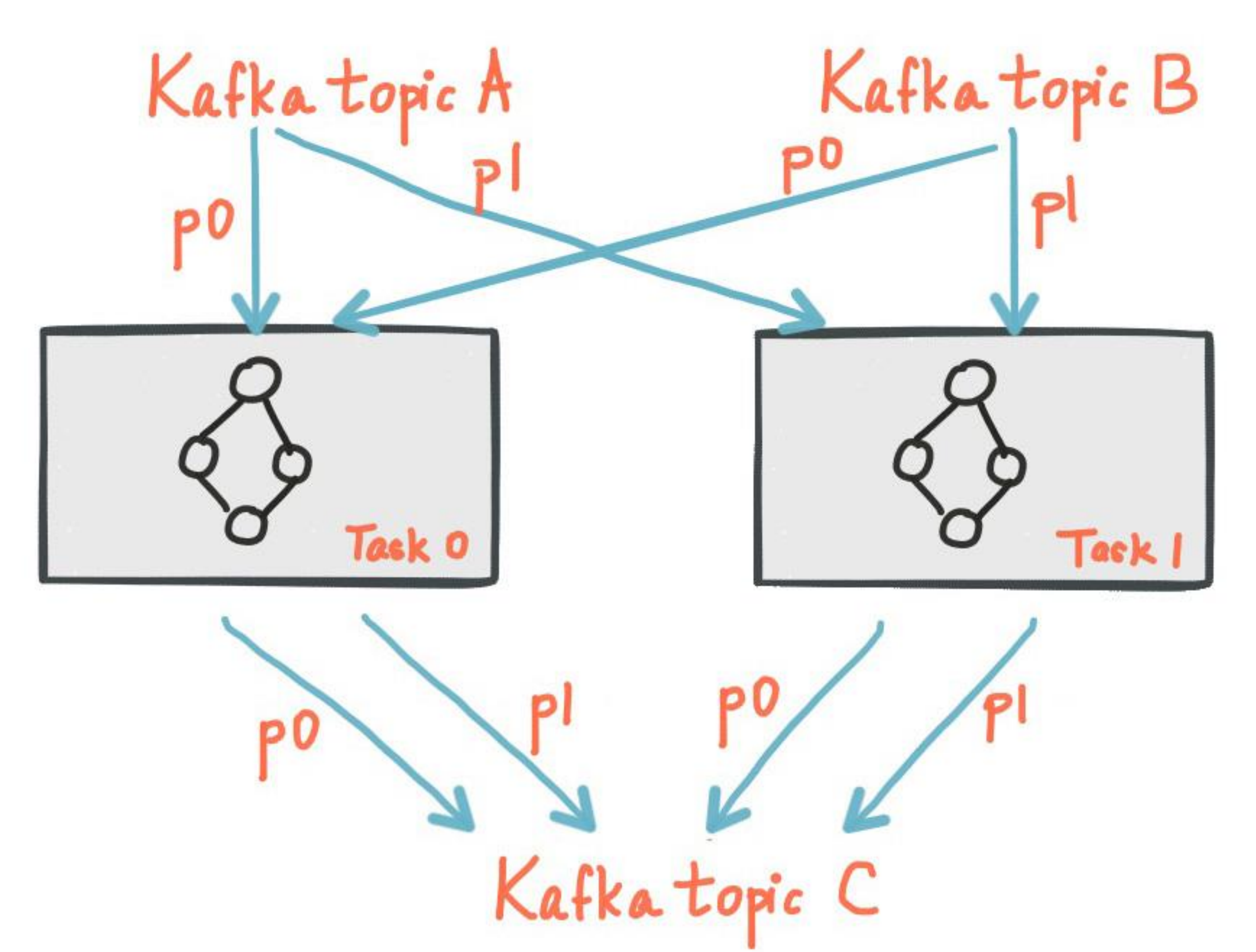
https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Следете актуалните обяви за Java

DEV.BG

Kafka Streams Partitions and Tasks - I



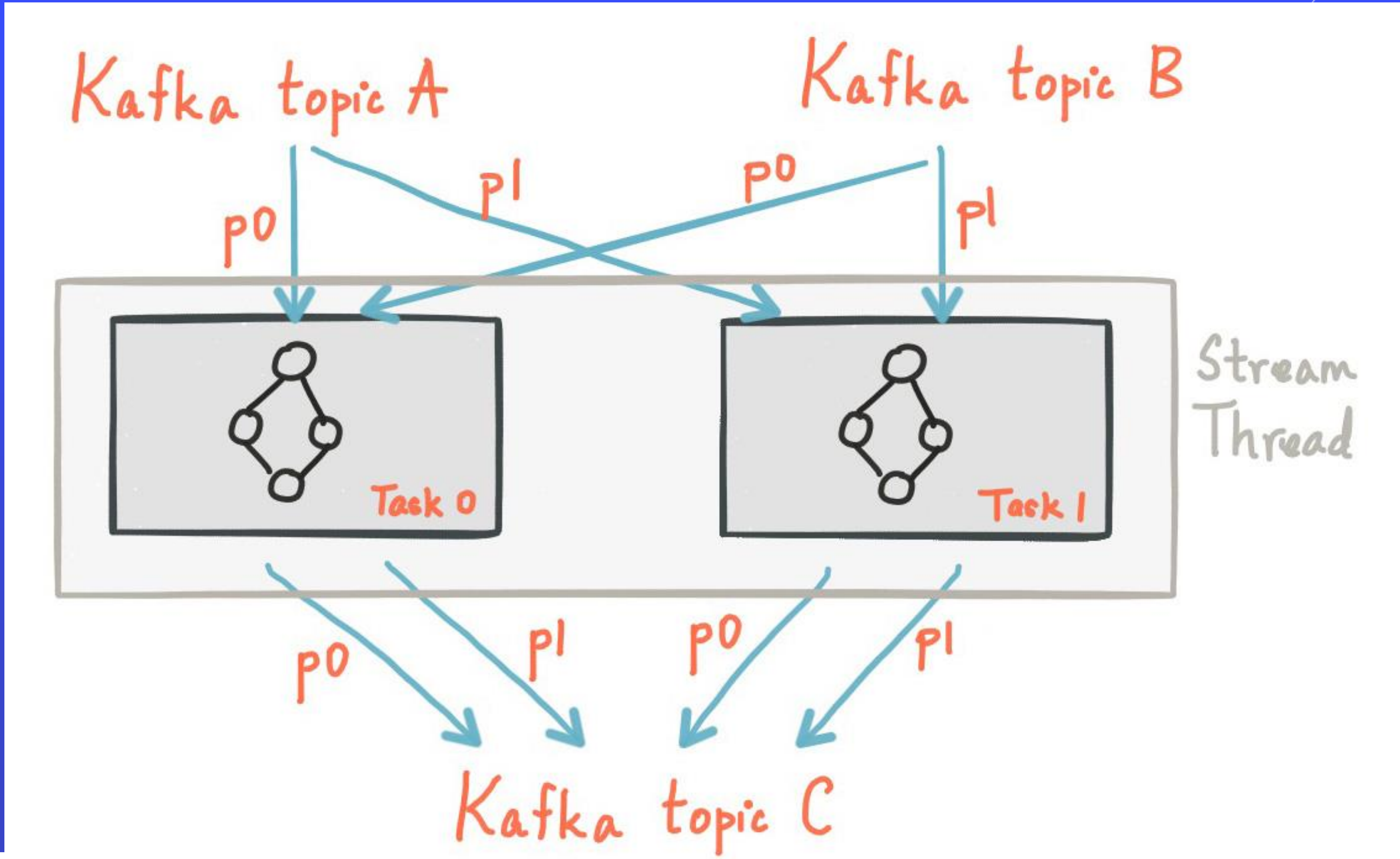
https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Следете актуалните обяви за Java

DEV.BG

Kafka Streams Partitions and Tasks - II



https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

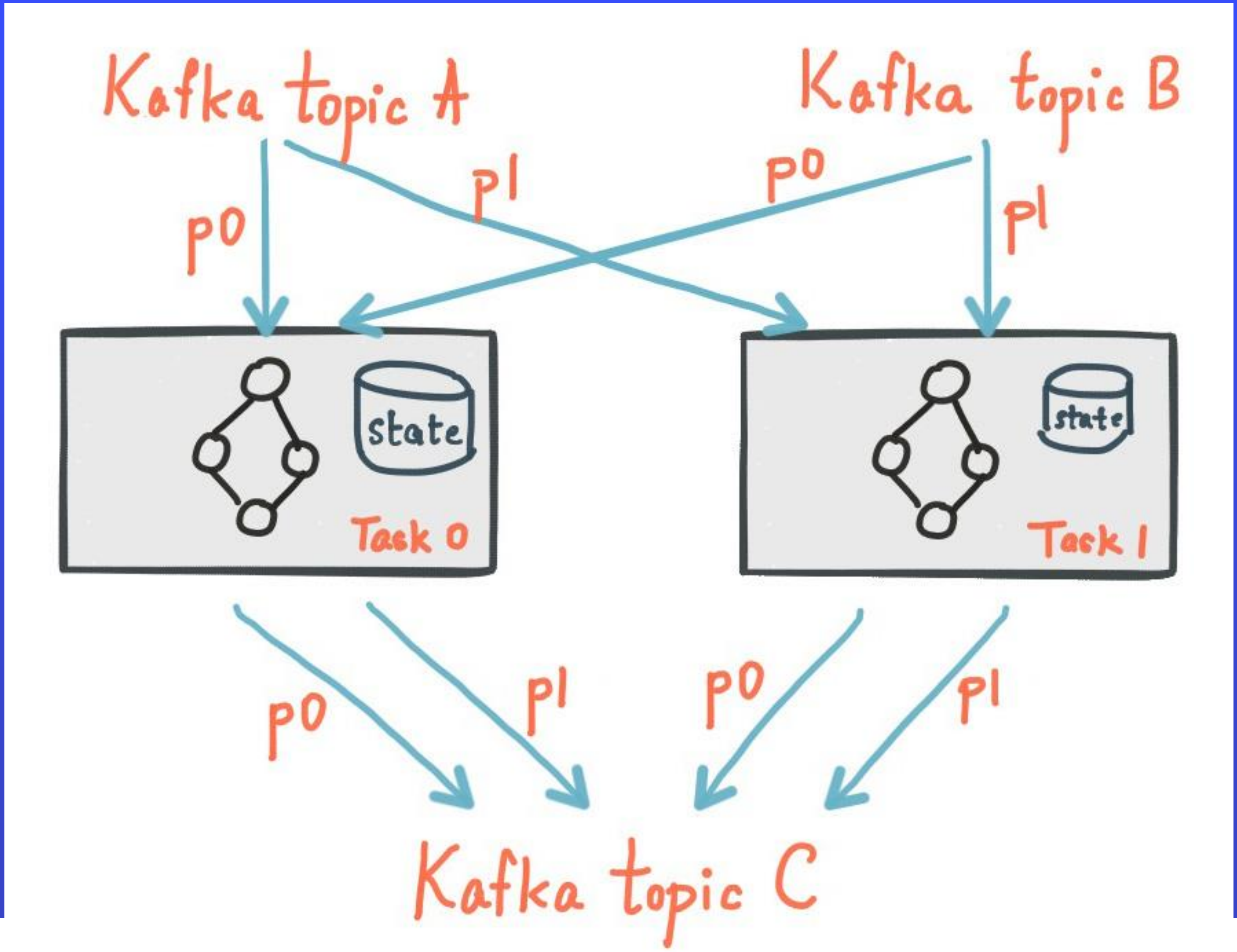
Следете актуалните обяви за Java

DEV.BG

Tasks Threading Model

- Starting more stream threads or more instances of the application merely amounts to replicating the topology and having it process a different subset of Kafka partitions, effectively parallelizing processing.
- It is worth noting that there is **no shared state amongst the threads**, so **no inter-thread coordination is necessary**.
- This makes it very simple to run topologies in parallel across the application instances and threads.
- The assignment of Kafka topic partitions amongst the various stream threads is transparently handled by Kafka Streams + Kafka coordination.
- You can start as many threads of the application as there are input topic partitions so that, across all running instances of an application, every thread (or rather, the tasks it runs) has at least one input partition to process.

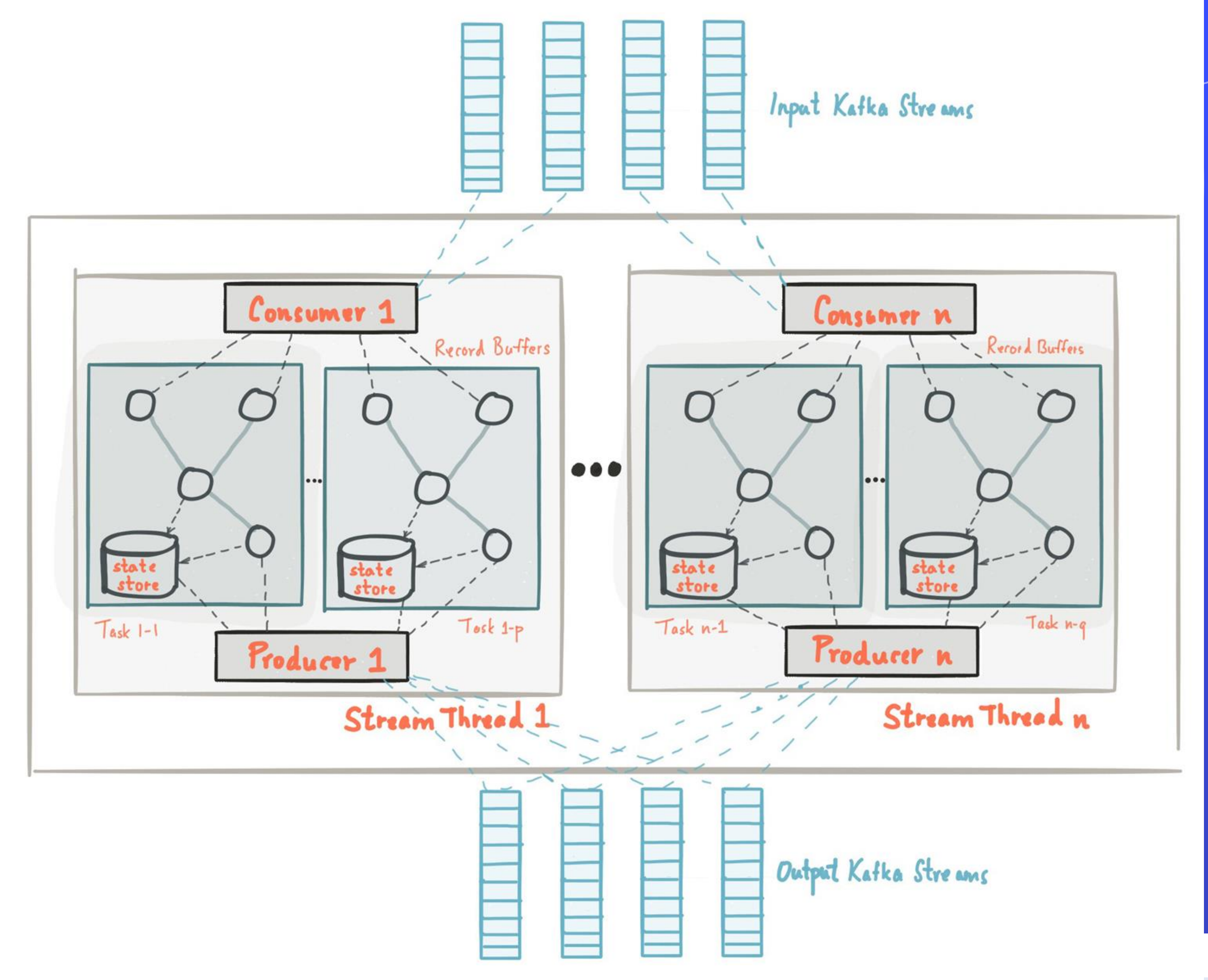
Kafka Streams Partitions and Tasks - III



https://kafka.apache.org/documentation/streams/
© 2017 Apache Software Foundation under the terms of the Apache License v2

Следете актуалните обяви за Java

DEV.BG



https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Kafka Streams DSL & Processor API

- Processor API - allows developers to define and connect custom processors and to interact with state stores. With the Processor API, you can define arbitrary stream processors that process one received record at a time, and connect these processors with their associated state stores to compose the processor topology that represents a customized processing logic.
- Processor API can be used to implement both stateless as well as stateful operations, where the latter is achieved through the use of state stores.
- Kafka Streams DSL (Domain Specific Language) is built on top of the Streams Processor API. It is the recommended for most users, especially beginners. Most data processing operations can be expressed in just a few lines of DSL code.
- Combining the DSL and the Processor API – you can combine the convenience of the DSL with the power and flexibility of the Processor API as described in [Applying processors and transformers (Processor API integration)](#).

https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Kafka Streams DSL: KStreams

- Only the Kafka Streams DSL has the notion of a KStream.
- KStream is an abstraction of a record stream, where each data record represents a self-contained datum in the unbounded data set. Using the table analogy, data records in a record stream are always interpreted as an "INSERT" -- thing: adding more entries to an append-only ledger -- because no record replaces an existing row with the same key. Examples are a credit card transaction, a page view event, or a server log entry.
- To illustrate, let's imagine the following two data records are being sent to the stream: **("alice", 1)** --> **("alice", 3)**
- If your stream processing application were to sum the values per user, it would return **4** for **alice**. Why? Because the second data record would not be considered an update of the previous record. Compare this behavior of KStream to KTable in next slide, which would return **3** for **alice**.

Kafka Streams DSL: KTables

- Only the Kafka Streams DSL has the notion of a KTable.
- KTable is an abstraction of a changelog stream, where each data record represents an update. More precisely, the value in a data record is interpreted as an "UPDATE" of the last value for the same record key, if any (if a corresponding key doesn't exist yet, the update will be considered an INSERT). Using the table analogy, a data record in a changelog stream is interpreted as an UPSERT aka INSERT/UPDATE because any existing row with the same key is overwritten. Also, null values (tombstones) are interpreted in a special way: a record with a null value represents a "DELETE" or tombstone for the record's key.
- To illustrate, let's imagine the following two data records are being sent to the stream: **("alice", 1)** --> **("alice", 3)**
- If a stream processing application is summing the values per user, it would return **3** for **alice**. Why? Second record would be considered update previous.

KTables and Log Compaction

- Another way of thinking about KStream and KTable is as follows: If you were to store a KTable into a Kafka topic, you'd probably want to enable Kafka's log compaction feature, e.g. to save storage space.
- However, it would not be safe to enable log compaction in the case of a KStream because, as soon as log compaction would begin purging older data records of the same key, it would break the semantics of the data. E.g. you'd suddenly get a 3 for alice instead of a 4 because of log compaction. Hence log compaction is perfectly safe for a KTable (**changelog stream**) but it is a mistake for a KStream (**record stream**).
- Example: **Change Data Capture (CDC)** records in the changelog of a relational DB, representing which row in database table was inserted/ updated/ deleted.
- KTable also provides an ability to look up current values of data records by keys. Table-lookup is available through join operations & Interactive Queries.

https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Kafka Streams DSL: GlobalKTable

- GlobalKTable is an abstraction of a changelog stream, where each data record represents an update.
- GlobalKTable differs from a KTable in the data that they are being populated with, i.e. which data from the underlying Kafka topic is being read into the respective table. Slightly simplified, imagine you have an input topic with 5 partitions. In your application, you want to read this topic into a table. You want to run your application across 5 application instances for maximum parallelism.
- If input topic read into a KTable, then "local" KTable instance of each application instance will be populated with data from only 1 partition of the topic 5 partitions.
- If input topic read into a GlobalKTable, then the local GlobalKTable instance of each application instance will be populated with data from all topic.
- GlobalKTable provides the ability to look up current values of data records by keys. This table-lookup functionality is available through join operations. Note that a GlobalKTable has no notion of time in contrast to a KTable.

https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Следете актуалните обяви за Java

DEV.BG

Benefits and Downsides of Using GlobalKTable

- Benefits:
  - More convenient and/or efficient joins: Notably, global tables allow you to perform star joins, they support "foreign-key" lookups (i.e., you can lookup data in the table not just by record key, but also by data in the record values), and they are more efficient when chaining multiple joins. Also, when joining against a global table, the input data does not need to be co-partitioned.
  - Can be used to "broadcast" information to all the running instances of your application.
- Downsides of global tables:
  - Increased local storage consumption compared to the (partitioned) KTable because the entire topic is tracked.
  - Increased network and Kafka broker load compared to the (partitioned) KTable because the entire topic is read.

Streams DSL: Creating a Stream

```
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.kstream.Consumed;
import org.apache.kafka.streams.kstream.KStream;

public class Temp {
    public static void main(String[] args) {
        StreamsBuilder builder = new StreamsBuilder();
        KStream<String, Long> wordCounts = builder.stream(
                "word-counts-input-topic", /* input topic */
                Consumed.with(
                        Serdes.String(), /* key serde */
                        Serdes.Long()   /* value serde */
                ));
    }
}
```

Streams DSL: Creating GlobalKTable

```java
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.common.utils.Bytes;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.kstream.GlobalKTable;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.state.KeyValueStore;

public class Temp {
    public static void main(String[] args) {
        StreamsBuilder builder = new StreamsBuilder();
        GlobalKTable<String, Long> wordCounts = builder.globalTable(
                "word-counts-input-topic",
                Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>as(
                                "word-counts-global-store" /* table/store name */)
                        .withKeySerde(Serdes.String()) /* key serde */
                        .withValueSerde(Serdes.Long()) /* value serde */
        );
    }
}
```

Streams DSL KStream and KTable Transformations

- KStream is an abstraction of a record stream of KeyValue pairs, i.e., each record is an independent entity/event in the real world. For example a user X might buy two items I1 and I2, and thus there might be two records <K:I1>, <K:I2> in the stream.
- A KStream is either defined from one or multiple Kafka topics that are consumed message by message, or the result of a KStream transformation.
- A KTable can also be converted into a KStream.
- A KStream can be **transformed** record by record, **joined** with another KStream, KTable, GlobalKTable, or can be **aggregated** into a KTable. Kafka Streams DSL can be mixed-and-matched with Processor API (PAPI) (c.f. Topology) via process(...), transform(...), and transformValues(...).

https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Следете актуалните обяви за Java

DEV.BG

Processor API (PAPI) Example - I

```
public class WordCountProcessor implements Processor<String, String, String, String> {
    private KeyValueStore<String, Long> kvStore;
    private ProcessorContext<String, String> context;

    @Override
    public void init(ProcessorContext<String, String> context) {
        this.context = context;
        kvStore = context.getStateStore("inmemory-word-counts");
    }

   @Override
    public void close() {
    }
```

Processor API (PAPI) Example - II

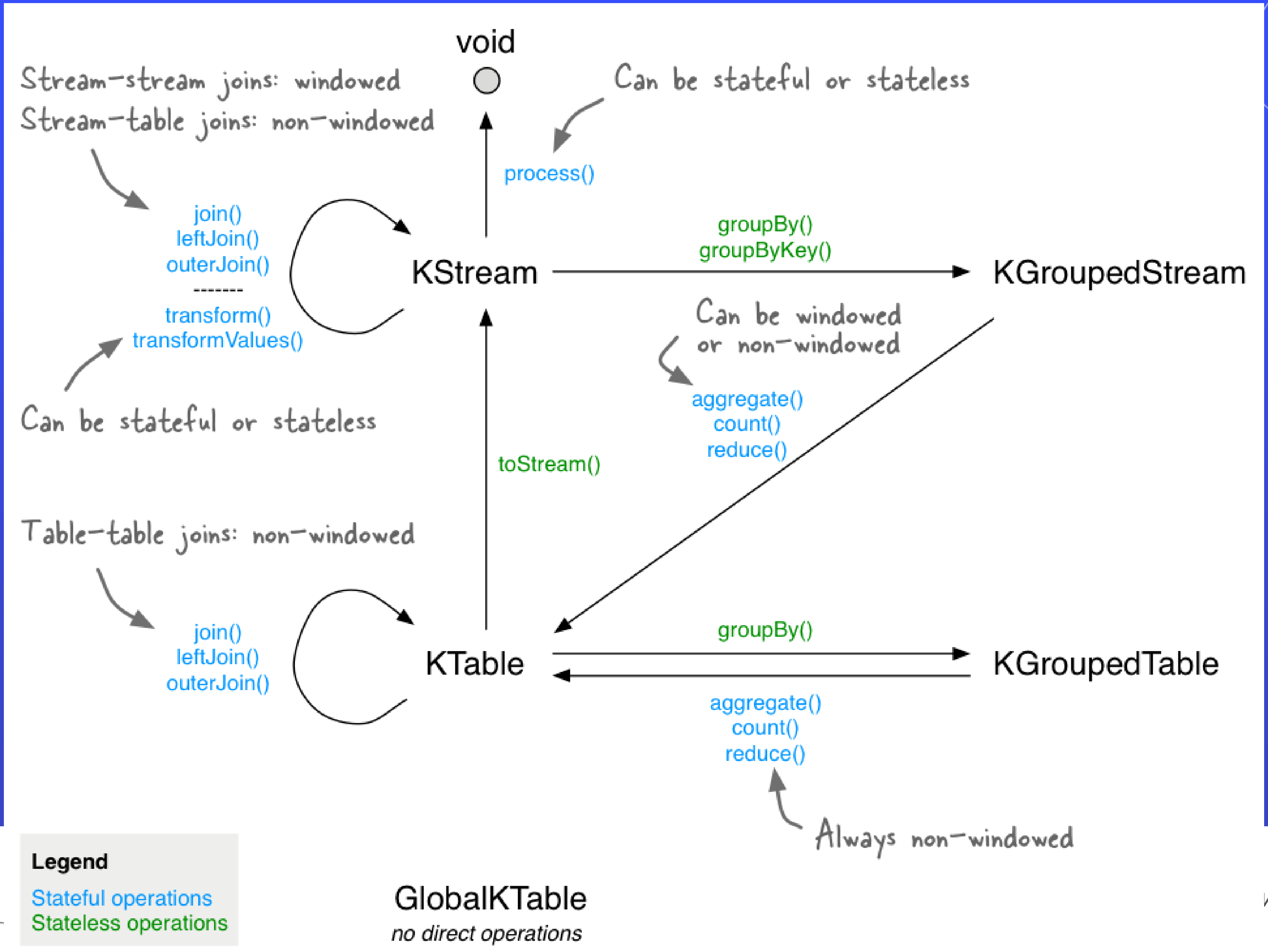
```
@Override
    public void process(Record<String, String> record) {
        final String[] words = record.value().toLowerCase().split("\\W+");

        for (final String word : words) {
            Long oldVal = kvStore.get(word);
            if (oldVal == null) {
                oldVal = 0L;
            }
            kvStore.put(word, oldVal + 1);
            context.forward(new Record<>(
                    word,
                    String.format("%-15s -> %4d", word, oldVal + 1),
                    record.timestamp()
            ));
        }
    }
}
```

Streams DSL: Stateless Transformations

Stateless transformations do not require state for processing and they do not require a state store associated with the stream processor. Kafka allows you to materialize the result from a stateless KTable transformation. This allows the result to be queried through interactive queries. To materialize a KTable, each of stateless operations can be augmented with an optional queryableStoreName argument:

Branch: KStream → BranchedKStream
Filter: KStream → Kstream, **Filter**: KTable → Ktable
Inverse Filter filterNot: KStream → Kstream, **filterNot**: KTable → Ktable
FlatMap: KStream → Kstream, **FlatMap (values only)**: KStream → Kstream
Foreach: KStream → void | KStream → void | KTable → void
GroupByKey: KStream → KGroupedStream, **GroupBy**: KStream → KGroupedStream

Streams DSL: Stateful Transformations



Streams DSL: Stateful Transformations

- Stateful transformations depend on state for processing inputs and producing outputs and require a state store associated with the stream processor. In aggregating operations, a windowing state store is used to collect the latest aggregation results per window. In join operations, a windowing state store is used to collect all of records received within the defined window boundary.
- **non-windowed** aggregations and non-windowed KTables use TimestampedKeyValueStores
- **time-windowed** aggregations and KStream-KStream joins use TimestampedWindowStores
- **session windowed** aggregations use SessionStores (there is no timestamped session store as of now)
- **State stores** are fault-tolerant. In case of failure, Kafka Streams guarantees to fully restore all state stores prior to resuming the processing.

Streams DSL: Stateful Transformations

- Stateful transformations depend on state for processing inputs and producing outputs and require a state store associated with the stream processor. In aggregating operations, a windowing state store is used to collect the latest aggregation results per window. In join operations, a windowing state store is used to collect all of records received within the defined window boundary.
- **non-windowed** aggregations and non-windowed KTables use TimestampedKeyValueStores
- **time-windowed** aggregations and KStream-KStream joins use TimestampedWindowStores
- **session windowed** aggregations use SessionStores (there is no timestamped session store as of now)
- **State stores** are fault-tolerant. In case of failure, Kafka Streams guarantees to fully restore all state stores prior to resuming the processing.

Types of Stateful Transformations

Available stateful transformations in the DSL include:

- Aggregating
- Joining
- Windowing (as part of aggregations and joins)
- Applying custom processors and transformers, which may be stateful, for Processor API integration

Aggregating

- After records are grouped by key via groupByKey or groupBy – and thus represented as either a KGroupedStream or a KGroupedTable, they can be aggregated via an operation such as reduce. Aggregations are key-based operations, which means that they always operate over records (notably record values) of the same key. You can perform aggregations on windowed or non-windowed data.
- Types of windows:

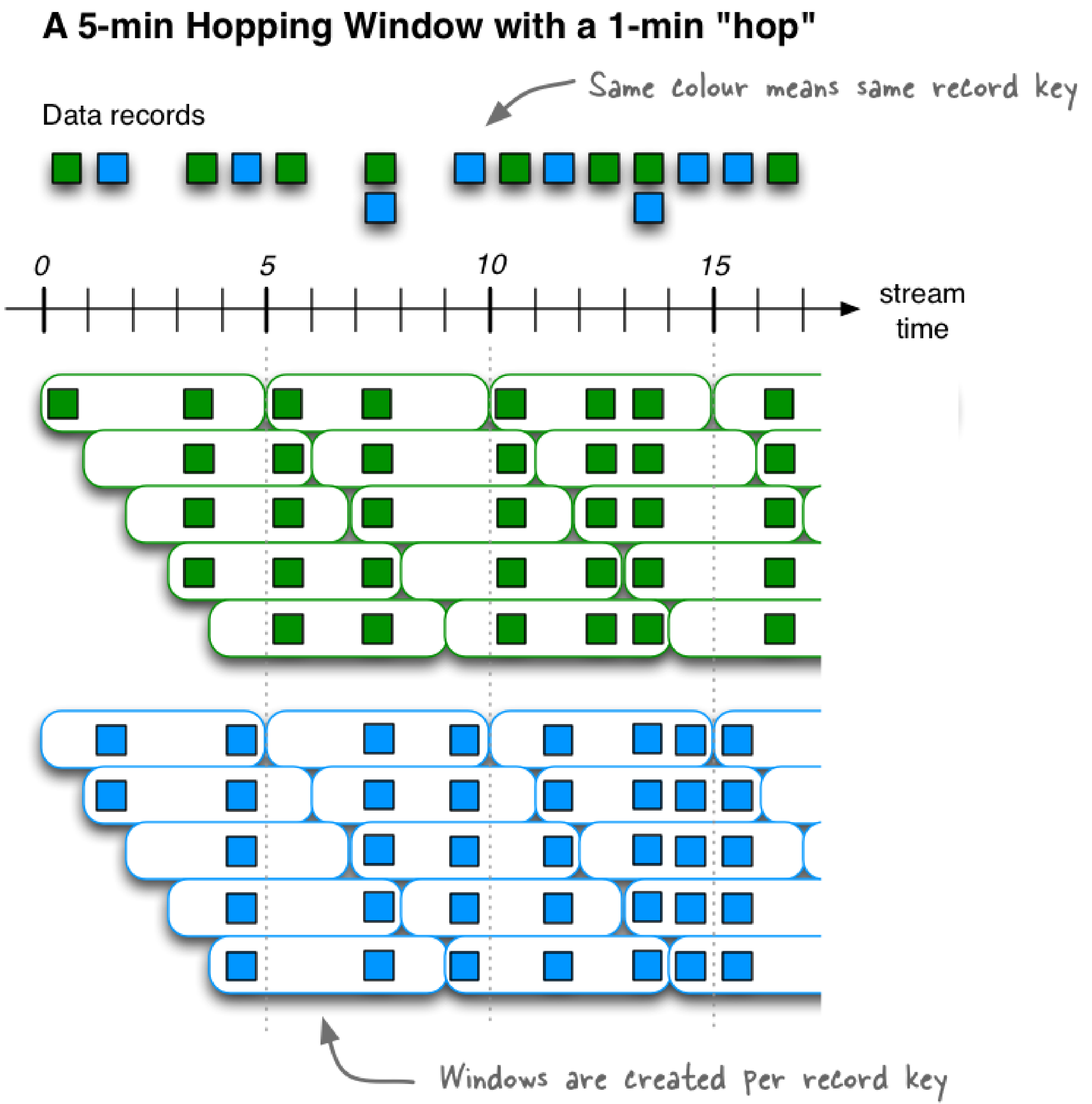
Window name	Behavior	Short description
Hopping time window	Time-based	Fixed-size, overlapping windows
Tumbling time window	Time-based	Fixed-size, non-overlapping, gap-less windows
Sliding time window	Time-based	Fixed-size, overlapping windows that work on differences between record timestamps
Session window	Session-based	Dynamically-sized, non-overlapping, data-driven windows

https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Hopping Windows

```java
import java.time.Duration;
import org.apache.kafka.streams.kstream.TimeWindows;

    // A hopping time window with a size of 5 minutes and an advance interval of 1 min.
    // The window's name -- the string parameter -- is used to e.g. name the backing state store.
    Duration windowSize = Duration.ofMinutes(5);
    Duration advance = Duration.ofMinutes(1);
    TimeWindows.ofSizeWithNoGrace(windowSize).advanceBy(advance);
```



https://kafka.apache.org/documentation/streams/

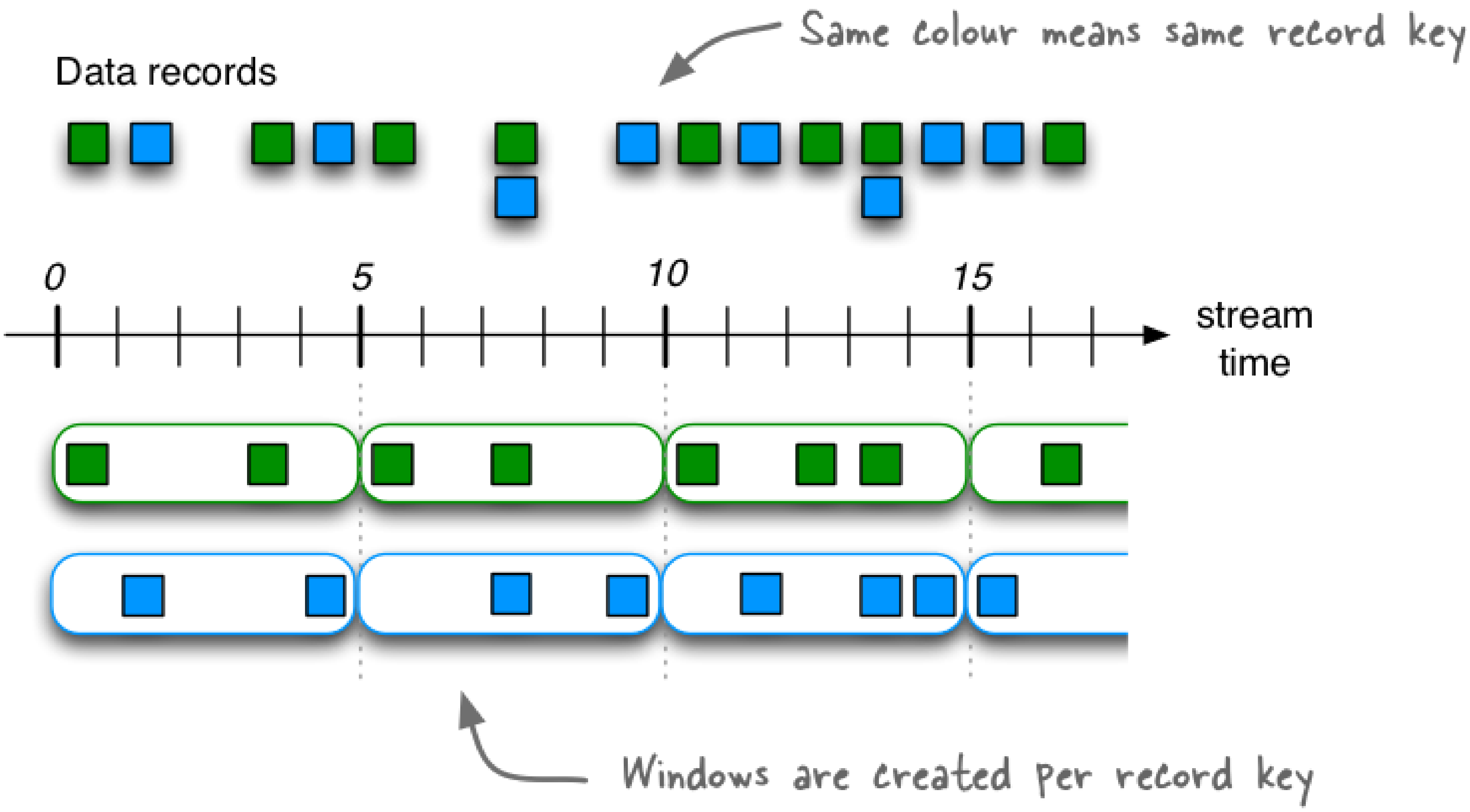
© 2017 Apache Software Foundation under the terms of the Apache License v2

Следете актуалните обяви за **Java**

Tumbling Time Windows

```java
import java.time.Duration;
import org.apache.kafka.streams.kstream.TimeWindows;

// A tumbling time window with a size of 5 minutes (and, by definition, an implicit
// advance interval of 5 minutes), and grace period of 1 minute.
Duration windowSize = Duration.ofMinutes(5);
Duration gracePeriod = Duration.ofMinutes(1);
TimeWindows.ofSizeAndGrace(windowSize, gracePeriod);

// The above is equivalent to the following code:
TimeWindows.ofSizeAndGrace(windowSize, gracePeriod).advanceBy(windowSize);
```

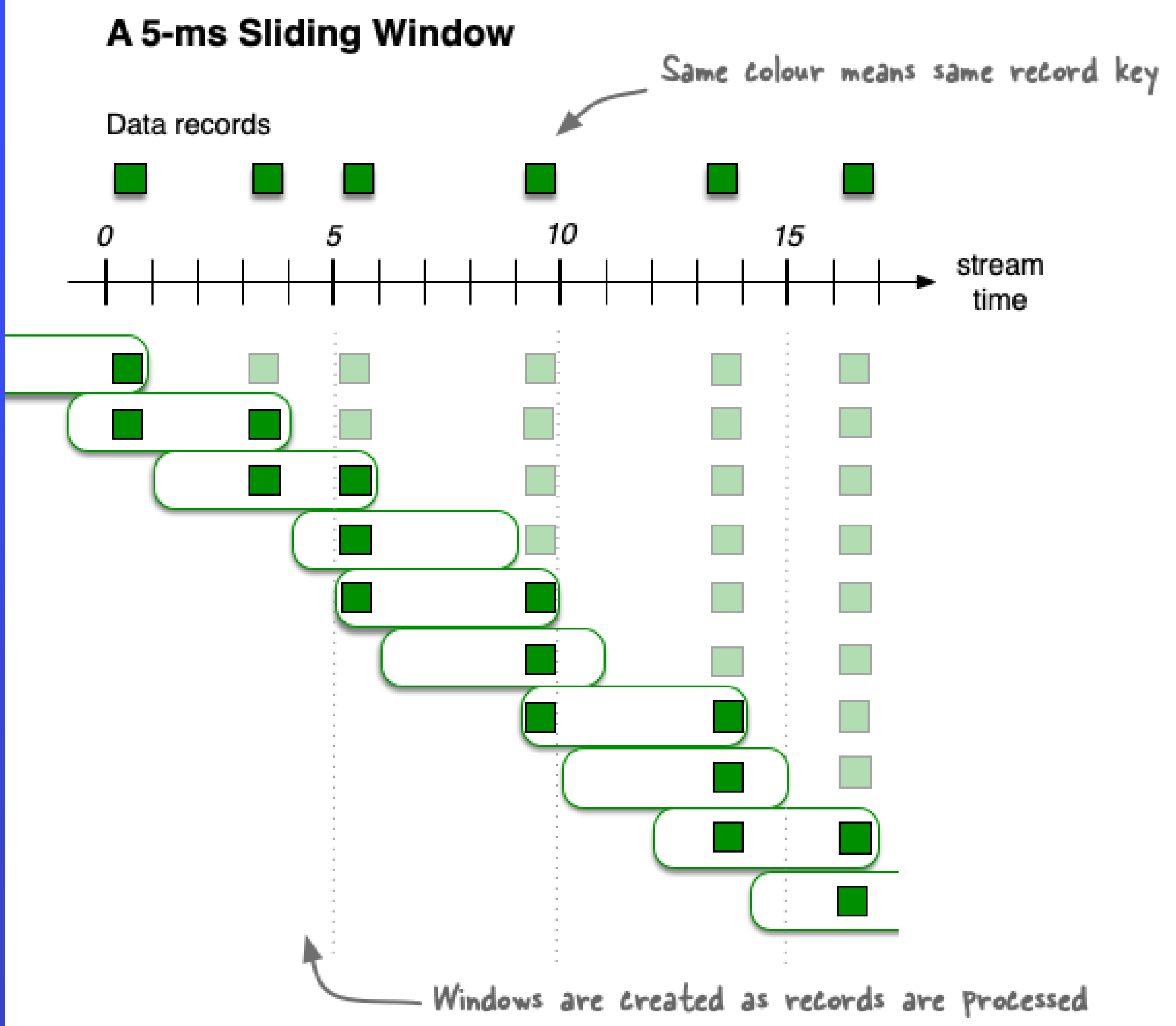
A 5-min Tumbling Window



© 2017 Apache Software Foundation under the terms of the Apache License v2
Следете актуалните обяви за Java
DEV.BG

Sliding Time Windows

```
import org.apache.kafka.streams.kstream.SlidingWindows;

// A sliding time window with a time difference of 10 minutes and grace period of 30 minutes
Duration timeDifference = Duration.ofMinutes(10);
Duration gracePeriod = Duration.ofMinutes(30);
SlidingWindows.ofTimeDifferenceAndGrace(timeDifference, gracePeriod);
```



https://kafka.apache.org/documentation/streams/
© 2017 Apache Software Foundation under the terms of the Apache License v2
Следете актуалните обяви за Java
DEV.BG

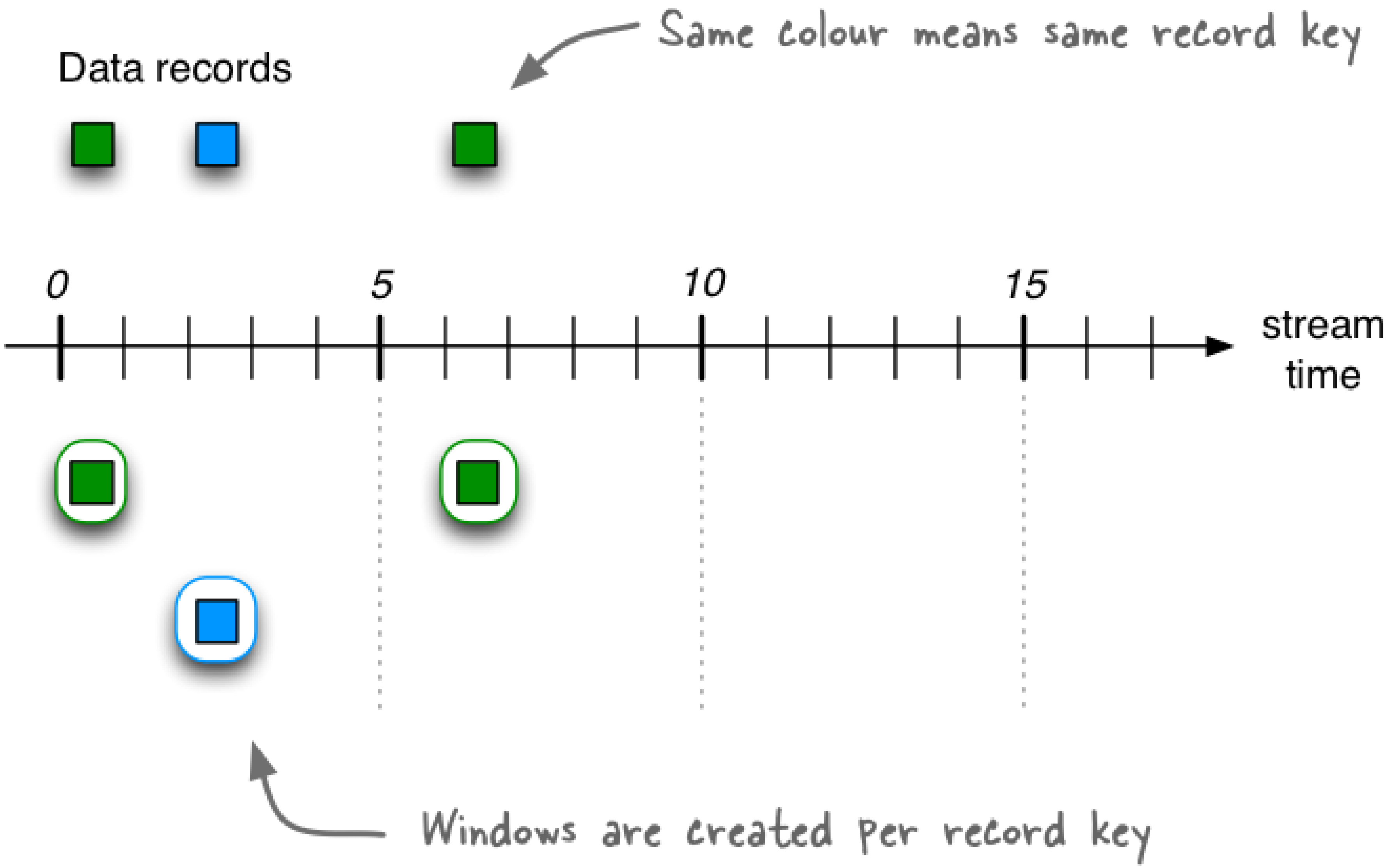
Session Windows

```
import java.time.Duration;
import org.apache.kafka.streams.kstream.SessionWindows;

// A session window with an inactivity gap of 5 minutes.
SessionWindows.ofInactivityGapWithNoGrace(Duration.ofMinutes(5));
```

Следете актуалните обяви за **Java**

DEV.BG

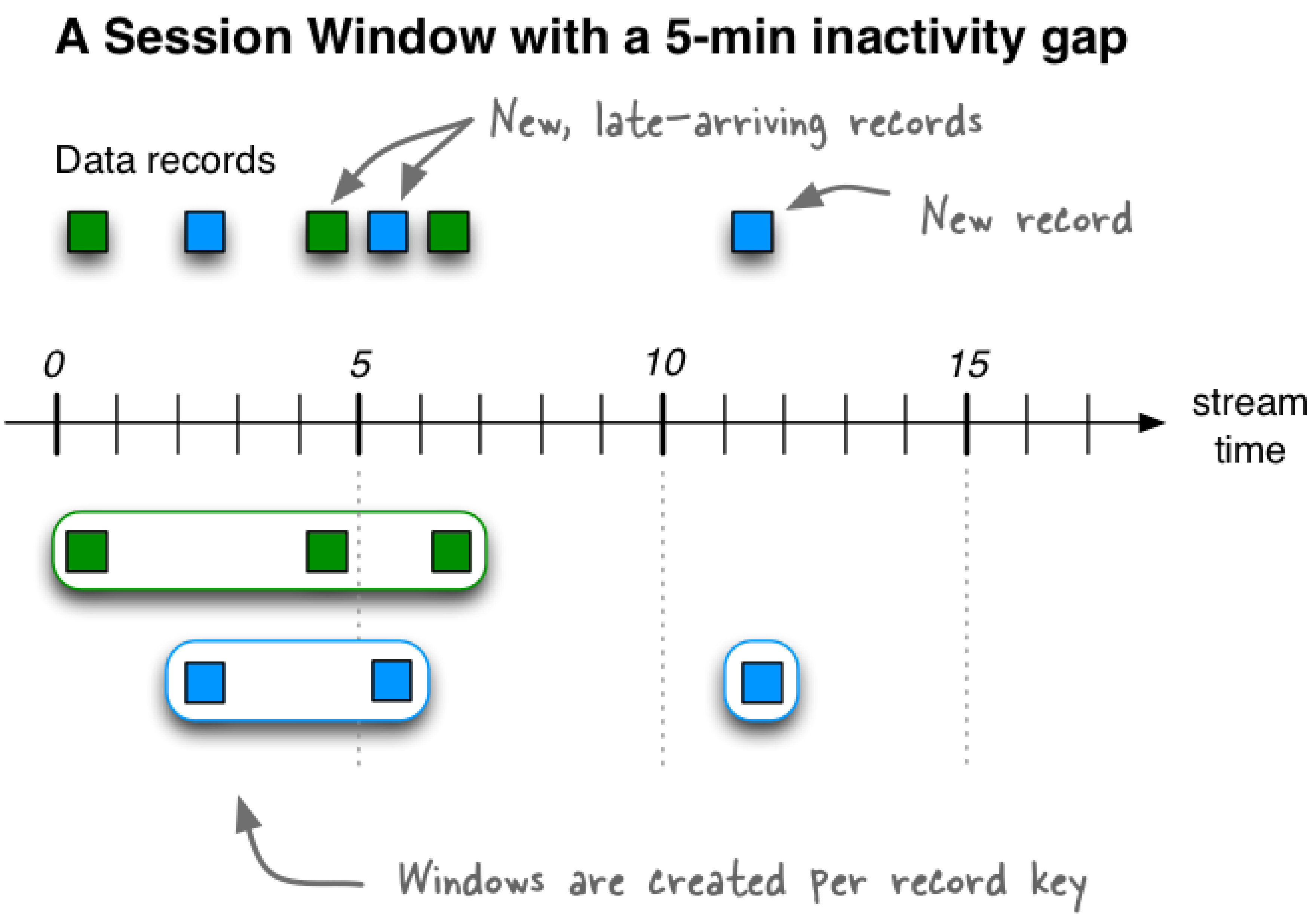
A Session Window with a 5-min inactivity gap



https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Следете актуалните обяви за Java
DEV.BG



https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Следете актуалните обяви за Java
DEV.BG

Window Final Results

- In Kafka Streams, windowed computations update their results continuously. As new data arrives for a window, freshly computed results are emitted downstream.
- However, some applications need to take action only on the final result of a windowed computation. Common examples of this are sending alerts or delivering results to a system that doesn't support updates.

```
KGroupedStream<UserId, Event> grouped = ...;
grouped
    .windowedBy(TimeWindows.ofSizeAndGrace(Duration.ofHours(1), Duration.ofMinutes(10)))
    .count()
    .suppress(Suppressed.untilWindowCloses(unbounded()))
    .filter((windowedUserId, count) -> count < 3)
    .toStream()
    .foreach((windowedUserId, count) -> sendAlert(windowedUserId.window(), windowedUserId.key(), count));
```

Следете актуалните обяви за **Java**

DEV.BG

Controlling KTable Emit Rate

- some applications need to take other actions, such as calling out to external systems, and therefore need to exercise some control over the rate of invocations, for example of KStream#foreach.
- Rather than achieving this as a side-effect of the KTable record cache, you can directly impose a rate limit via the KTable#suppress operator.

```
KGroupedTable<String, String> groupedTable = ...;
groupedTable
        .count()
        .suppress(untilTimeLimit(ofMinutes(5), maxBytes(1_000_000L).emitEarlyWhenFull()))
        .toStream()
        .foreach((key, count) -> updateCountsDatabase(key, count));
```

Joining

Join operands	Type	(INNER) JOIN	LEFT JOIN	OUTER JOIN
KStream-to-KStream	Windowed	Supported	Supported	Supported
KTable-to-KTable	Non-windowed	Supported	Supported	Supported
KTable-to-KTable Foreign-Key Join	Non-windowed	Supported	Supported	Not Supported
KStream-to-KTable	Non-windowed	Supported	Supported	Not Supported
KStream-to-GlobalKTable	Non-windowed	Supported	Supported	Not Supported
KTable-to-GlobalKTable	N/A	Not Supported	Not Supported	Not Supported

https://kafka.apache.org/documentation/streams/

© 2017 Apache Software Foundation under the terms of the Apache License v2

Join Co-partitioning Requirements

- For equi-joins, input data must be co-partitioned when joining. This ensures that input records with the same key from both sides of the join, are delivered to the same stream task during processing.
- Co-partitioning is not required when performing KTable-KTable Foreign-Key joins and Global KTable joins.
- The input topics of the join (left side and right side) must have the same number of partitions.
- All applications that write to the input topics must have the same partitioning strategy so that records with the same key are delivered to same partition number. In other words, the keyspace of the input data must be distributed across partitions in the same manner.

Join Co-partitioning Requirements - II

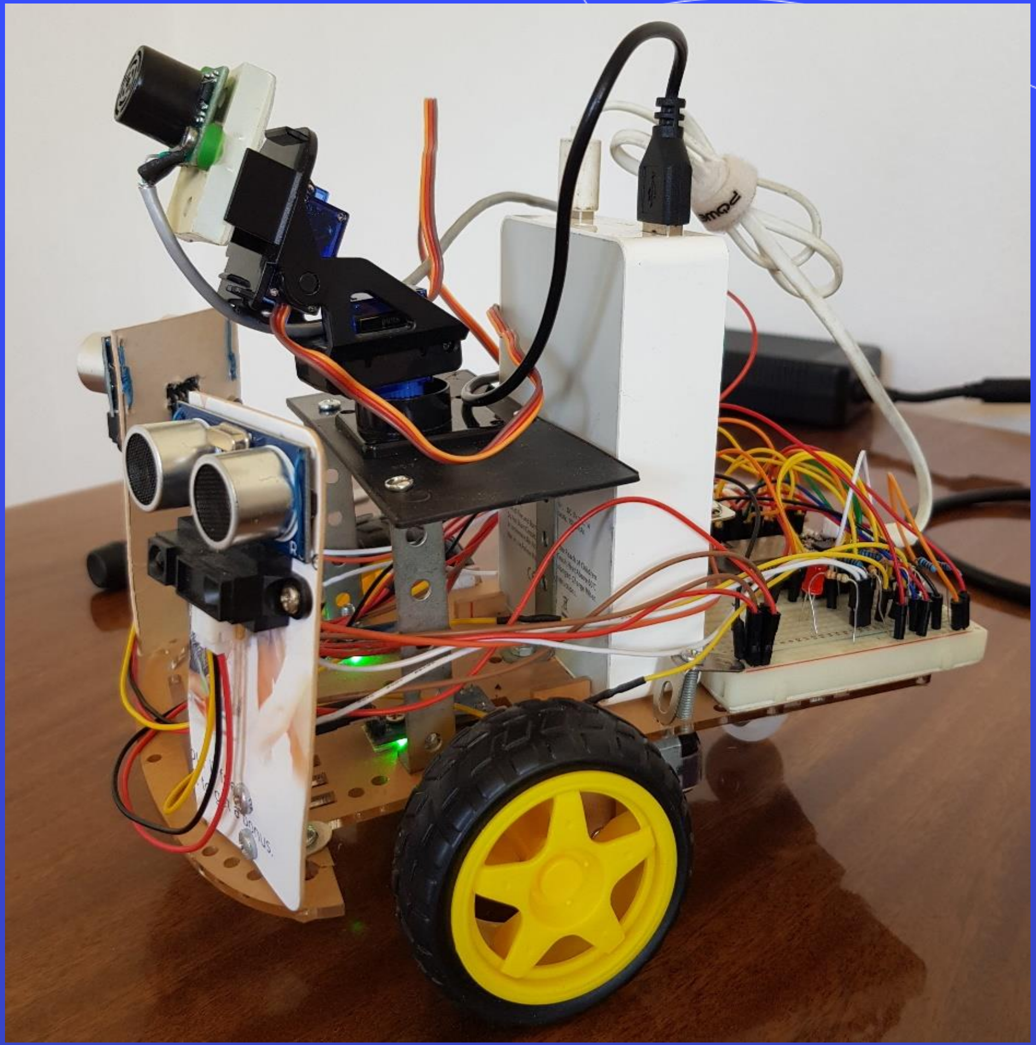
- Why is data co-partitioning required? Because KStream-KStream, KTable-KTable, and KStream-KTable joins are performed based on the keys of records (e.g., leftRecord.key == rightRecord.key), it is required that the input streams/tables of a join are co-partitioned by key.

- There are two exceptions where co-partitioning is not required. For KStream-GlobalKTable joins, co-partitioning is not required because all partitions of the GlobalKTable's underlying changelog stream are made available to each KafkaStreams instance. That is, each instance has a full copy of the changelog stream. Further, a KeyValueMapper allows for non-key based joins from the KStream to the GlobalKTable. KTable-KTable Foreign-Key joins also do not require co-partitioning. Kafka Streams internally ensures co-partitioning for Foreign-Key joins.

Demos

Available @ Github:

https://github.com/iproduct/kafka-streams-devbg



Thank you!

Contacts:

https://www.linkedin.com/in/trayaniliev/

https://www.facebook.com/IPT.EACAD

https://github.com/iproduct

https://twitter.com/trayaniliev

СЛЕДВАЩО СЪБИТИЕ

https://dev.bg/groups/java/

Лектор

Дата

Език