



Design Patterns for working with Fast Data

Portland Java User Group

October 18, 2016



Abstract

- Apache Kafka is an open-source message broker project that provides a platform for storing and processing real-time data feeds. In this presentation Ian Downard will describe the concepts that are important to understand in order to effectively use the Kafka API. You will see how to prepare a development environment from scratch, how to write a basic publish/subscribe application, and how to run it on a variety of different cluster types, starting with a single-node cluster on Virtual Box, then on multi-node cluster using Heroku's "Kafka as a Service", and finally on an enterprise-grade multi-node cluster using MapR's Converged Data Platform.
- Ian will also discuss strategies for working with "fast data" and how to maximize the throughput of your Kafka pipeline. He'll describe which Kafka configurations and data types have the largest impact on performance and provide some useful JUnit tests, combined with statistical analysis in R, that can help quantify how various configurations effect throughput.
- The code and presentation for this talk will be available at
<https://github.com/iandow/design-patterns-for-fast-data>



Author

- Ian Downard is a technical evangelist for MapR where he is focused on creating developer-friendly ways to use the [MapR Converged Data Platform.](#)
- Personal Blog: <http://www.bigendiandata.com>
- Twitter: <https://twitter.com/iandownard>
- GitHub: <https://github.com/iandow>
- LinkedIn: <https://www.linkedin.com/in/iandownard>



What is Kafka?

- A cluster based platform for storing and processing data.
- A distributed publish-subscribe messaging system.
- Persists on disk (not “in-memory”).
- Supports true streaming (not just fast batching).
- Can be used for data storage AND data processing (not just storage)
- Oriented around small messages and dynamic data sets (i.e. “streams”).



Why Kafka?

- Decoupling is one of the most important things that make services scalable.
- Kafka is elastic, resilient to failure, and low latency.
- Kafka fits well in microservice architectures.
- Kafka is becoming the default for inter-service communication.
- It's open source and doesn't require big hardware.



What is Streaming Data?

- Retail: orders, sales, shipments, price adjustments
- Finance: stock prices, other time series data
- Web sites: clicks, impressions, searches
- Operating Systems: machine metrics, and logs



Some examples of streaming datasets

- Social Media
 - <https://dev.twitter.com/streaming/overview>
- IoT
 - Lots of examples at <https://data.sparkfun.com/streams/>
 - <http://dweet.io> is pretty cool
- Servers:
 - /var/log/syslog
 - tcpdump
- Banking
 - Yahoo Finance API: <https://github.com/cjmatta/TwitterStream>,
<http://meumobi.github.io/stocks%20apis/2016/03/13/get-realtime-stock-quotes-yahoo-finance-api.html>
 - NYSE markets: <http://www.nyxdata.com/Data-Products/Daily-TAQ>
- Other
 - screen scrape anything



Overview

- Connecting to a streaming endpoint
- Streaming API request parameters
- Streaming message types
- Processing streaming data

Public API

User Streams

Site Streams

Firehose

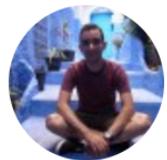
The Streaming APIs Overview

The Streaming APIs give developers low latency access to Twitter's global stream of Tweet data. A proper implementation of a streaming client will be pushed messages indicating Tweets and other events have occurred, without any of the overhead associated with polling a REST endpoint.

If your intention is to conduct singular searches, read user profile information, or post Tweets, consider using the [REST APIs](#) instead.

Twitter offers several streaming endpoints, each customized to certain use cases.

Public streams	Streams of the public data flowing through Twitter. Suitable for following specific users or topics, and data mining.
User streams	Single-user streams, containing roughly all of the data corresponding with a single user's view of Twitter.
Site streams	The multi-user version of user streams. Site streams are intended for servers which must connect to Twitter on behalf of many users.

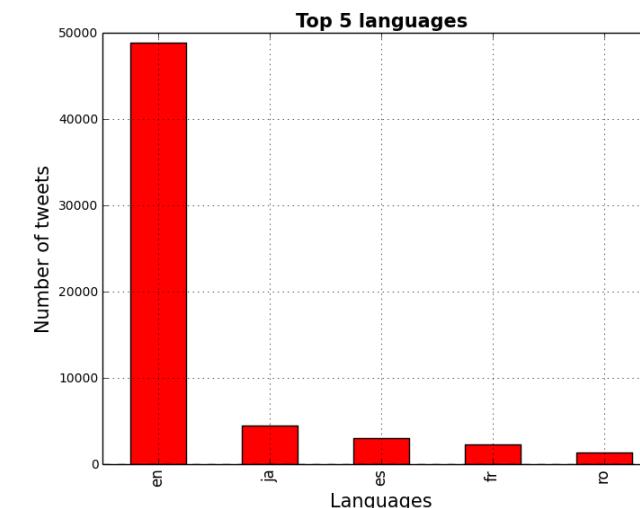
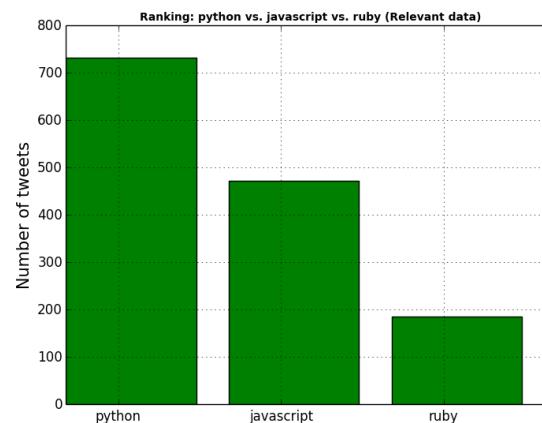
<http://adilmoujahid.com/posts/2014/07/twitter-analytics/>

Adil Moujahid

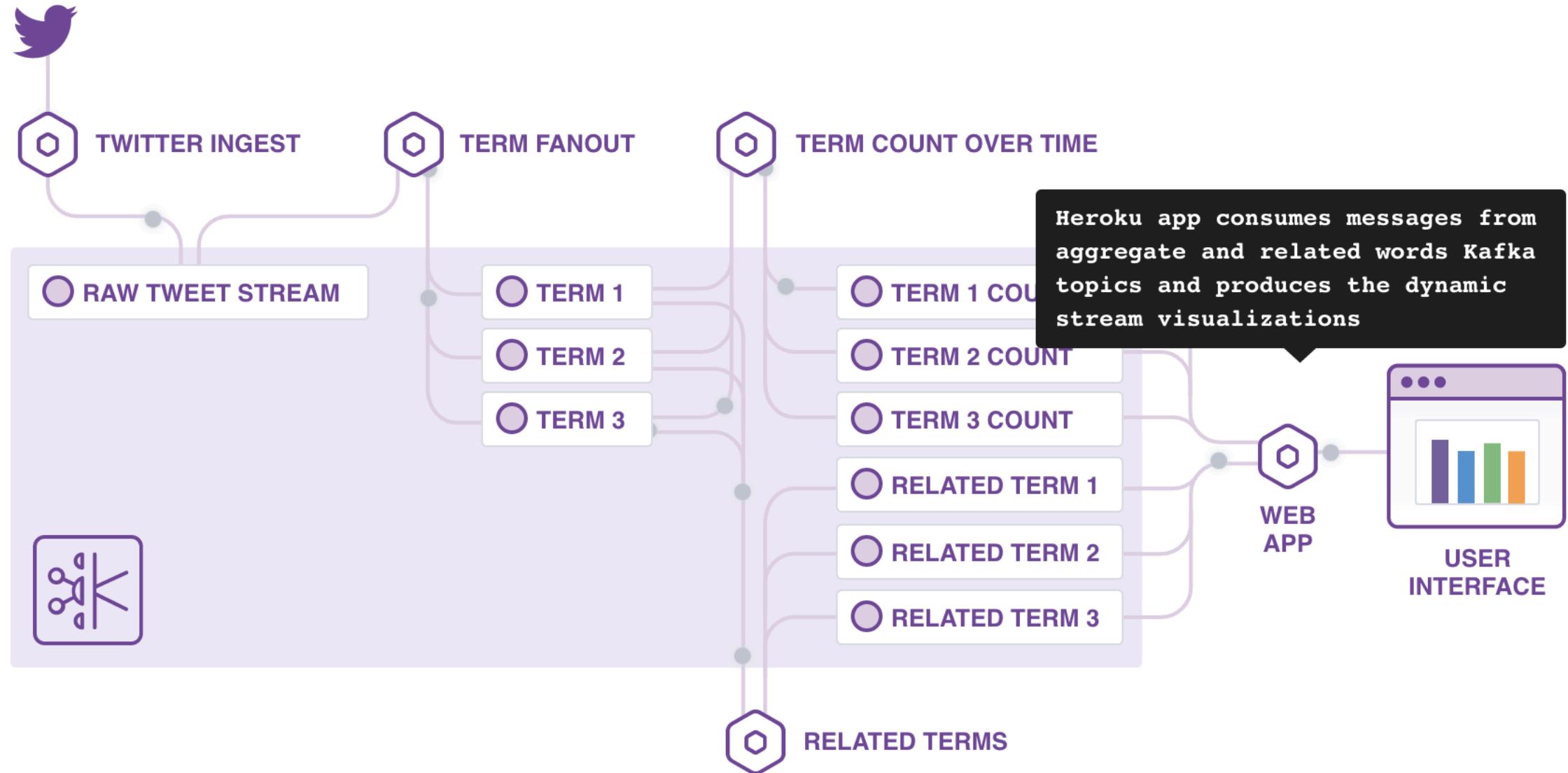
An Introduction to Text Mining using Twitter Streaming API and Python

// tags [python](#) [pandas](#) [text mining](#) [matplotlib](#) [twitter](#) [api](#)

“Text mining is the application of natural language processing techniques and analytical methods to text data in order to derive relevant information.”



<https://heroku.github.io/kafka-demo/>

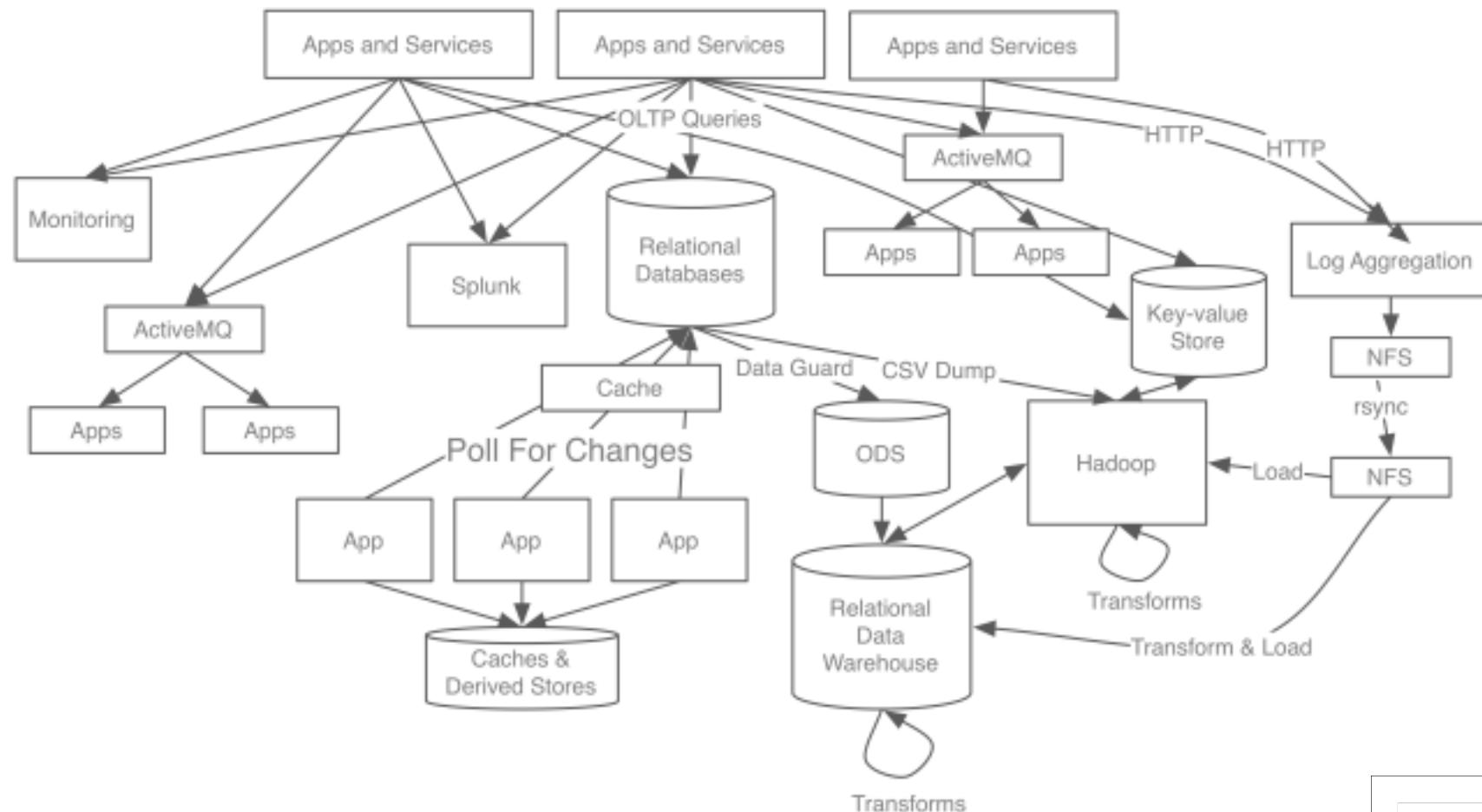


Share your thing – like it ain't no thang.

Try It Now

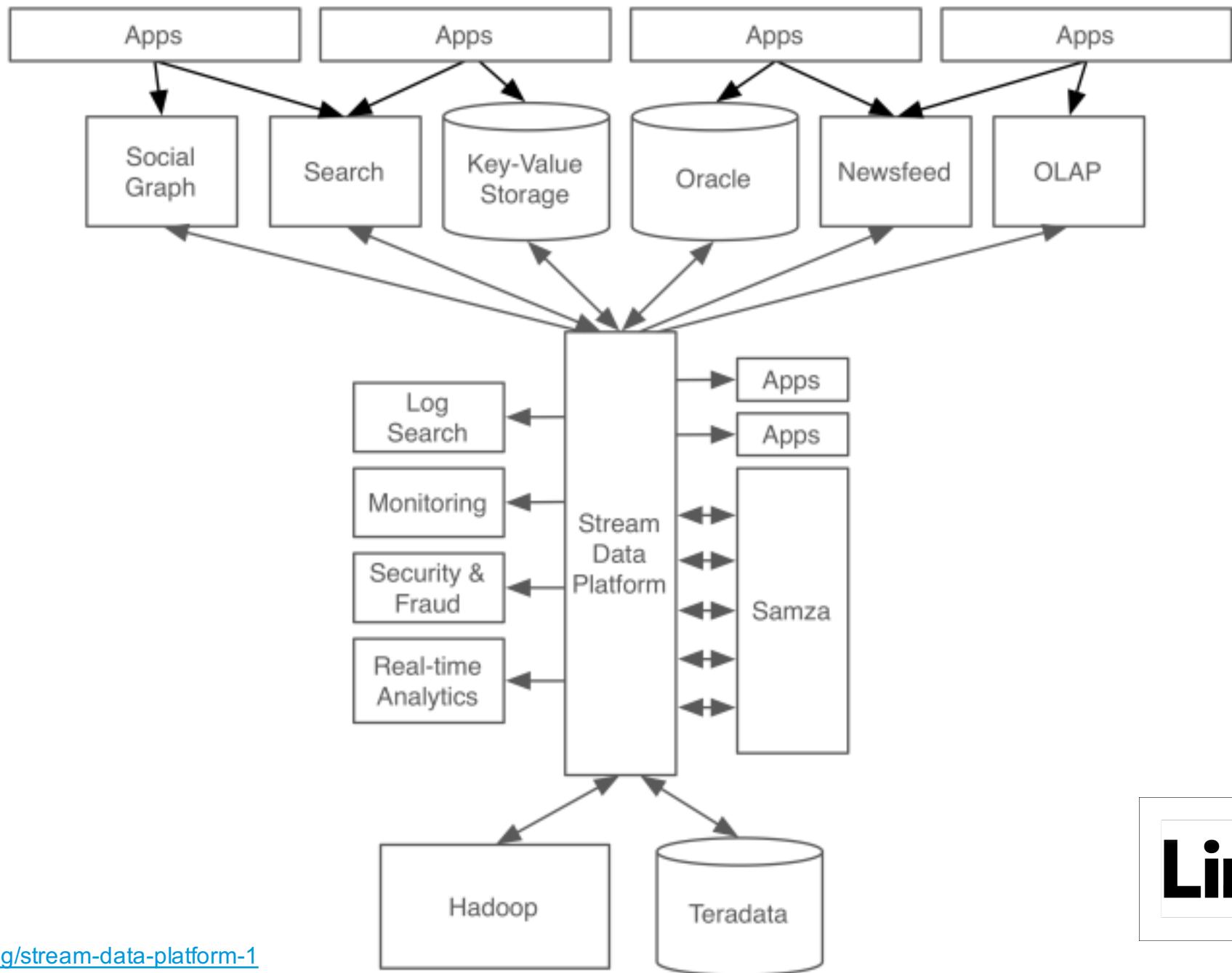
(Hint: For BIG fun, try it on your smartphone or tablet)





Citation:

<http://www.confluent.io/blog/stream-data-platform-1>

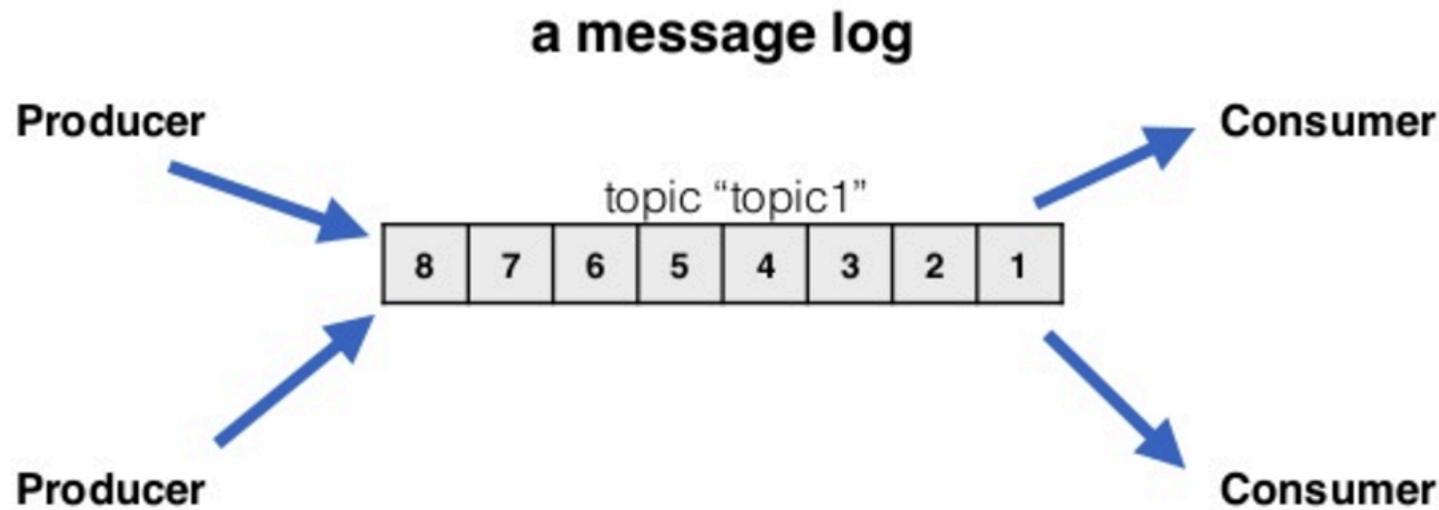


Citation:

<http://www.confluent.io/blog/stream-data-platform-1>

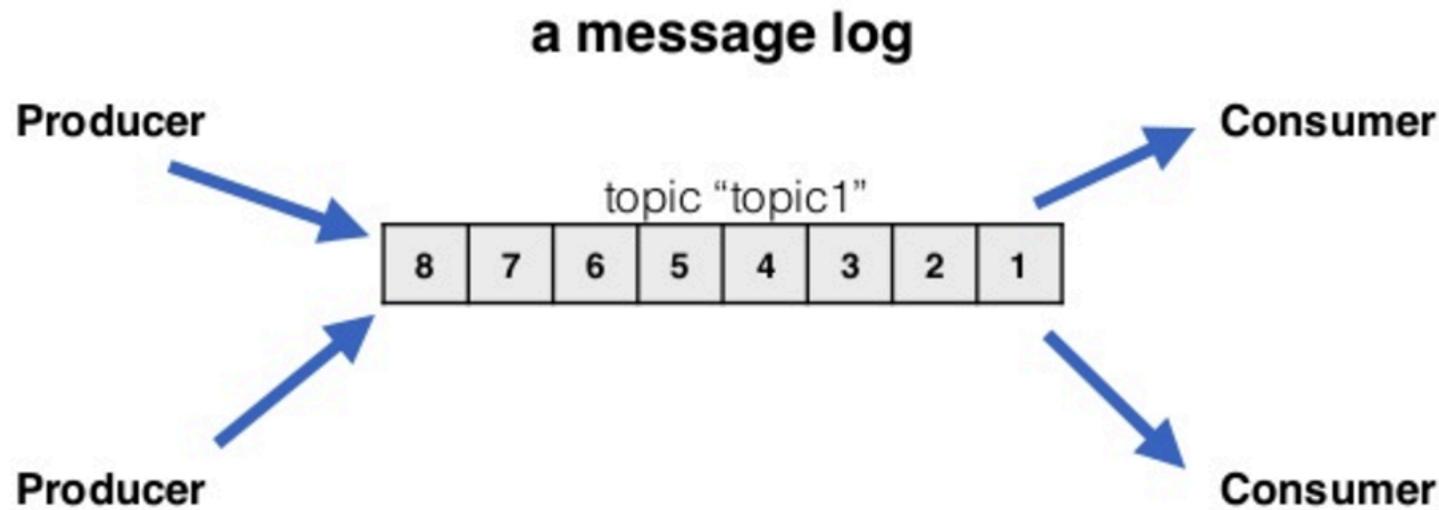
LinkedIn

Apache Kafka



- Messages contain <Key, Value>
- delivered in order
- transferred as bytes (you specify serializer / deserializer)
- remain available for TTL
- with **at least once semantics**

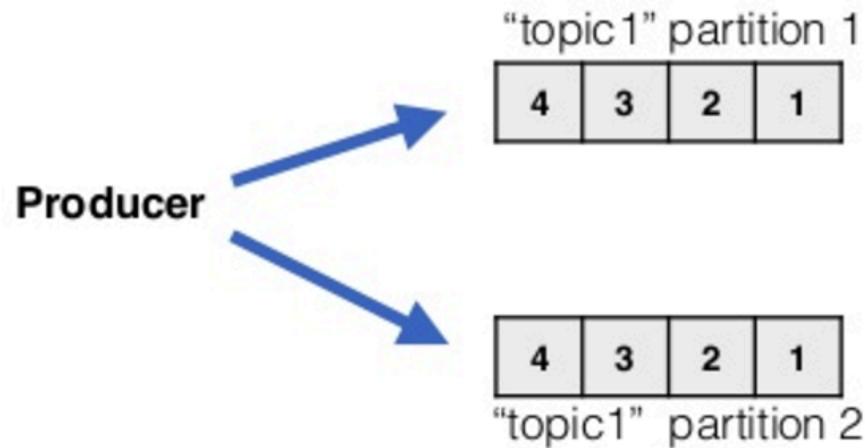
Apache Kafka



- Producers send messages one at a time
- Consumers subscribe to topics, poll for messages, and receive records many at a time

Apache Kafka

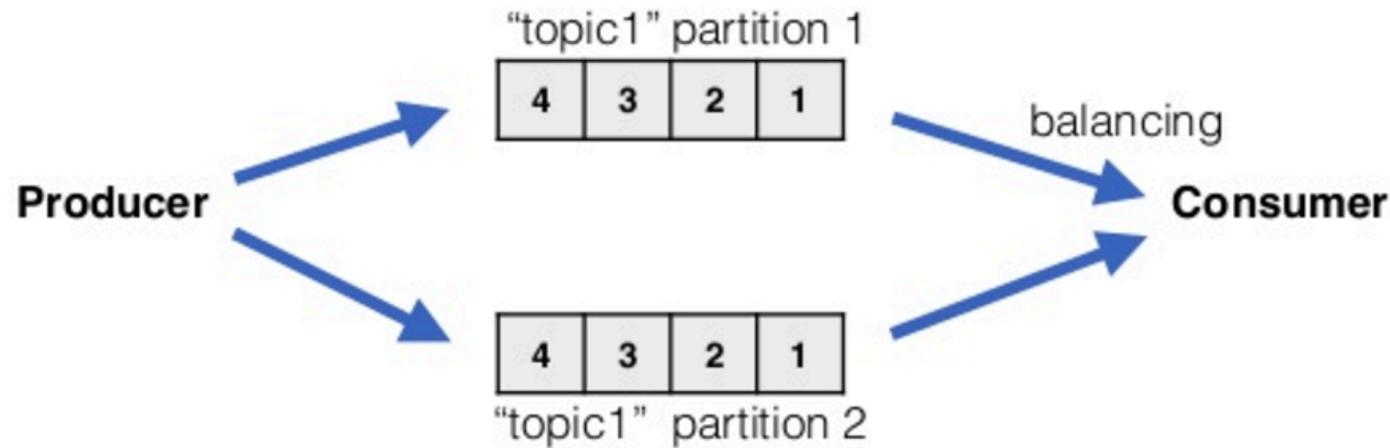
a distributed message log



- Producer sees a topic as one logic object.
- The order of messages within a single partition is guaranteed

Apache Kafka

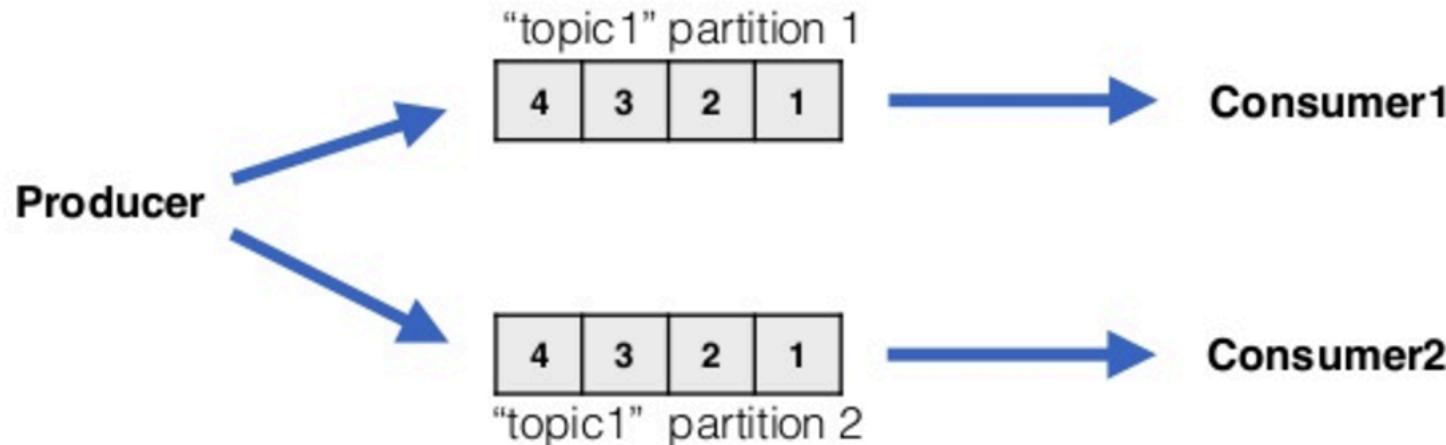
a distributed message log



- Partitioning facilitates concurrent consumers because a single partition can only be read by one consumer.
- Broker will deal with balancing with multiple consumers.

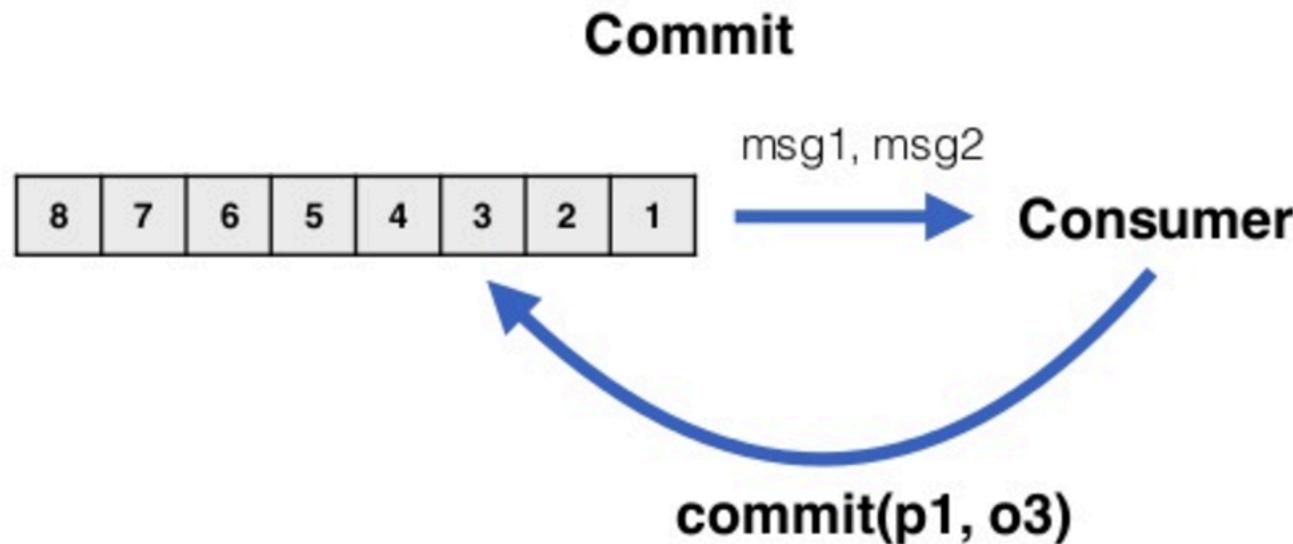
Apache Kafka

a distributed message log



- But if another consumer joins, a rebalancing will occur and some partitions will be assigned to one consumer and some to another.
- The server will also automatically assign partitions to a consumer, and rebalance as new consumers subscribe

Apache Kafka



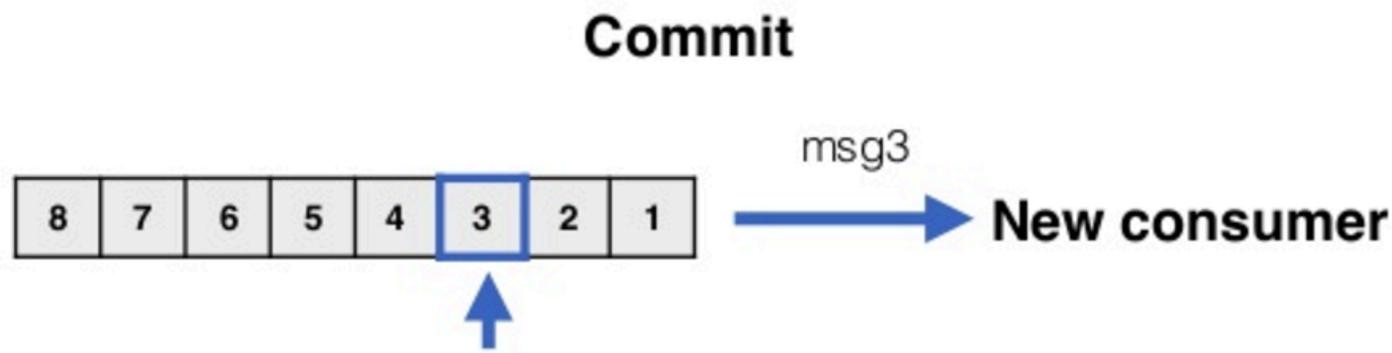
- Calling `commit` marks where a consumer has finished processing.

Apache Kafka



- Thanks to commit, when a consumer goes down we let it crash.

Apache Kafka



- When another consumer takes its place, it continues where the old consumer left off.

Apache Kafka

Uses Zookeeper

Leader election

Cluster coordination

Shared configuration

(I copied his Kafka intro slides.)

 **Krzysztof Ciesielski** @kpciesielski · Oct 4

Thanks for attending my talk "Reactive Kafka With Akka Streams" at #reactivesummit :) I just published the slides:



The thumbnail shows a blue play button in the center, with the title "Reactive Kafka with Akka Streams" above it. Below the play button, it says "Krzysztof Ciesielski" and "Reactive Summit 2016, Austin, TX".

Reactive Kafka with Akka Streams

A walk through the features of akka-stream-kafka, a library for representing Apache Kafka as stages of Akka Streams.

[slideshare.net](#)

<https://twitter.com/kpciesielski/status/783417291568402432>

<https://www.youtube.com/watch?v=mgVH3NMGMUg>

DEMO: Kafka CLI

- Create, alter, pub/sub, and delete topic.
- See where topics are saved.



DEMO: Java API

- Basic producer/consumer example



DEMO: Heroku Kafka-as-a-Service

- Demo app:
 - <https://github.com/iandow/heroku-kafka-demo-java>

- Docs and Pricing:
 - <https://devcenter.heroku.com/articles/kafka-on-heroku>
 - \$1,500/month !!!
 - Oh, but rounding credits.

Account details				Expand All
▶ bookfast			\$0.00	
▶ murmuring-cliffs-10090	deleted		\$0.06	
▼ secret-peak-66000	deleted		\$0.70	
Dyno-hours				
▶ web	Free	0.0002154 months	\$0.00/month	\$0.00
dyno credit				(\$0.00)
				Subtotal: \$0.00
Add-ons				
heroku-kafka:standard-0		0.0004682 months	\$1,500.00/month	\$0.70
				Subtotal: \$0.70

Credits

Rounding credit

(-\$0.76)



DEMO: Kafka in a web app

- How to send a message
 - How to ensure delivery guarantee
 - How to handle failure
 - Demonstrate it:
 - Change 20 second timeout in DemoResources.java
 - Observe the java.util.concurrent.Timeout in the user specified callback
 - Also observe that the message may ultimately get there, (if its in transit)
 - To get rid of Dropwizard metrics, change timeout in DemoApplication.java
 - reporter.start(1, TimeUnit.MINUTES);



Consumer Groups, Cursors, and Partitions



ProducerRecord vs ConsumerRecord

- Producers send messages one at a time as ProducerRecord
- Consumers subscribe to topics and poll for messages and receive them many-at-a-time as ConsumerRecords



Consumer Groups, Cursors, and Partitions

- Consumer Groups – groups of consumers working together
- Cursors – track positions of consumers
- Partitions – subdivide a topic for scale
- Replication Factor – replicate partitions across nodes for fault tolerance

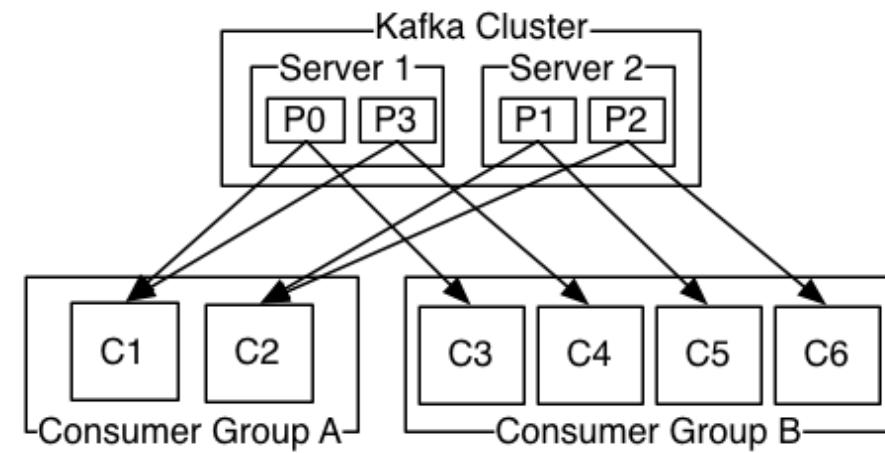


Partition Balancing

- Available partitions are balanced among consumers for a given group
 - Single partition can be read by only one consumer
 - A single consumer can read multiple partitions
 - Examples:
 - 3 partitions, 2 consumers: 2 & 1
 - 5 partitions, 1 consumer: 5
 - 3 partitions, 4 consumers: 1, 1, 1, 0 (4th consumer gets no messages)
- When new consumers join or new partitions added, rebalance occurs
 - Some consumers will gain or lose partitions and there may be a slight pause in message delivery while this happens
 - Can cause duplicates if consumers don't properly commit cursors

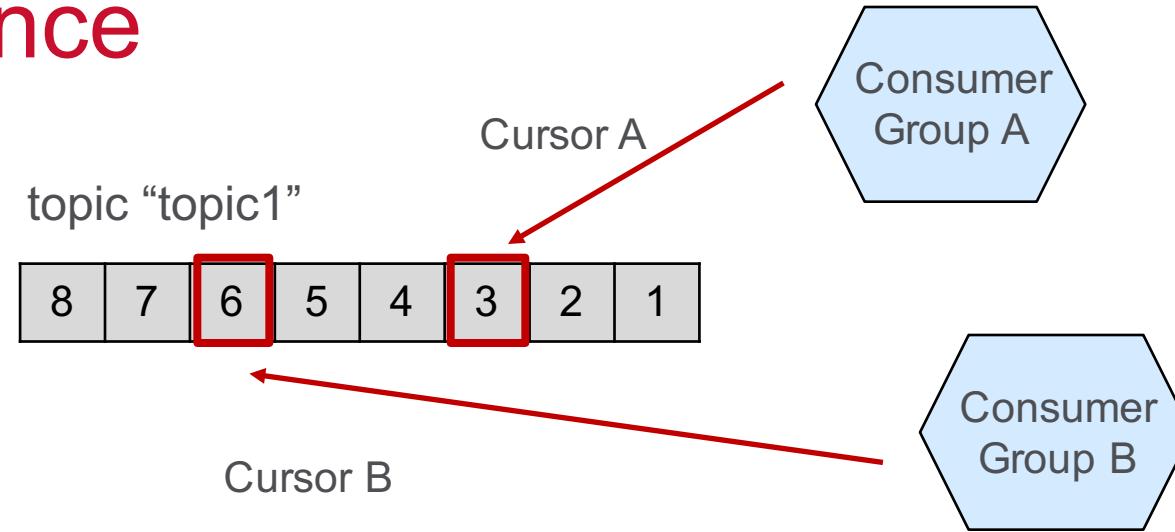
Consumer Groups

- Only one consumer IN A GROUP can consume one topic/partition simultaneously
- Consumers in different groups will see the same messages if they read the same topic/partition



<http://kafka.apache.org/documentation.html>

Cursor Persistence



- Ensure consumers pick up from where they left off after a stop
- Tracks last known commit point for message reads
- By default, auto commit after 5s (`auto.commit.interval.ms`)
 - If slow consumer code, `auto.commit()` might go out before processed
 - You can explicitly commit with `consumer.commit()`
- **If you need to concurrent consumers, would you use consumer groups or topic partitions?**

At Least Once semantics

- Messages may be seen multiple times
 - Producer side
 - Retransmit on network
 - Producer might fail and resend previously sent messages on restart
 - Consumer side
 - Client might crash after get but before cursor.commit()
- Applications must be **idempotent** message processors



Batching Behavior

- Producer.send() places messages in client-side memory buffer
- That buffer is not flushed until one of three criteria met:
 - Timeout (linger.ms)
 - Max memory (buffer.memory)
 - Producer.flush() called



Serialization

- Each Kafka message is a byte array.
- You must convert $\langle K, V \rangle$ to/from bytes with a serializer
- Two provided out of the box
 - ByteArraySerializer/ByteArrayDeserializer
 - StringSerializer/StringDeserializer
- You can write your own serializer.



Key Concept - Serialization

- If you're streaming data that is a byte array, use the byte array serializer.
- If you're streaming data that is a String, use the String serializer.
- If you're streaming data that is a POJO, use ???



How to stream a POJO

1. Make it serializable

- Otherwise you'll get a `java.io.NotSerializableException` later when we write the object to a byte array

2. Write the object to a byte array

3. Send that byte array

- Why can't we just send the Person object?

4. In the consumer, poll the topic, then convert `record.value()` to your POJO.

```
//Prepare bytes to send:  
ByteArrayOutputStream bos = new ByteArrayOutputStream();  
ObjectOutput out = null;  
out = new ObjectOutputStream(bos);  
out.writeObject(person1);  
out.flush();  
byte[] value = bos.toByteArray();
```

```
//Create object from bytes:  
ByteArrayInputStream bis = new ByteArrayInputStream(record.value());  
ObjectInput in = null;  
try {  
    in = new ObjectInputStream(bis);  
    Object obj = in.readObject();  
    System.out.println("Consumed object " + obj);  
    Person p2 = (Person)obj;  
    System.out.println("Consumed person " + p2.getName());
```



Problems with streaming POJOs

- What if your raw data is dirty, and casting fails because you streamed a record with missing POJO fields?
- The overhead of encoding/decoding a POJO to/from bytes is redundant, especially if this is done at multiple stages in the Kafka pipeline.
- It's better to postpone parsing/casting downstream.



How would you stream this?

080449201DAA T 0000019570000103000N0000000000000004CT100100710071007
080449201DAA T 0000013100000107000N0000000000000005CT10031007
080449201DAA T 000006660000089600N0000000000000006CT10041005
080449201DAA T 0000018020000105100N0000000000000007CT100310051009
080449201DAA T 00000132200000089700N0000000000000008CT100410051005
080449201DAA T 000009350000089400N0000000000000009CT10031007
080449201DAA T 000007510000105400N0000000000000010CT100410081006
080449201DAA T 000003130000088700N0000000000000011CT1004100810081007
080449201DAA T 000002180000105100N0000000000000012CT100410091005
080449201DAA T 0000019130000104500N0000000000000013CT10041006
080449201DAA T 0000012450000105100N0000000000000014CT1001100610071008



Schema excerpt

FIELD	OFFSET	SIZE	FORMAT	DESCRIPTION
Time	0	9	Date	Time of quote with milliseconds (HHMMSSXXX) This is the time the quote was published by the SIP (CTA or UTP)
Exchange	9	1	Text	The Exchange that issued the quote: <ul style="list-style-type: none">■ A – NYSE MKT Stock Exchange■ B – NASDAQ OMX BX Stock Exchange■ Z – BATS Exchange
Symbol	10	16	Text	Security symbol (six-character root, ten-character suffix)
Bid Price	26	11	Number	Bid price, seven characters for the whole number, four characters for decimals
Bid Size	37	7	Number	Bid size in units of trade, seven characters
Ask Price	44	11	Number	Ask price, seven characters for the whole number, four characters for decimals
Ask Size	55	7	Number	Ask size in units of trade, seven characters

Option A?

- Parse each record into a JSON object,
- Publish json_object.toString()
- Stream with String serializer

```
JSONObject trade_info = new JSONObject();
trade_info.put("date", record.substring(0, 9));
trade_info.put("exchange", record.substring(9, 10));
trade_info.put("symbol root", record.substring(10, 16).trim());
trade_info.put("symbol suffix", record.substring(16, 26).trim());
trade_info.put("saleCondition", record.substring(26, 30).trim());
trade_info.put("tradeVolume", record.substring(30, 39));

producer.send(
    new ProducerRecord<String, String>(key, trade_info.toString())
);
```



Option B?

- Create a POJO (“Tick”) with attributes for each field
- Parse each record and create a Tick object
- Stream Tick objects using a **custom serializer**

```
public class Tick implements Serializable {  
    private String date;  
    private String exchange;  
    private String symbol_root;  
    private String symbol_suffix;  
    private String saleCondition;  
    private String tradeVolume;  
  
    public String getDate() {  
        return date;  
    }  
  
    public void setDate(String date) {  
        this.date = date;  
    }  
  
    public String getExchange() {  
        return exchange;  
    }  
  
    public void setExchange(String exchange) {  
        this.exchange = exchange;  
    }  
}
```



Option C?

- Create a POJO “Tick” with a single byte[] attribute, and getters that return fields by directly indexing the byte[]
- Annotate getters with @JsonProperty

```
public class Tick implements Serializable {  
    private byte[] data;  
  
    public Tick(byte[] data) { this.data = data; }  
  
    public Tick(String data) { this.data = data.getBytes(Charsets.ISO_8859_1); }  
  
    @JsonProperty("date")  
    public String getDate() { return new String(data, 0, 9); }  
  
    @JsonProperty("exchange")  
    public String getExchange() { return new String(data, 9, 1); }  
  
    @JsonProperty("symbol-root")  
    public String getSymbolRoot() { return trim(10, 6); }  
  
    @JsonProperty("symbol-suffix")  
    public String getSymbolSuffix() { return trim(16, 10); }  
  
    @JsonProperty("sale-condition")  
    public String getSaleCondition() { return trim(26, 4); }  
  
    @JsonProperty("trade-volume")  
    public double getTradeVolume() { return digitsAsInt(30, 9); }  
}
```



```
JSONObject trade_info = new JSONObject();
trade_info.put("date", record.substring(0, 9));
trade_info.put("exchange", record.substring(9, 10));
trade_info.put("symbol root", record.substring(10, 16).trim());
trade_info.put("symbol suffix", record.substring(16, 26).trim());
trade_info.put("saleCondition", record.substring(26, 30).trim());
trade_info.put("tradeVolume", record.substring(30, 39));

producer.send(
    new ProducerRecord<String, String>(key, trade_info.toString())
);
```

A

```
public class Tick implements Serializable {
    private String date;
    private String exchange;
    private String symbol_root;
    private String symbol_suffix;
    private String saleCondition;
    private String tradeVolume;

    public String getDate() {
        return date;
    }

    public void setDate(String date) {
        this.date = date;
    }

    public String getExchange() {
        return exchange;
    }

    public void setExchange(String exchange) {
        this.exchange = exchange;
    }
}
```

B

```
public class Tick implements Serializable {
    private byte[] data;

    public Tick(byte[] data) { this.data = data; }

    public Tick(String data) { this.data = data.getBytes(Charsets.ISO_8859_1); }

    @JsonProperty("date")
    public String getDate() { return new String(data, 0, 9); }

    @JsonProperty("exchange")
    public String getExchange() { return new String(data, 9, 1); }

    @JsonProperty("symbol-root")
    public String getSymbolRoot() { return trim(10, 6); }

    @JsonProperty("symbol-suffix")
    public String getSymbolSuffix() { return trim(16, 10); }

    @JsonProperty("sale-condition")
    public String getSaleCondition() { return trim(26, 4); }

    @JsonProperty("trade-volume")
    public double getTradeVolume() { return digitsAsInt(30, 9); }
}
```

C

Major Performance Factors

- Replication turned on?
- Streaming large messages?
- Producers sending synchronously or async.?
- Send buffer flushed until one of three criteria met:
 - Timeout (linger.ms)
 - Max memory (buffer.memory)
 - Producer.flush() called
- **Topics / Producer affinity**



Major Performance Factors

- What affects producer latency?
 - Using send futures? (i.e. synchronous sends)
 - acks=all,1,or 0 (ack after leader logs / after all replicas log / or not at all)
 - "Nagle's algorithm" and the linger.ms config
- What affects producer throughput?
 - Producer batch.size
 - Concurrent producer.send()
 - Number of topics
 - Message size
- What affects consumer throughput?
 - Partitions



Major Performance Factors

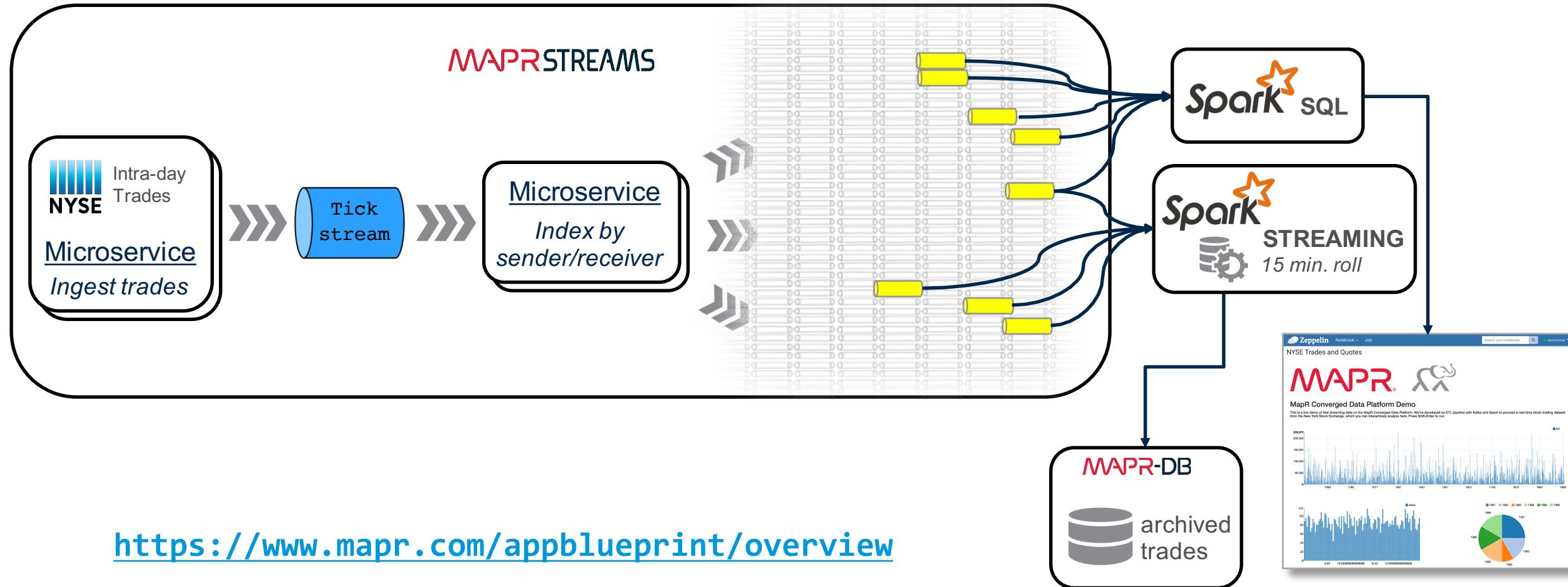
- What affects Kafka server stability?
 - Garbage Collection: Long GCs from sending large messages can break the kafka/zookeeper connection.
 - Topics can easily consume disk space
 - May need to increase file handles
 - Topic data and index files are in log.dir (e.g. /tmp/kafka-logs)
 - Kafka keeps an open file for each topic.
 - Reference:
<http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/>



Read vs Processing Parallelism

- Consumer read parallelism
 - Concurrent reads can be achieved via topic partitioning and consumer groups
 - If your consumer logic is CPU-bound, use partitions or consumer groups
- Downstream Processing Parallelism
 - If you need more processing threads than read threads, then “**fan-out**” the data to more topics.

What I mean by, "Fan-out":



Performance Anti-Pattern

- Problem: I want to count all the messages read in each consumer thread.
- Anti-pattern: I'll just increment a counter in a synchronized block.



Performance Anti-Pattern

- I want to count all the messages read in each consumer thread.
- Fine, I'll just increment a counter in a synchronized block.
 - Lesson learned: **synchronization really hurts throughput.**
 - Solution: I decreased the cost of metric collection to near zero by giving each thread its own metrics structure, and every second or so, updating a shared metric structure with locks.



Performance Anti-Pattern

- I want to be sure never to read a duplicate message.
- Fine, I'll just commit the read point for every message I consume.



Performance Anti-Pattern

- I want to be sure never to read a duplicate message.
- Fine, I'll just commit the read point for every message I consume.
 - Lesson learned, you're still not guaranteed not to get a duplicate message. Kafka consumers must be idempotent.
 - Solution: Postpone deduplication downstream. (i.e. Give up till' later.)

Performance Anti-Pattern

- I want to stream JSON strings to simplify DB persistence and columnar selections in Spark.
- Fine, I'll just convert every message to a JSON object at the first step of my pipeline and use a JSON serializer.



Performance Anti-Pattern

- I want to stream JSON strings to simplify DB persistence and columnar selections in Spark.
- Fine, I'll just convert every message to a JSON object at the first step of my pipeline and use a JSON serializer.
 - Lesson learned: **parsing is expensive!** Handling parsing errors is expensive. Custom serializers are not as easy I as thought.
 - This anti-pattern is illustrated here: <https://github.com/mapr-demos/finserv-ticks-demo/commit/35a0ddaad411bf7cec34919a4817482df2bf6462>
 - Solution: Stream a data structure containing a byte[] to hold raw data, and annotate it with com.fasterxml.jackson.annotation.JsonProperty.
 - Example: <https://github.com/mapr-demos/finserv-ticks-demo/blob/35a0ddaad411bf7cec34919a4817482df2bf6462/src/main/java/com/mapr/demo/finserv/Tick2.java>



Good Practices

- Design your data structures to prefer arrays of objects, and primitive types, instead of the standard Java or Scala collection classes (e.g. `HashMap`).
- Maintain affinity between topic and producer thread.
 - Kafka will batch multiple sends together. If a send hits only one topic, it will benefit from writes to sequential memory locations.
 - Reference code:
<https://github.com/mapr-demos/finserv-application-blueprint/blob/master/src/test/java/com/mapr/demo/finserv/ThreadCountSpeedIT.java>
- Consider using numeric IDs or enumeration objects instead of strings for keys.

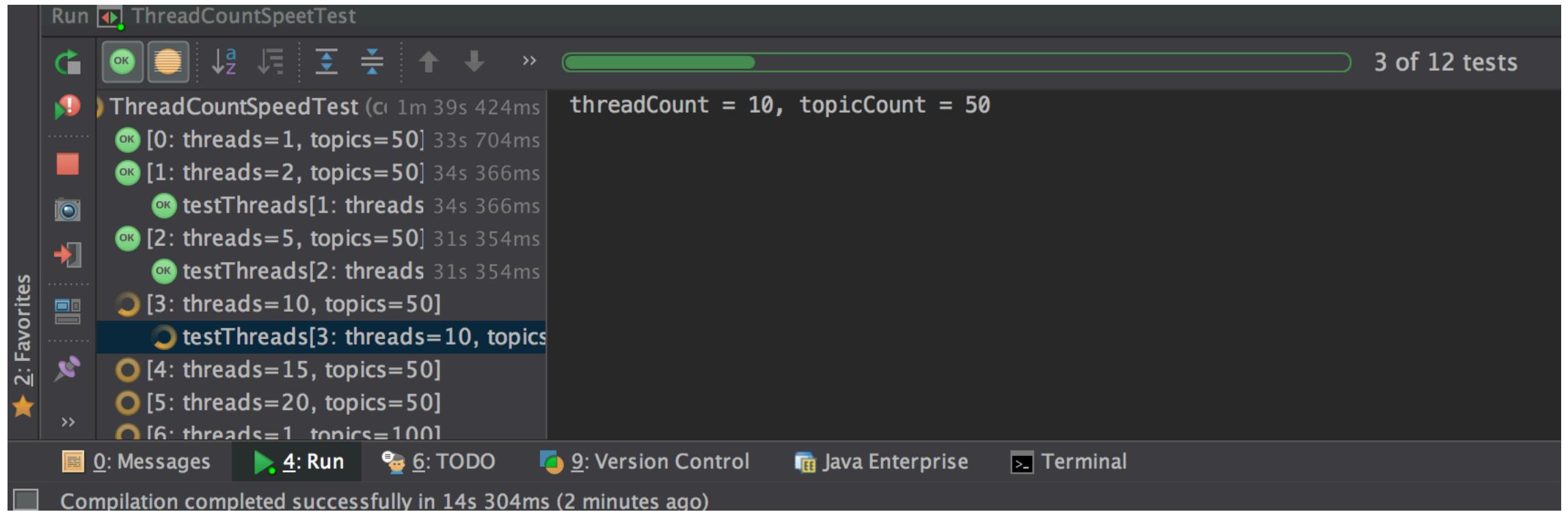


Three Performance Takeaways

1. Process as little as possible in the pipeline.
 - Transfer byte arrays. It's faster than serializers based on strings, POJOs, or custom serialization.
2. Use JSON annotation. It makes persistence easier in the final stages. It also makes data analysis easier because it gives you an easy schema.
3. Categorize raw data into multiple topics (aka “fan-out”). It makes real-time analysis easier.
4. Use parameterized tests in Junit to tune Kafka.



Junit tests



https://github.com/iandow/kafka_junit_tests

How to create Junit parameterized tests

1. Annotate test class with `@RunWith(Parameterized.class)`.
2. Create a public static method annotated with `@Parameters` that returns a Collection of Objects (as Array) as test data set.
3. Create a public constructor that takes in one "row" of test data.



Benchmarks

- Lazy benchmark, by Jay Kreps at LinkedIn (2014):
 - <https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines>
- Answers questions like these:
 - Will Kafka slow down if I never consumer messages?
 - Should I stream small messages or large messages (in order to maximize Throughput)?
 - Should I stream small messages or large messages?

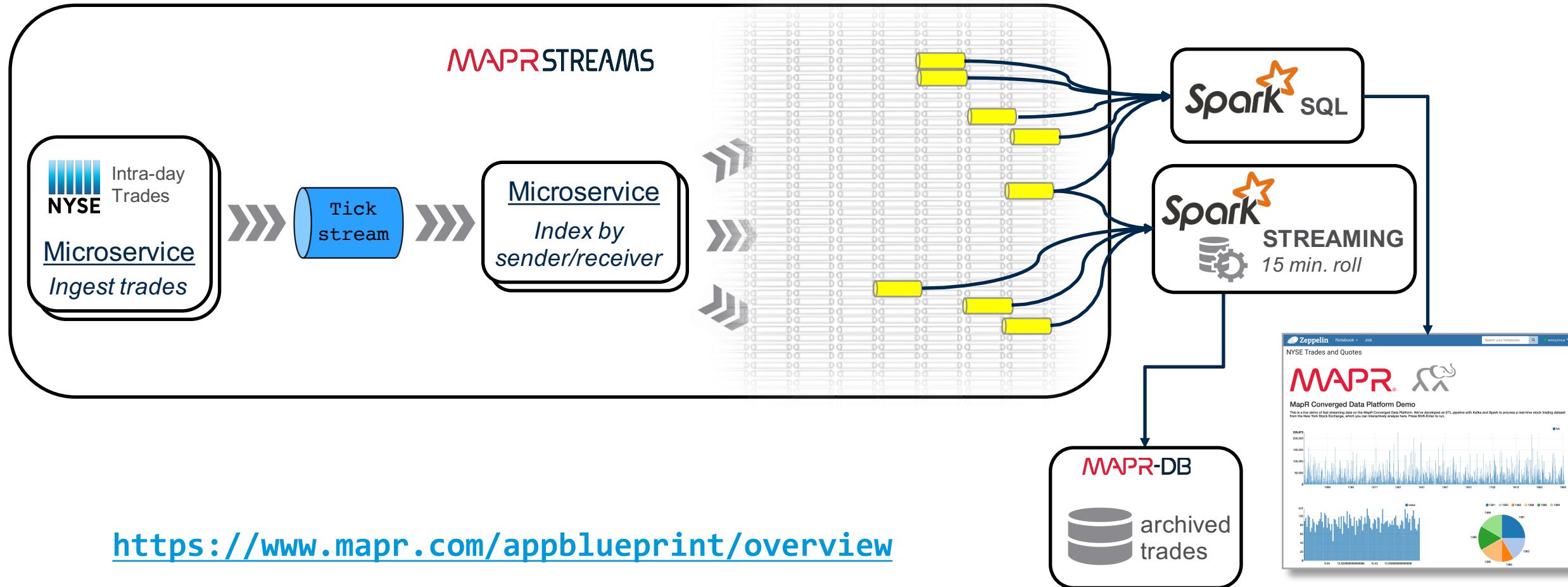
MapR Streams → (more scalable Kafka)

- MapR Streams load balances topics more effectively
 - Kafka assumes all partitions are equal in terms of size and throughput
- MapR Streams stores stream data in a distributed file system (MapR-FS), allowing topics to save much more data and replicate much faster.
 - Kafka partitions are pinned to a single node, meaning they can't outgrow the storage capacity of that node.
- MapR Streams can span across clusters in multi-datacenters
 - Kafka, “More Clusters, More Problems”

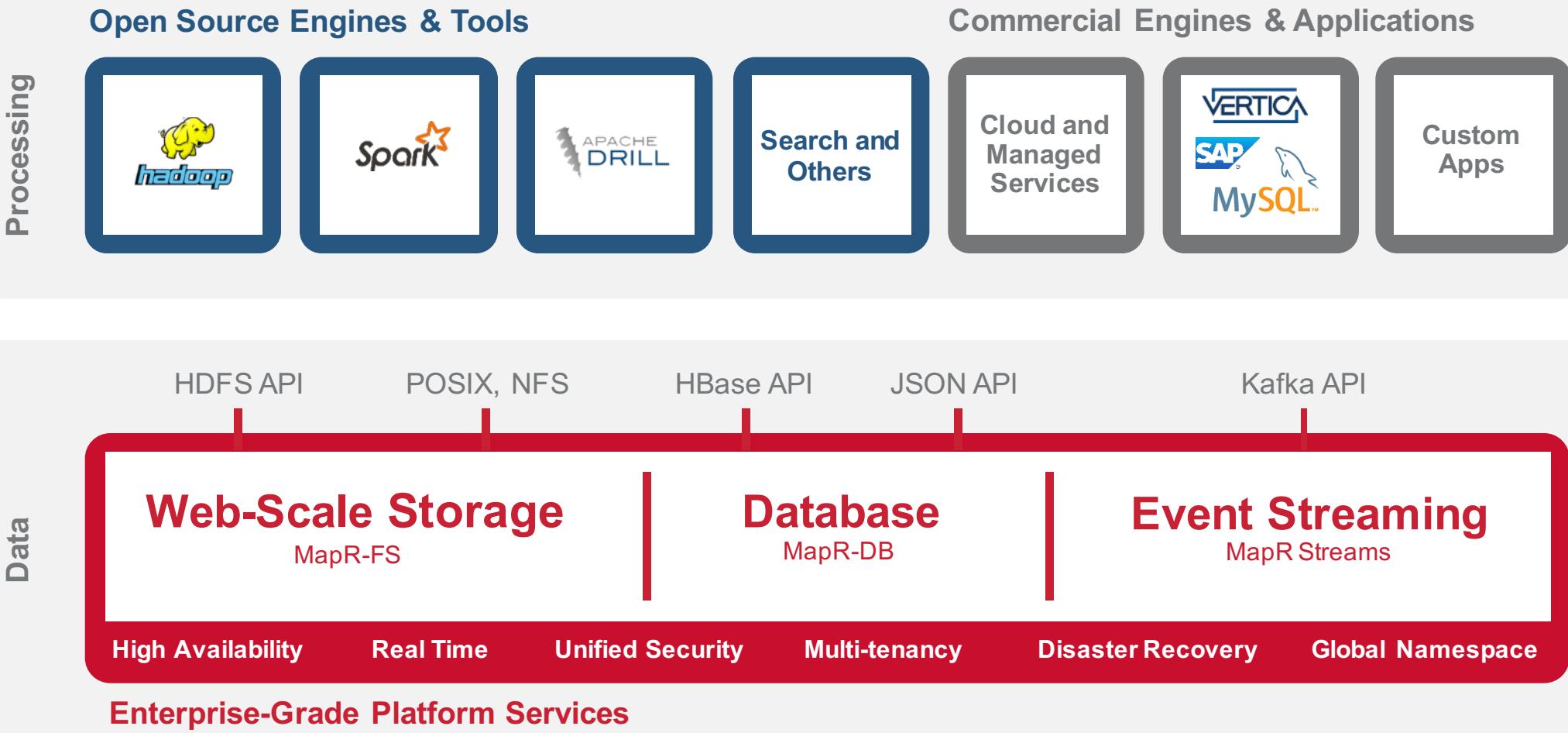
<https://www.mapr.com/blog/scaling-kafka-common-challenges-solved>



Blueprint of a Converged Application for real-time tick streaming



MapR Converged Data Platform



Resources

- MapR Streams
<https://www.mapr.com/blog/how-apache-kafka-and-mapr-streams-handle-topic-partitions>
- Heroku kafka
 - <https://github.com/heroku/heroku-kafka-demo-java>
 - heroku plugins:install heroku-kafka
 - heroku addons:create heroku-kafka:beta-dev
 - <https://heroku.github.io/kafka-demo/#architecture>
- <https://twitter.com/kpciesielski/status/783417291568402432>
 - (I copied his Kafka intro slides)
- Basic Kafka operations:
 - <https://kafka.apache.org/090/ops.html>



References

- Jay Kreps, [@jaykreps](#)
 - A Practical Guide to using Kafka:
 - <http://www.confluent.io/blog/stream-data-platform-1/>
 - The canonical post on Kafka design: "[The Log](#)"
 - <https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>
 - Kafka intro and why it's the defacto standard for streaming (Oct. 2016)
<https://youtu.be/rXzuZb3fHHM?list=PLKKQHTLcxDVZDAHuclOhAxMaaPtZ8ZDAi>

