



Streaming



Fundamentals






Stream Processing

A stream is a constant flow of data that is never interrupted and "never ends".

- » **Stream processing** is the process of that stream to get information from the dataflow.
- » **Real-time processing** is a specialized case of stream processing, where the result are obtained at such a fast pace that they "seem like they are calculated as they happend".

Currently there is a growing need to be able to process streaming data in real-time.





Stream Examples

» **Web Servers:**

- click stream
- visits
- log HTTP.

» **Twitter:** conversation timeline.

» **Youtube / Instagram / Facebook:** Comment feed.

» **Sensors:** temperature, humidity, weather stations, etc.

» **Cell Phones:** location information, gyroscope, app events, etc.




Stream Example: Twitter Firehose

```
{
  "created_at": "Tue May 23 17:58:20 +0000 2017",
  "id": "867077248448331776",
  "id_str": "867077248448331776",
  "text": "RT @manuginobili: TRIVIA:\n- Name the only 2 WC teams I've never played against in playoffs.\n- \u00danicos 2 equipos del Oeste contra los que no\u0020",
  "truncated": false,
  "entities": {
    "hashtags": [],
    "symbols": [],
    "user_mentions": {
      "screen_name": "manuginobili",
      "name": "Manu Ginobili",
      "id": "24342206",
      "id_str": "24342206",
      "indices": [3, 16]
    },
    "urls": []
  },
  "source": "\u003ca href=\"http://twitter.com/download/android\" rel=\"nofollow\" \u003eTwitter for Android\u003c/a\u003e",
  "in_reply_to_status_id": null,
  "in_reply_to_status_id_str": null,
  "in_reply_to_user_id": null,
  "in_reply_to_user_id_str": null,
  "in_reply_to_screen_name": null,
  "user": {
    "id": "471530433",
    "id_str": "471530433",
    "name": "",
    "screen_name": "GoodbyeGodbless",
    "location": "",
    "description": "",
    "url": null,
    "entities": {
      "description": {
        "urls": []
      }
    },
    "protected": false,
    "followers_count": 280,
    "friends_count": 323,
    "listed_count": 23,
    "created_at": "Mon Jan 23 00:09:53 +0000 2012",
    "favourites_count": 6647,
    "utc_offset": null,
    "time_zone": null,
    "geo_enabled": false,
    "verified": false,
    "statuses_count": 23451,
    "lang": "en",
    "contributors_enabled": false,
    "is_translator": false,
    "is_translation_enabled": false,
    "profile_background_color": "131516",
    "profile_background_image_url": "http://abs.twimg.com/themes/theme14/bg.gif",
    "profile_background_image_url_https": "https://abs.twimg.com/themes/theme14/bg.gif",
    "profile_background_tile": true,
    "profile_image_url": "http://pbs.twimg.com/profile_images/833040576828289024/cuS2koV0_normal.jpg",
    "profile_image_url_https": "https://pbs.twimg.com/profile_images/833040576828289024/cuS2koV0_normal.jpg",
    "profile_link_color": "009999",
    "profile_sidebar_border_color": "EEEEEE",
    "profile_sidebar_fill_color": "EFEFEF",
    "profile_text_color": "333333",
    "profile_use_background_image": true,
    "has_extended_profile": false,
    "default_profile": false,
    "default_profile_image": false,
    "following": false,
    "follow_request_sent": false,
    "notifications": false,
    "translator_type": "none"
  },
  "geo": null,
  "coordinates": null,
  "place": null,
  "contributors": null,
  "retweeted_status": {
    "created_at": "Fri Apr 14 14:10:02 +0000 2017",
    "id": "852886667568480256",
    "id_str": "852886667568480256",
    "text": "TRIVIA:\n- Name the only 2 WC teams I've never played against in playoffs.\n- \u00danicos 2 equipos del Oeste contra los que no jugu\u000e9 en playoffs.",
    "truncated": false,
    "entities": {
      "hashtags": [],
      "symbols": [],
      "user_mentions": [],
      "urls": []
    },
    "source": "\u003ca href=\"http://twitter.com/Client\u003c/a\u003e",
    "in_reply_to_status_id": null,
    "in_reply_to_status_id_str": null,
    "in_reply_to_user_id": null,
    "in_reply_to_user_id_str": null,
    "in_reply_to_screen_name": null,
    "user": {
      "id": "24342206",
      "id_str": "24342206",
      "name": "Manu Ginobili",
      "screen_name": "manuginobili",
      "location": "San Antonio V Bahia Blanca",
      "description": "El 20 de los Spurs y el 5 de Argentina.",
      "url": "http://t.co/vJvrnVhGQzV",
      "entities": {
        "url": {
          "urls": [
            {
              "url": "http://t.co/vJvrnVhGQzV",
              "expanded_url": "http://www.manuginobili.com",
              "display_url": "manuginobili.com",
              "indices": [0, 22]
            }
          ]
        },
        "description": {
          "urls": []
        }
      },
      "protected": false,
      "followers_count": 4739191,
      "friends_count": 92,
      "listed_count": 11036,
      "created_at": "Sat Mar 14 06:48:43 +0000 2009",
      "favourites_count": 119,
      "utc_offset": -18000,
      "time_zone": "Central Time (US & Canada)",
      "geo_enabled": false,
      "verified": true,
      "statuses_count": 5622,
      "lang": "en",
      "contributors_enabled": false,
      "is_translator": false,
      "is_translation_enabled": false,
      "profile_background_color": "9AE4E8",
      "profile_background_image_url": "http://pbs.twimg.com/profile_background_images/36155597/test.jpg",
      "profile_background_image_url_https": "https://pbs.twimg.com/profile_background_images/36155597/test.jpg",
      "profile_background_tile": false,
      "profile_image_url": "http://pbs.twimg.com/profile_images/919752875/3281_85574346933_71769686933_2527120_1371640_n_normal.jpg",
      "profile_image_url_https": "https://pbs.twimg.com/profile_images/919752875/3281_85574346933_71769686933_2527120_1371640_n_normal.jpg",
      "profile_link_color": "B9B9B6",
      "profile_sidebar_border_color": "0A0B0A",
      "profile_sidebar_fill_color": "080808",
      "profile_text_color": "030202",
      "profile_use_background_image": true,
      "has_extended_profile": false,
      "default_profile": false,
      "default_profile_image": false,
      "following": false,
      "follow_request_sent": false,
      "notifications": false,
      "translator_type": "regular"
    },
    "geo": null,
    "coordinates": null,
    "place": null,
    "contributors": null,
    "is_quote_status": false,
    "retweet_count": 89,
    "favorite_count": 757,
    "favorited": false,
    "retweeted": false,
    "lang": "und",
    "is_quote_status": false,
    "retweet_count": 89,
    "favorite_count": 0,
    "favorited": false,
    "retweeted": false,
    "lang": "und"
  }
}
```



Streams of events

- » An **event** is an **identifiable occurrence** that has meaning within a data system.
 - » Structuring data ingestion and **processing as a stream of data helps make data systems more scalable**, robust and maintainable.
 - » The key in this kind of systems is to **focus on storing actions that change** the state and not the state in each moment.
- 



Bounded Data vs Unbounded Data

Type of dataset (according to size)

- » Bounded data sets => batch processing.
- » Unbounded data sets => stream processing.

Batch processing can be considered as a special case of stream processing (finite stream).

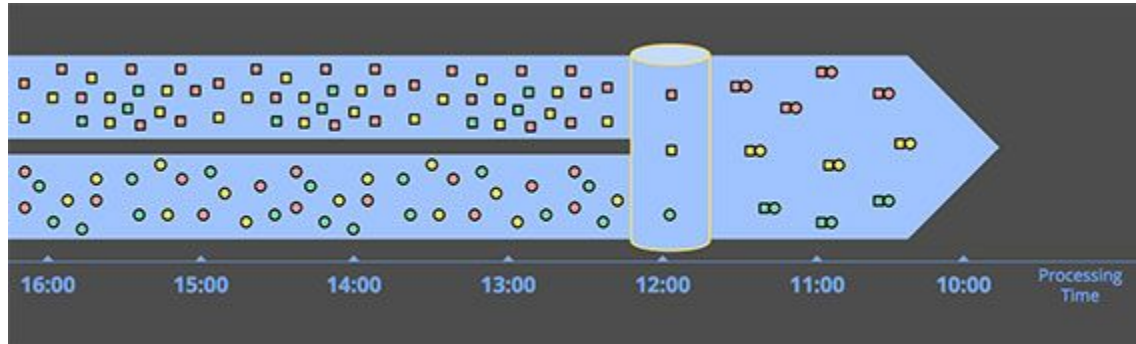


Operations

» Filter

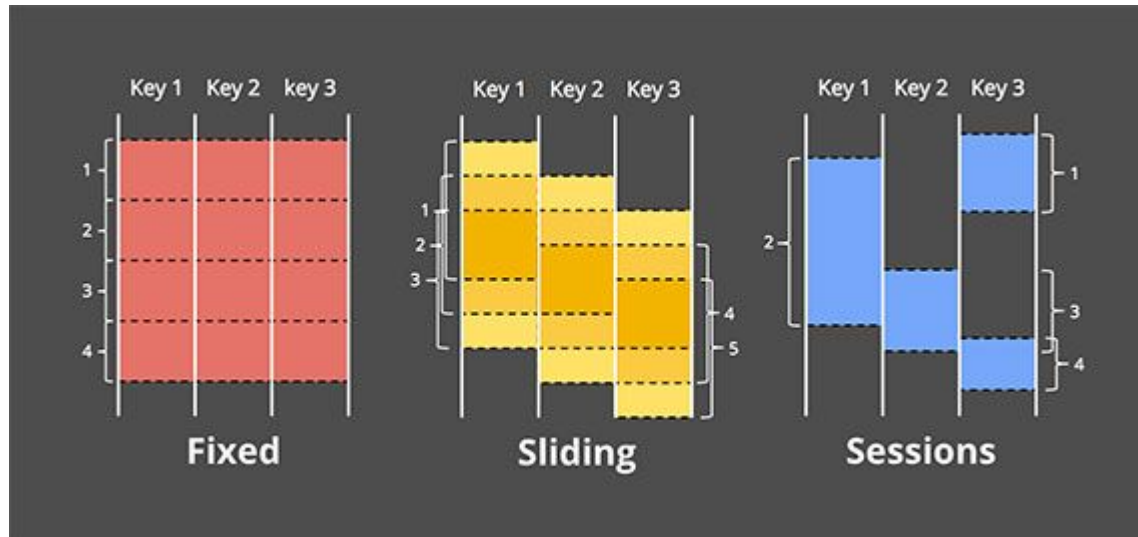


» Join




Windowing

- » Partition a stream of data into discrete batches.
- » Needed to compute metrics that require context (average by minute, partial counts, etc.).

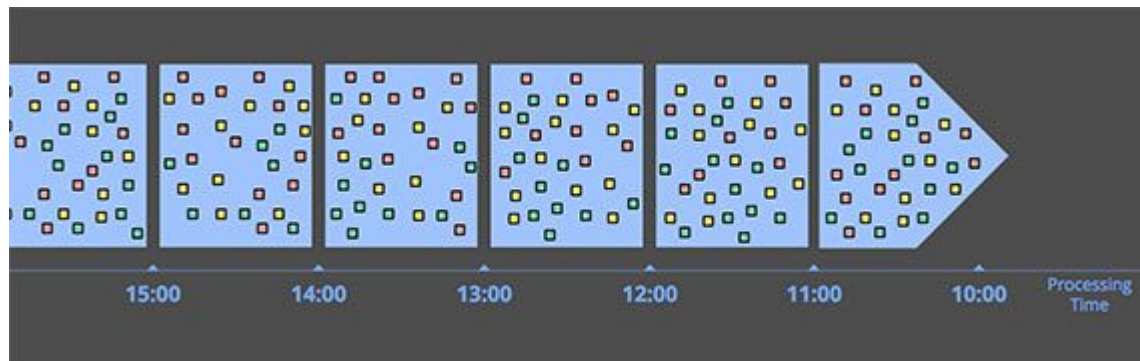




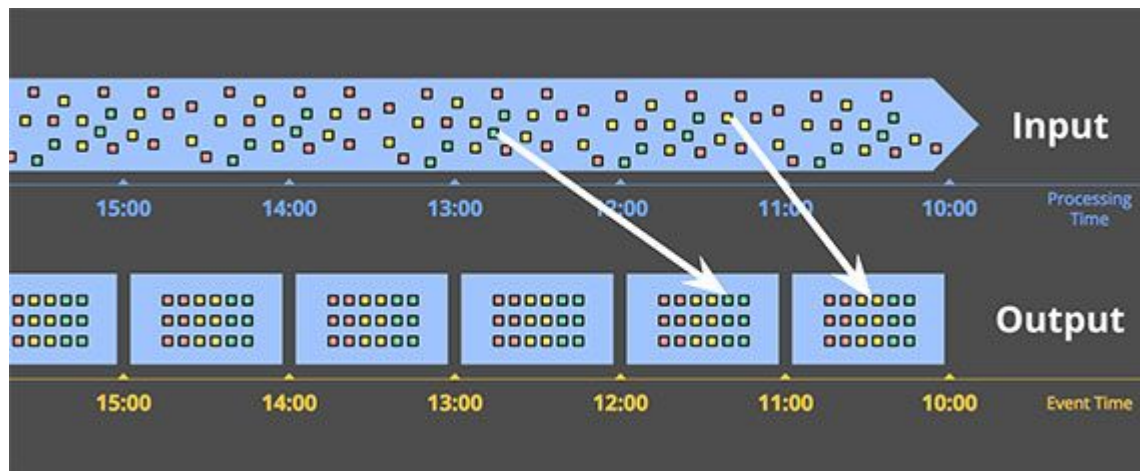
Event Time vs Processing Time

- » **Event Time:** moment when the event actually happened.
 - » **Processing Time:** moment when the data system processes the event
 - » **Ideally equal.** In the real world, there is a significant and variable difference given by:
 - Characteristics of the data source. (distribution, throughput, burstiness, etc.)
 - Hardware. (network, RAM, CPUs, etc.)
 - Processing and transport technologies.
 - Interaction between processes.
- 

Processing Time



Event Time



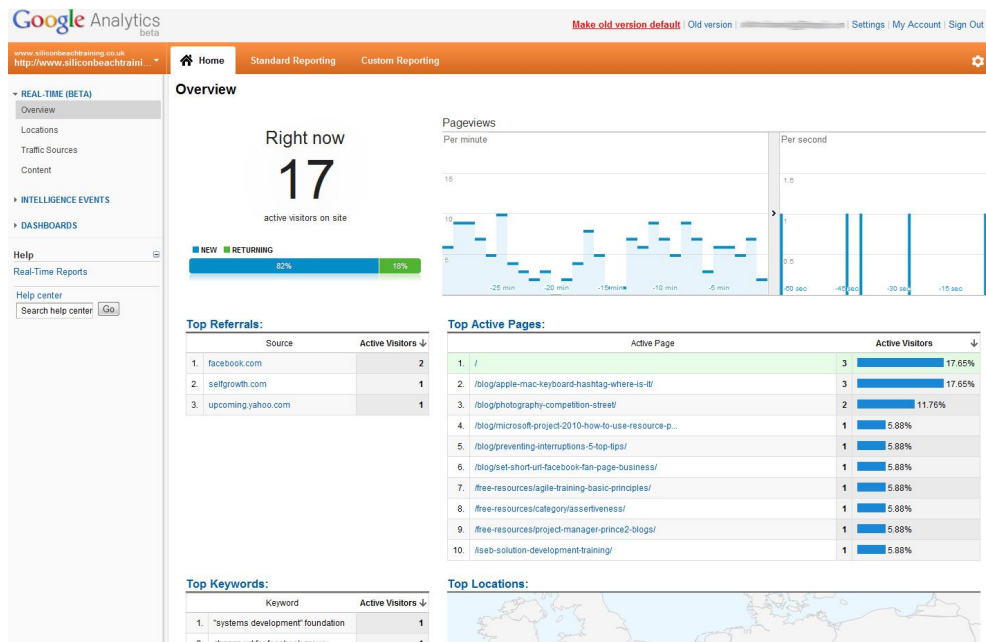


Raw Events vs Aggregated Events

Example: Google Analytics

Analytics solution for keeping track of which pages have been viewed by which visitors.

```
{  
  eventType:    PageViewEvent,  
  timestamp:    1413215518,  
  ipAddress:    12.34.56.78,  
  sessionId:    106d2a521d3c6abcf36,  
  pageUrl:      /talks.html,  
  referrer:     google.com/search?q=...,  
  browser:      Chrome 39  
}
```

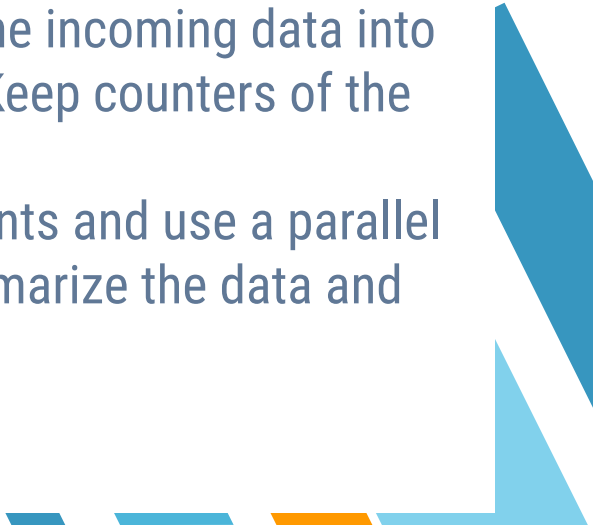




Building an analytics dashboard

Each **page view event** is a **simple, immutable fact**. It simply records that something happened.

How can we build a data system that permits querying capabilities for showing the data in the dashboard?

- » **Datawarehouse and/or OLAP Cubes:** summarize the incoming data into table grouped according to different dimensions. Keep counters of the occurrence of different events.
 - » **Raw events and parallel processing:** store raw events and use a parallel processing engine like Hadoop, Spark, etc. to summarize the data and calculate the needed counters.
- 




Raw Events vs Aggregated Data

Raw events:

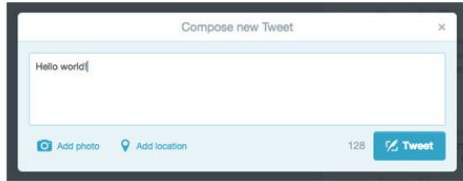
- » Provides a **detailed history** of what happened.
- » Ideal way of **storing data for writes**. (no normalization or multiple updates)
- » Associated with controls that **input data**.
- » Maximum flexibility and Minimum granularity.
- » Needs human or computer error fault tolerance.

Aggregated Data:

- » Provides the **end result** or current state of the system.
 - » Ideal way of storing **data for reads**. (no extra operations needed)
 - » Associated with controls that **display data** (graphs, lists, etc.)
 - » More efficient in memory and computation costs.
 - » Permits reacting quickly to accumulated events.
- 

Example: Twitter

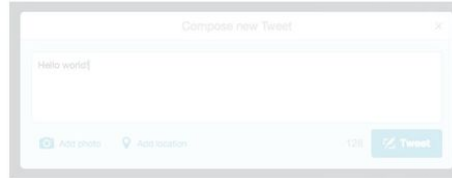
Input (write)



Output (read)



Input (write)



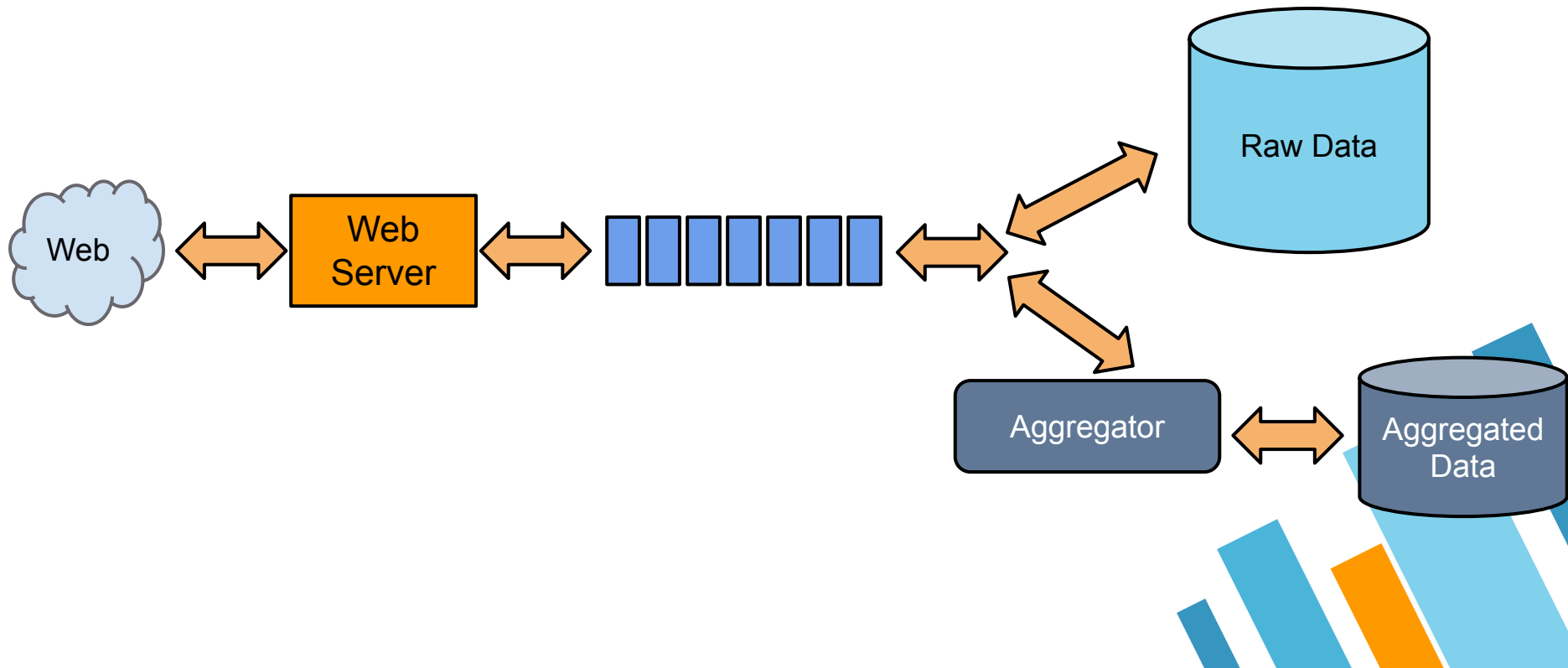
```
{  
  "user_id": 17055506,  
  "timestamp": 1421777578,  
  "status": "Hello world!"  
}
```

Output (read)

```
{  
  "timeline_id": 17055506,  
  "tweets": [  
    {  
      "tweet_id": 557595969962127360,  
      "username": "hotnumbers",  
      "name": "Hot Numbers Coffee",  
      "timestamp": 1421777123,  
      "status": "Open till 9pm tonight",  
      "picture_url": "http://twimg.com/...",  
    },  
    {  
      "tweet_id": 557515007622414337,  
      "username": "Philip Potter",  
      "name": "Philip Potter",  
      "timestamp": 1421777123,  
      "status": "Naive realism proves physics and physics shows naive realism to be false. Therefore naive realism, if true, is false; therefore it is false.",  
      "picture_url": "http://twimg.com/...",  
    },  
    ...  
  ]  
}
```

Multiple tables need to be join for output or multiple tables update on input.

Implementing Both Options





Raw Events and Aggregated Data

Raw Events:


- » **Lower complexity** and amount of data.
- » **Immutable collection** of events.
- » Store them all as a **source of truth**.

Aggregated Data:

- » Can be completely **derived from raw events**.
 - » Can be considered as **cached views** of the raw events.
- 



Why Use Both Schemas?

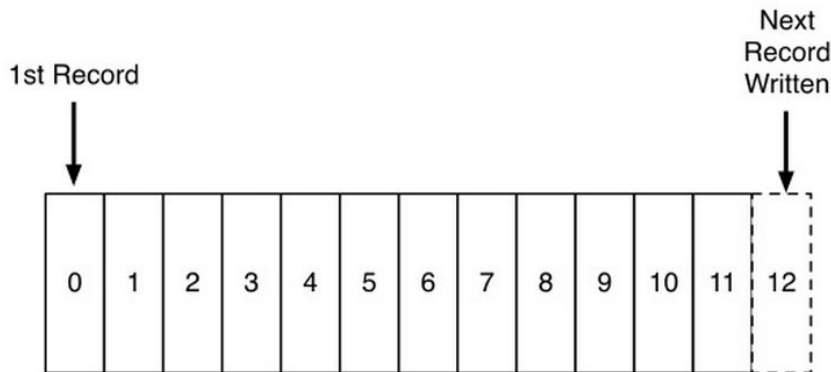
- » **Loose coupling:** different schema for reads and writes means no interdependence.
 - » **Read and write performance:** different schemas for writes and reads mean debate over normalization (faster writes) vs. denormalization (faster reads) ends.
 - » **Scalability:** using immutable raw events and decoupling writes from reads makes parallelization easier which in turn makes scalability easier.
 - » **Flexibility and agility:** multiple different cached views can be derived from the same data in order to compare or implement new features.
 - » **Error recovery:** having the raw events means being able to replay history and re-calculated views any time.
- 



The log

The Log

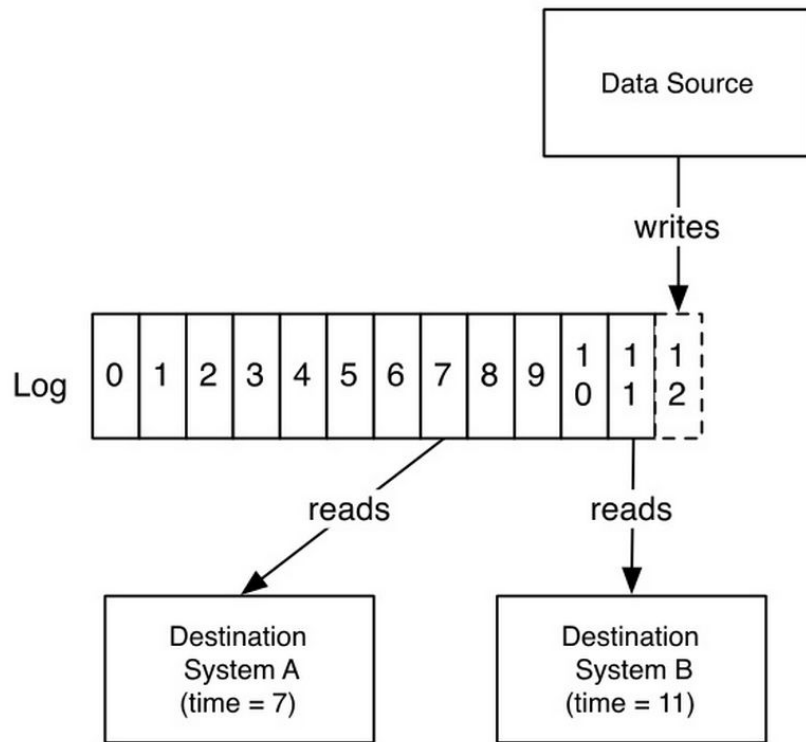
- » **Append-only, totally-ordered sequence of records.**
- » Records are appended to the end of the log, and reads proceed left-to-right. Each entry is assigned a **unique sequential log entry number**.
- » **The ordering of records defines a notion of "time"** since entries to the left are defined to be older than entries to the right.
- » **Databases use them to write out information** about the records they will be modifying, **before applying the changes.(WAL).**



The Log: Events vs Tables

The log is similar to the list of all credits and debits and bank processes; a table is all the current account balances.

If you have a table taking updates, you can **record these changes and publish a "changelog"**. This is exactly what you need to support near-real-time replicas.

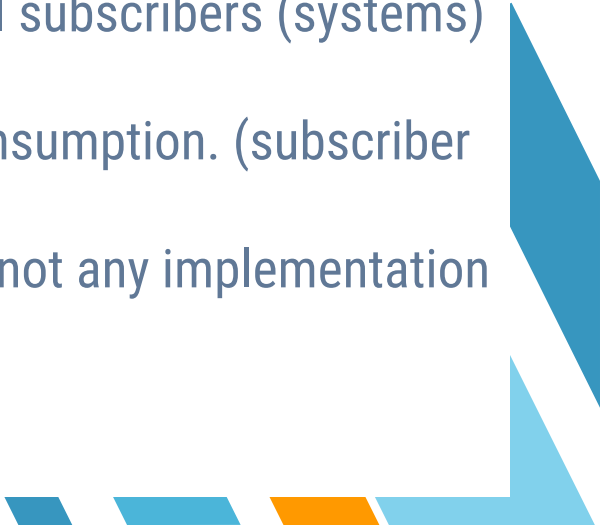




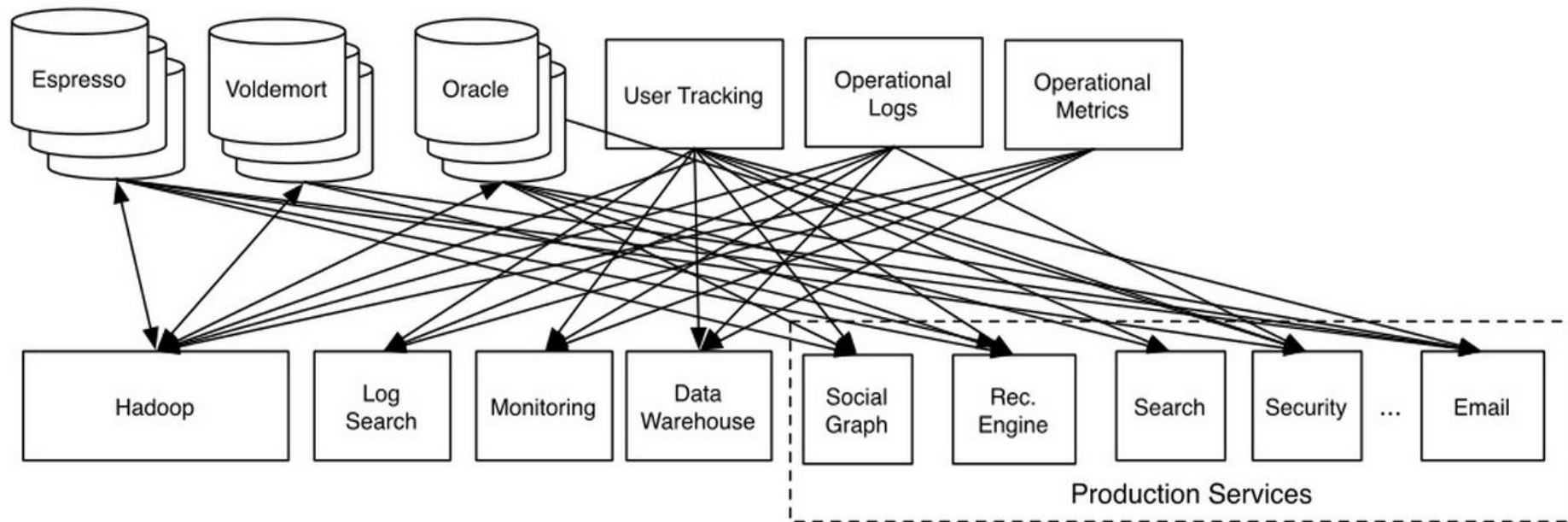
The Log: Data Integration

Big Data in the enterprise **brought specialized data systems** for each kind of use case Hadoop, HBase, Spark, Search, etc.

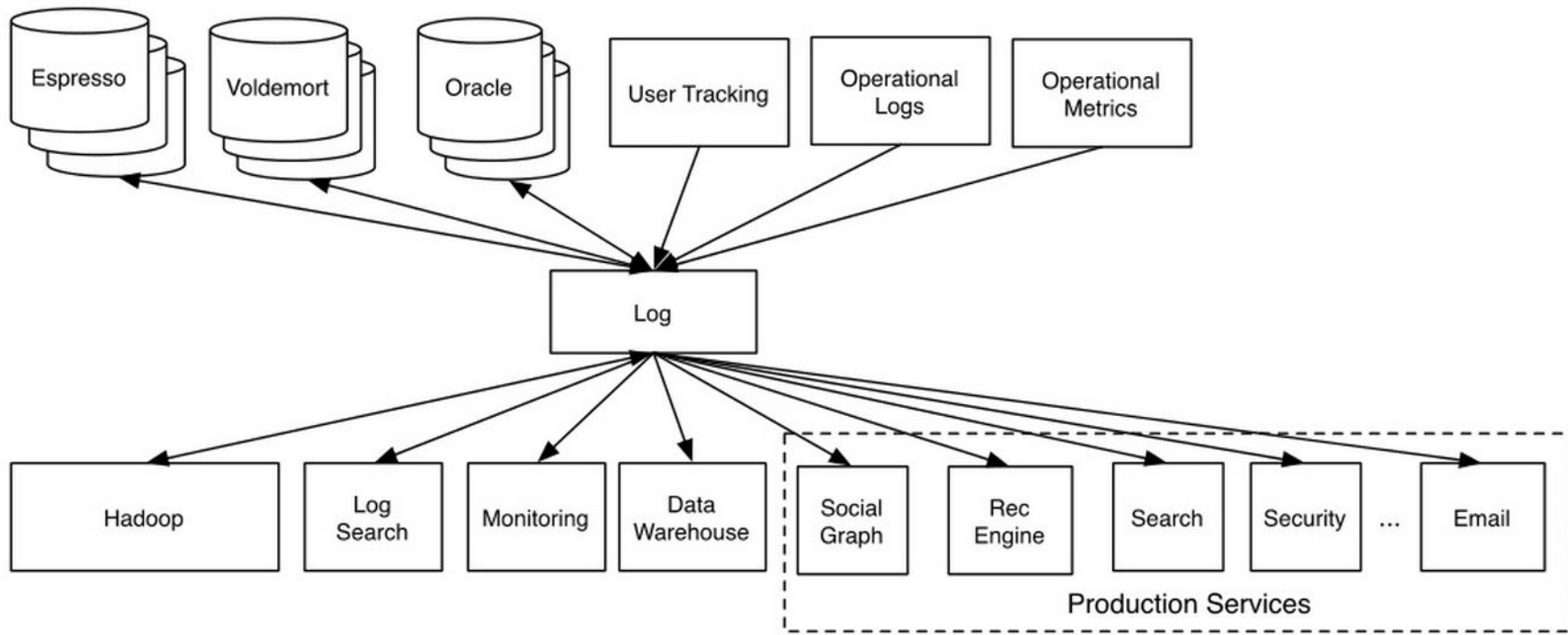
Making all **raw events data available through a centralized persistent order log** would:

- » Gives a **logical clock for each change** against which all subscribers (systems) can be measured.
 - » **Data production becomes asynchronous** from data consumption. (subscriber controls data consumption rate)
 - » **The destination system only knows about the log** and not any implementation details of the system of origin.
- 

The Log: LinkedIn's before infrastructure



The Log: LinkedIn's after infrastructure






Apache Kafka: The Log for Big Data

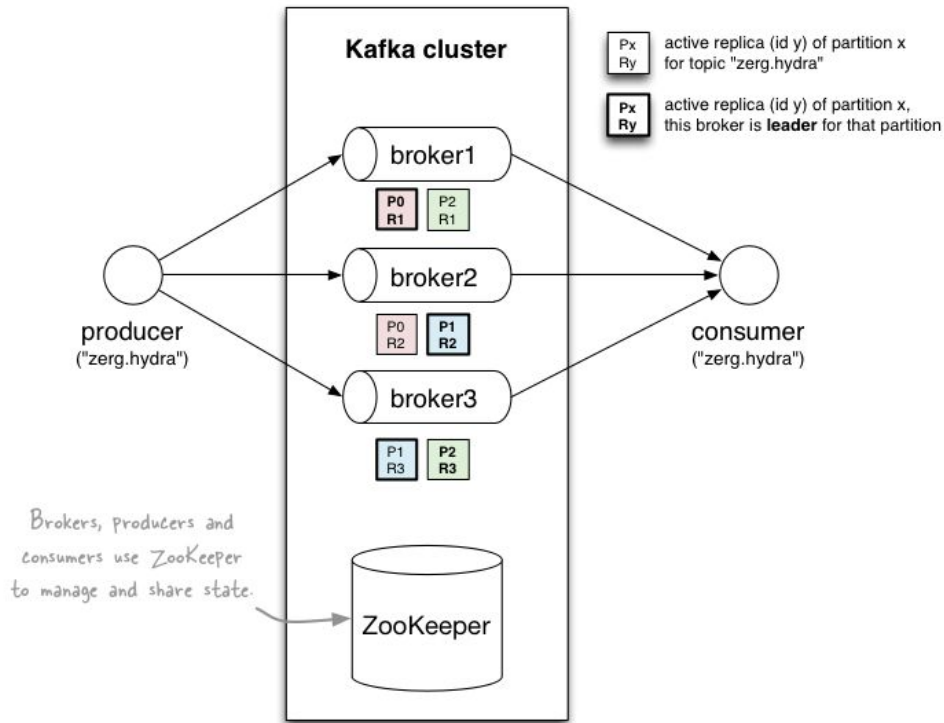


Kafka is a distributed publish-subscribe messaging system that is designed to be fast, scalable, and durable.

- » **Originally developed at LinkedIn**, open sourced in early 2011.
 - » Implemented **in Scala and some Java**.
 - » **In use at:** Netflix, Twitter, Spotify, Loggly, Mozilla, Uber, etc.
 - » The engineers that built Kafka at LinkedIn **created a new company named Confluent** around the project.
- 


Apache Kafka: Concepts

- » **Producers write** data to brokers.
- » **Consumers read** data from brokers.
- » One **broker per node** is recommended for a Kafka cluster.
- » Data is **stored in topics**.
- » Topics are **split into partitions**, which are replicated.
- » Messages are **ordered within a partition but not between partitions**.






Apache Kafka: Concepts

- » By default Kafka **stores the whole history of messages**.
 - » A **message is an array of bytes and has a key** that is also an array of bytes.
 - » Messages are **assigned to a certain partition by the producer** (usually by key).
 - » **Consumers can** either choose to **keep track of the offset** of the last consumed message or Kafka can **do it through Zookeeper**.
 - » **Compaction can be enabled** and in which case only the last message with a certain key is kept (emulates a database table).
 - » The **amount of partitions in a topic determines the level of parallelism** in which data can be consumed from that topic.
 - » The **amount of partitions** need to be **set when the topic is created**.
- 

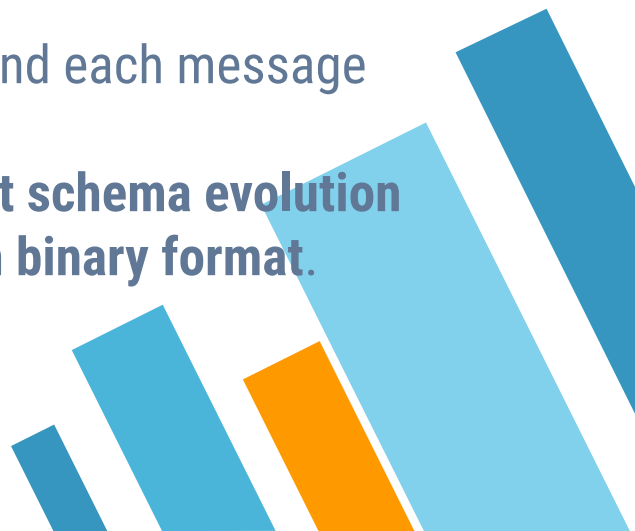


Combining Apache Kafka

- » **Kafka + (Samza, Spark or Storm)** for real-time stream processing.
 - » **Flume or Logstash + Kafka** for log shipping.
 - » **Kafka + Flume** for dumping event/stream data into HDFS.
 - » **Kafka + custom** for data replication, etc.
 - » **Kafka + Kafka Connectors and Kafka + Kafka Stream** to replace everything.
- 



Schemas in Apache Kafka


- » Kafka stores messages as raw bytes so they have no **default schema**.
 - » Usually there is utility code for reading/writing messages shared by the team.
 - » Two ways of managing messages with schema:
 - **Embed** the schema in the message.
 - Create a centralized repository for schemas and each message contains a reference to a schema id.
 - » Ideally, the schema technology used **should permit schema evolution** with backwards compatibility and **serialize data in binary format**.
- 



Avro



Apache Avro

- » Data serialization system that permits **schema evolution**. (add or remove nullable fields and keep reading data serialized by old schemas)
 - » **Schemas are written in JSON.**
 - » Can serialize / deserialize data to and from bytes or JSON.
 - » Support for **many languages** (Java, Python, C++, etc) and permits creating classes from schema definitions.
 - » Can **embed schema** in serialized data or not.
- 

Apache Avro: Example

```
{  
  "namespace": "example.avro",  
  "type": "record",  
  "name": "User",  
  "fields": [  
    {"name": "name", "type": "string"},  
    {"name": "favorite_number", "type": ["int", "null"]},  
    {"name": "favorite_color", "type": ["string", "null"]} ]  
}
```

```
$> java -jar /path/to/avro-tools-1.7.7.jar compile schema user.avsc .
```

```
User user1 = new User();  
user1.setName("Alyssa");  
user1.setFavoriteNumber(256);
```



Dmitriy Ryaboy

@squarecog



Follow

Schemas are an impediment to moving fast the same way street lights are. Useless if you are alone, critical when there's a 100 of you.



RETWEETS

110

FAVORITES

110




12:21 PM - 21 Aug 2014



Streaming Frameworks




Stream Processing Frameworks

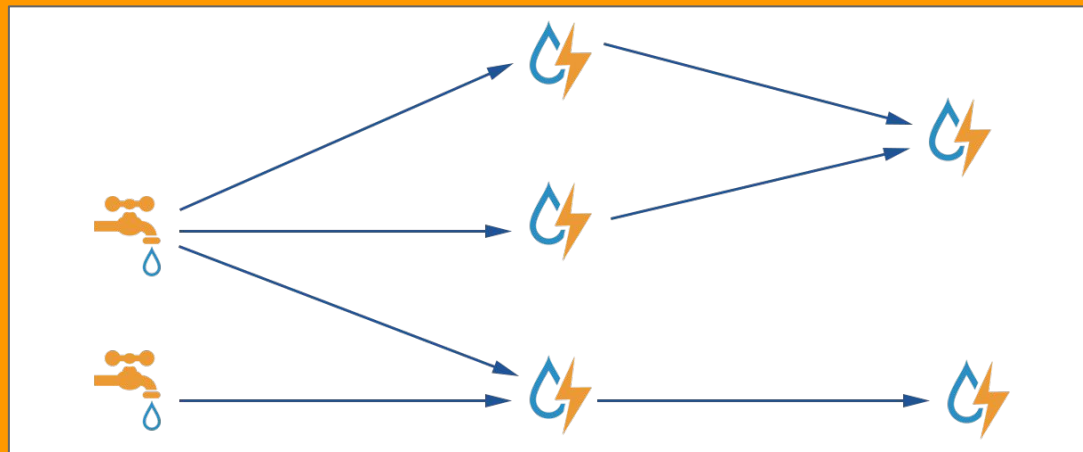
- » **Provide an infrastructure for continuous data processing.**
 - » If the stream processing infrastructure processes data quickly (according to human standards), then it's considered **real-time data processing**.
 - » **Stream processing needs data collected in streams**, that is why Kafka is so fundamental for doing this at scale.
 - » The most widely adopted scalable stream processing frameworks are open source and come from the Apache foundation:
 - Samza
 - Spark
 - Storm
 - Flint
 - Kafka
- 



Stream Processing: Guarantees


- » The simplest mode is **at-most-once delivery**, which drops messages if **they are not processed correctly**, or if the machine doing the processing fails.
 - » There is **also at-least-once** delivery, which tracks whether each input events (and any downstream events it generated) were successfully processed within a configured timeout, by keeping an in-memory record of all emitted events. **Any events that are not fully processed within the timeout are re-emitted.**
 - » **The best guarantee level is exactly-once.** Events are actually processed at least once, but **the frameworks state implementation allows duplicates to be detected and ignored.** The tool needs to implement micro batching (like Spark Streaming) to allow this level of guarantee.
- 

Apache Storm

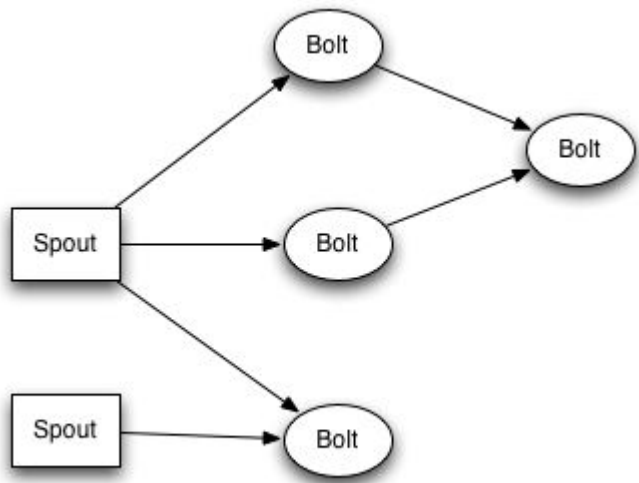




Apache Storm

- » Distributed computation framework written predominantly in the Clojure programming language.
 - » **Originally created by Nathan Marz** and team at BackType, the project was **open sourced after being acquired by Twitter**.
 - » The first open-source Apache stream processing framework **which had real traction**.
 - » Processes messages one at a time, **but processing micro batches can be done through the use of Trident** which **adds primitives for doing stateful, incremental processing** on top of any database or persistence store.
 - » Relies on **external data stores for maintaining state**.
 - » Used at Twitter, Yahoo, Spotify, WeatherChannel, etc.
- 

Apache Storm Topology



- » Storm continuous jobs are executed in topologies **composed by “spouts” and “bolts”**.
- » **A spout is a source of streams** (e.g. connect to the Twitter API and emit a stream of tweets).
- » **A bolt consumes any number of input streams**, does some processing, and possibly emits new streams.

Apache Storm Spout

```
public class TestWordSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    public void open(Map<String, Object> conf, TopologyContext context, SpoutOutputCollector collector) {
        _collector = collector;
    }

    public void nextTuple() {
        Utils.sleep(100);
        final String[] words = new String[] {"nathan", "mike", "jackson", "golda", "bertels"};
        final Random rand = new Random();
        final String word = words[rand.nextInt(words.length)];
        _collector.emit(new Values(word));
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

    public void ack(Object msgId) { }
    public void fail(Object msgId) { }
    public void close() { }
}
```

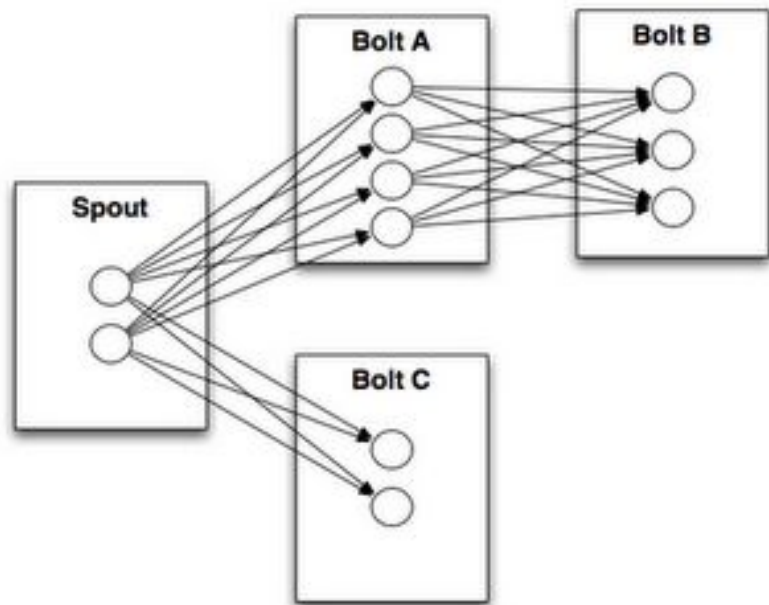
Apache Storm Bolt

```
public static class ExclamationBolt extends BaseRichBolt {  
    OutputCollector _collector;  
  
    @Override  
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {  
        _collector = collector;  
    }  
  
    @Override  
    public void execute(Tuple tuple) {  
        _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));  
        _collector.ack(tuple);  
    }  
  
    @Override  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
        declarer.declare(new Fields("word"));  
    }  
}
```

Apache Storm Configuration

```
public static void main(String[] args) throws Exception {  
  
    Config conf = new Config();  
    conf.setNumWorkers(2); // use two worker processes  
  
    TopologyBuilder builder = new TopologyBuilder();  
    builder.setSpout("words", new TestWordSpout(), 10);  
    builder.setBolt("exclaim1", new ExclamationBolt(), 3).shuffleGrouping("words");  
    builder.setBolt("exclaim2", new ExclamationBolt(), 2).shuffleGrouping("exclaim1");  
  
    boolean runLocal = shouldRunLocal();  
    if(runLocal){  
        LocalCluster cluster = new LocalCluster();  
        cluster.submitTopology("test-topology", conf, builder.createTopology());  
    } else {  
        StormSubmitter.submitTopology("test-topology", conf, builder.createTopology());  
    }  
  
}
```


Apache Storm Stream groupings



Part of defining a topology is specifying **for each bolt which streams it should receive as input. A stream grouping defines how that stream should be partitioned** among the bolt's tasks. There are eight built-in stream groupings in Storm, and you can implement a custom stream grouping.




Apache Storm Stream groupings

- » **Shuffle grouping:** Tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed to get an equal number of tuples.
 - » **Fields grouping:** The stream is partitioned by the fields specified in the grouping.
 - » **Partial Key grouping:** The stream is partitioned by the fields specified in the grouping, but are load balanced between two downstream bolts, which provides better utilization of resources when the incoming data is skewed.
 - » **All grouping:** The stream is replicated across all the bolt's tasks. Use this grouping with care.
- 



Apache Storm Stream groupings

- » **Global grouping:** The entire stream goes to a single one of the bolt's tasks. Specifically, it goes to the task with the lowest id.
 - » **None grouping:** This grouping specifies that you don't care how the stream is grouped.
 - » **Direct grouping:** This is a special kind of grouping. A stream grouped this way means that the producer of the tuple decides which task of the consumer will receive this tuple.
 - » **Local or shuffle grouping:** If the target bolt has one or more tasks in the same worker process, tuples will be shuffled to just those in-process tasks. Otherwise, this acts like a normal shuffle grouping.
- 




Apache Storm Stream groupings

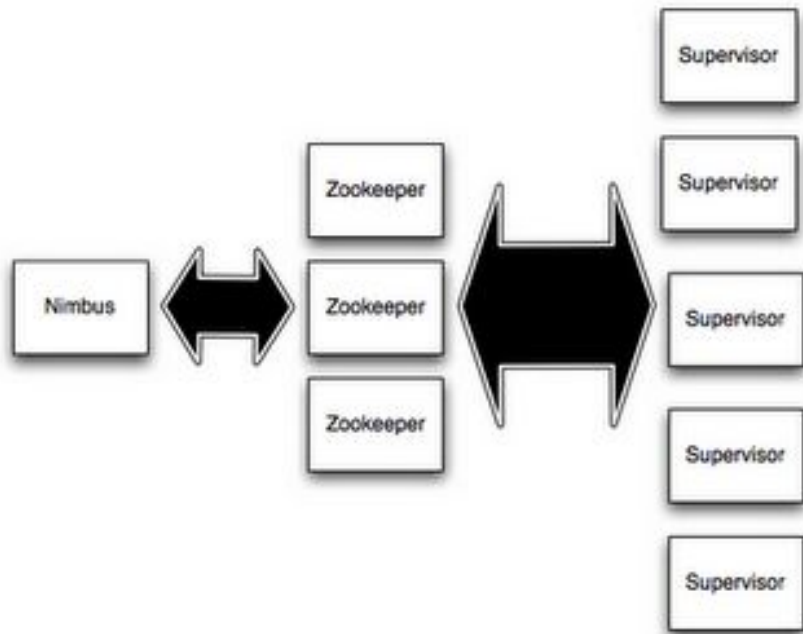
Word count

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("sentences", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(), 8)
    .shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 12)
    .fieldsGrouping("split", new Fields("word"));
```

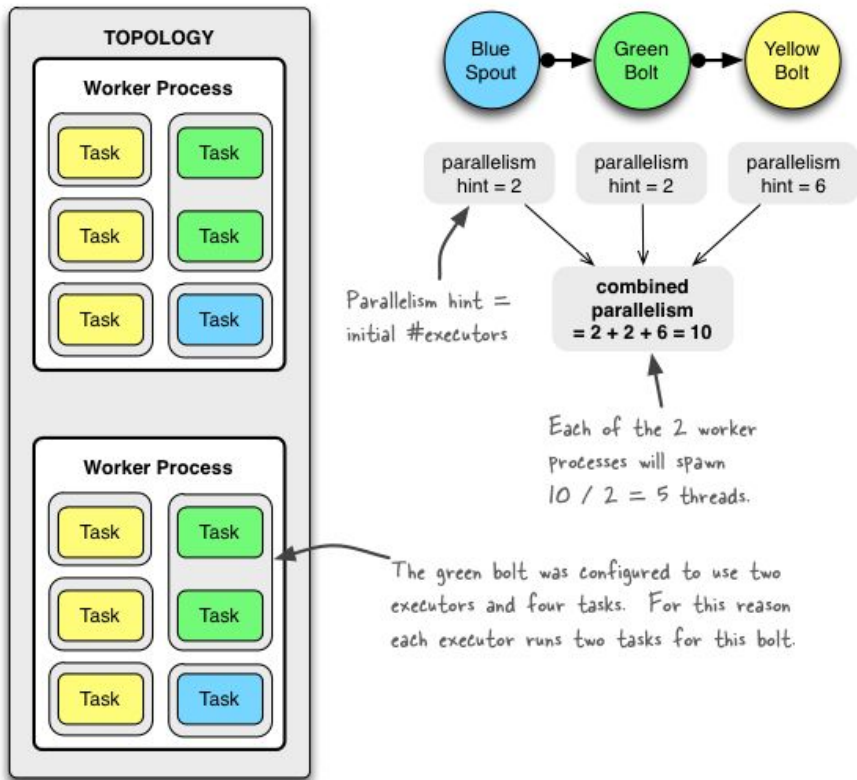


Apache Storm Cluster



- » A Storm cluster has a **master node which runs a daemon called "Nimbus"** that is similar to Hadoop's "JobTracker".
- » **Each worker node** runs a daemon called the **"Supervisor"**.
- » Nimbus **coordinates** with Supervisor **using Zookeeper**.

Apache Storm Cluster



```
Config conf = new Config();
conf.setNumWorkers(2); // use two worker processes
// set parallelism hint to 2
builder.setSpout("blue-spout", new BlueSpout(), 2);

builder.setBolt("green-bolt", new GreenBolt(), 2)
.setNumTasks(4).shuffleGrouping("blue-spout");

builder.setBolt("yellow-bolt", new YellowBolt(), 6)
.shuffleGrouping("green-bolt");

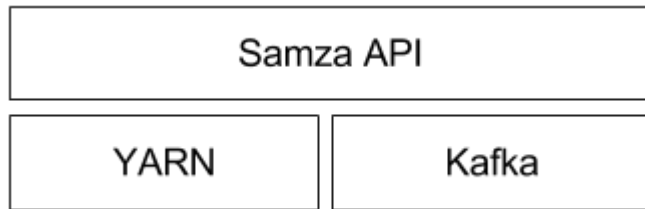
StormSubmitter.submitTopology("mytopology", conf,
builder.createTopology());
```



Apache Samza


Apache Samza

- » **Created at LinkedIn** to work in conjunction with Apache Kafka.
- » Streams are ordered, partitioned, replayable, and fault tolerant.
- » **YARN is used for processor isolation**, security, and fault tolerance.
- » **Mostly written in Scala** but has Java API.





Apache Samza

- » Samza **supports fault-tolerant local state**. State can be thought of as tables that are split up and co-located with the processing tasks. State is itself modeled as a stream. If the local state is lost due to machine failure, the state stream is replayed to restore it.
 - » Processes messages **one at a time**.
 - » Provides 2 abstraction to process messages **StreamTask** and **AsyncStreamTask**
 - » Configuration and Topology definition is made via configuration files.
- 

Apache Samza Tasks

```
public class WordCount implements StreamTask {  
    Map<String, Integer> store = new HashMap<String, Integer>();  
  
    @Override  
    public void process(IncomingMessageEnvelope envelope, MessageCollector collector,  
TaskCoordinator coordinator) {  
        String word = (String) envelope.getMessage();  
        Integer count = store.get(word);  
        if (count == null) count = 0;  
        count++;  
        store.put(word, count);  
        System.out.println("Word: " + word + " / Count: " + count);  
    }  
}
```

Apache Samza Tasks

This is the class above, which Samza will instantiate when the job is run

```
task.class=com.example.samza.Wordcount
```

Define a system called "kafka" (you can give it any name, and you can define

multiple systems if you want to process messages from different sources)

```
systems.kafka.samza.factory=org.apache.samza.system.kafka.KafkaSystemFactory
```

The job consumes a topic called "Words" from the "kafka" system

```
task.inputs=kafka.Words
```

Define a serializer/deserializer called "json" which parses JSON messages

```
serializers.registry.json.class=org.apache.samza.serializers.JsonSerdeFactory
```

```
systems.kafka.streams.Words.samza.msg.serde=json # Use the "json" serializer for messages in the "Words" topic
```



Kafka Streams



Kafka Streams

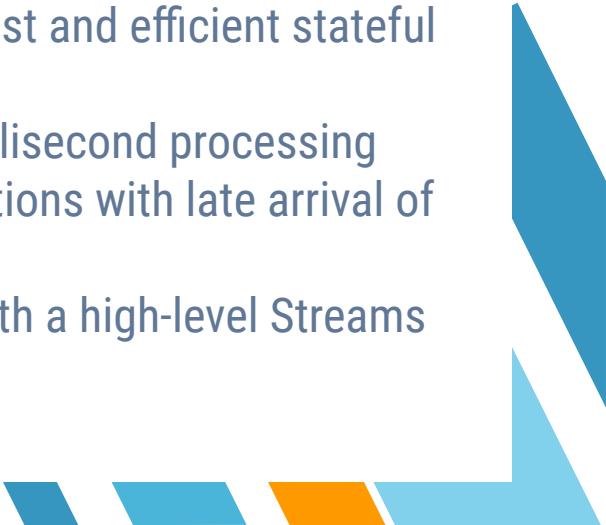
Kafka Streams is a **client library** for **processing and analyzing data stored in Kafka** and either write the resulting data back to Kafka or send the final output to an external system.

It builds upon important stream processing concepts such as properly distinguishing between event time and processing time, windowing support, and simple yet efficient management of application state.





Kafka Streams

- » Designed as a **simple and lightweight client library**, which can be easily embedded in any Java application.
 - » 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.
 - » **Employs one-record-at-a-time processing** to achieve millisecond processing latency, and supports event-time based windowing operations with late arrival of records.
 - » **Offers necessary stream processing primitives**, along with a high-level Streams DSL and a low-level Processor API.
- 



Kafka Streams

Kafka Streams is a **client library** for **processing and analyzing data stored in Kafka** and either write the resulting data back to Kafka or send the final output to an external system.

It builds upon important stream processing concepts such as properly distinguishing between event time and processing time, windowing support, and simple yet efficient management of application state.c



Kafka Streams WordCount

```
public class WordCountDemo {  
    public static void main(String[] args) throws Exception {  
        Properties props = new Properties();  
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-wordcount");  
        Pattern pattern = Pattern.compile("\\W+", Pattern.UNICODE_CHARACTER_CLASS);  
  
        KStreamBuilder builder = new KStreamBuilder();  
        KStream<String, String> source = builder.stream("streams-file-input");  
  
        KStream<String, Long> wordCounts = source  
            .flatMapValues(value-> Arrays.asList(pattern.split(value.toLowerCase())))  
            .map((key, word) -> new KeyValue<>(word, word))  
            .countByKey("Counts")  
            .toStream();  
    }  
}
```




Kafka Streams WordCount

```
counts.to(Serdes.String(), Serdes.Long(), "streams-wordcount-output");

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

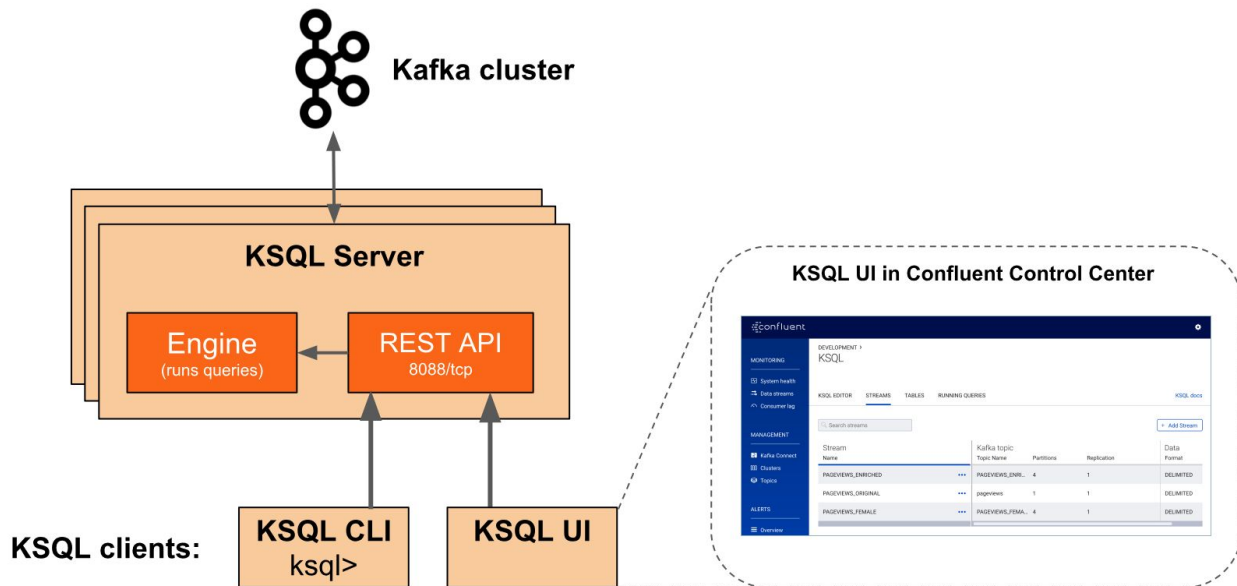
// usually the stream application would be running forever,
// in this example we just let it run for some time and stop since the input data is finite.
Thread.sleep(5000L);

streams.close();
}
}
```

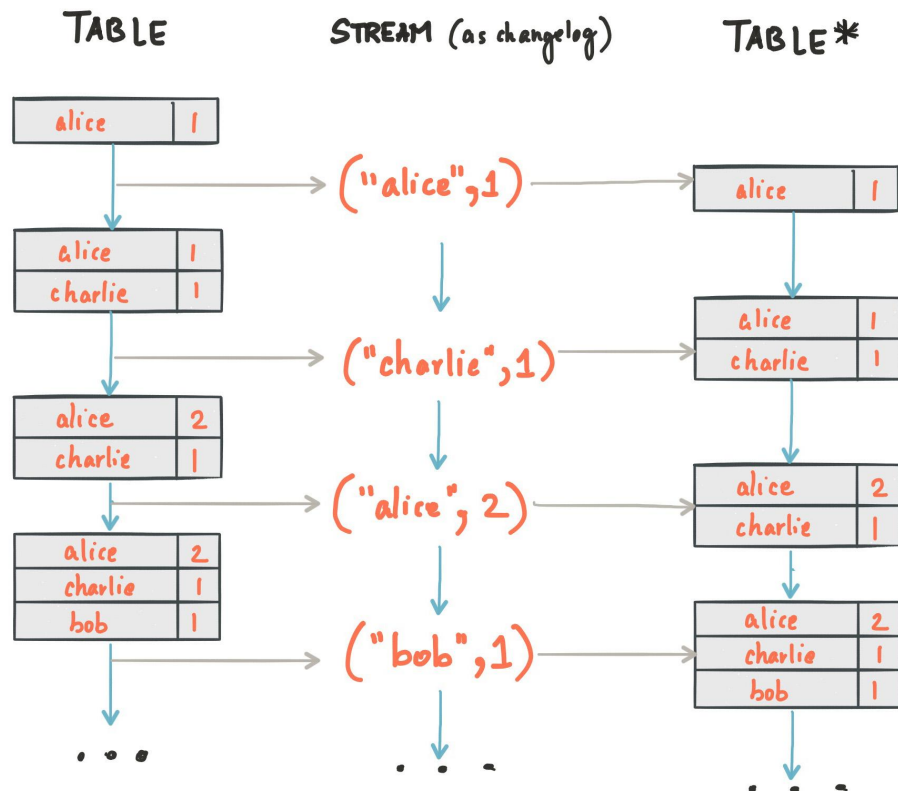


KSQL

KSQL architecture and components



KSQL






KSQL

Para generar un stream de un tópico

```
CREATE STREAM pageviews
(viewtime BIGINT,
userid VARCHAR,
pageid VARCHAR)
WITH (KAFKA_TOPIC='pageviews',
VALUE_FORMAT='DELIMITED',
KEY='pageid',
TIMESTAMP='viewtime');
```



KSQL

Para generar una tabla de un tópico

```
CREATE TABLE users
  (registertime BIGINT,
   gender VARCHAR,
   regionid VARCHAR,
   userid VARCHAR,
   interests array<VARCHAR>,
   contactinfo map<VARCHAR, VARCHAR>)
WITH (KAFKA_TOPIC='users',
      VALUE_FORMAT='JSON',
      KEY = 'userid');
```


KSQL

De un stream puedo consultar y generar otro stream.

```
CREATE STREAM pageviews_transformed_priority_1
  WITH (TIMESTAMP='viewtime', PARTITIONS=5, VALUE_FORMAT='JSON') AS


  SELECT viewtime,
         userid,
         pageid,
         TIMESTAMPTOSTRING(viewtime, 'yyyy-MM-dd HH:mm:ss.SSS') AS timestring
  FROM pageviews
  WHERE userid='User_1' OR userid='User_2'
  PARTITION BY userid;
```



KSQL

También puedo generar tablas (y usar operaciones de tiempo como ventanas)

```
CREATE TABLE pageviews_per_region_per_minute AS
  SELECT regionid,
         count(*) FROM pageviews_enriched
  WINDOW TUMBLING (SIZE 1 MINUTE)
  GROUP BY regionid;
```


- Hopping Window: is Time-based and has Fixed-duration, overlapping windows
 - Tumbling Window: is Time-based and has Fixed-duration, non-overlapping, gap-less windows
 - Session Window: is Session-based and has Dynamically-sized, non-overlapping, data-driven windows
- 



KSQL

Join

```
CREATE STREAM pageviews_enriched AS
  SELECT pv.viewtime,
         pv.userid AS userid,
         pv.pageid,
         pv.timestring,
         u.gender,
         u.regionid,
         u.interests,
         u.contactinfo
  FROM pageviews_transformed pv
  LEFT JOIN users u ON pv.userid = u.userid;
```






KSQL

Para tener la salida persistente en otro tópico se puede generar un stream o tabla y asignarle un tópico.

```
CREATE STREAM visitors WITH (kafka_topic=visitors, value_format='delimited')  
AS SELECT age FROM pageviews_enriched;
```






Cloud processing

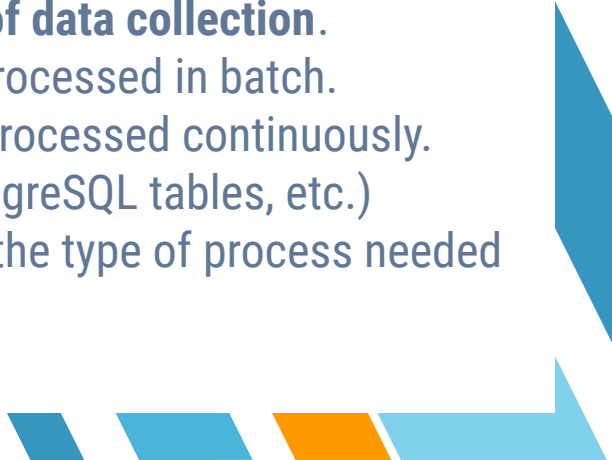
Beyond the possibility to deploy this applications on a cloud server, or even consuming "hosted" versions of the processing frameworks, cloud services provide "serverless computation services". This services allow to deploy code, directly into the service, point it to the data to process and wait for the results.

The key players are:

- » [Amazon lambda](#)
 - » [Google Cloud Functions](#)
 - » [Azure Functions](#)
- 




Key Take Aways

- » **Batch** data processing can be considered as a **specialized case of stream processing**.
 - » **The Log is the unifying abstraction** that permits real-time data processing and syncing between all the specialized data systems in an enterprise.
 - » **Kafka is the becoming the defacto standard** to enable a scalable, durable, enterprise-wide **event log**.
 - » **The real driver for the processing model is the method of data collection.**
 - When Data which is collected in batch is naturally processed in batch.
 - When data is collected continuously, it is naturally processed continuously.
 - » We can use any store for persisting a log (HDFS, S3, PostgreSQL tables, etc.)
 - » The processing framework to choose depends mainly in the type of process needed for the application.
- 



References

- » <http://radar.oreilly.com/2015/08/the-world-beyond-batch-streaming-101.html>
 - » <https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>
 - » <http://blog.confluent.io/2015/01/29/making-sense-of-stream-processing/>
 - » <http://blog.confluent.io/2015/02/25/stream-data-platform-1/>
 - » <http://blog.confluent.io/2015/02/25/stream-data-platform-2/>
 - » <http://blog.confluent.io/2015/03/04/turning-the-database-inside-out-with-apache-samza/>
 - » <http://www.cakesolutions.net/teamblogs/comparison-of-apache-stream-processing-frameworks-part-1>
- 



References

- » <http://www.michael-noll.com/blog/2014/08/18/apache-kafka-training-deck-and-tutorial/>
 - » <http://blog.cloudera.com/blog/2014/09/apache-kafka-for-beginners/>
 - » <http://samza.apache.org/learn/documentation/latest/comparisons/introduction.html>
 - » <https://storm.apache.org/documentation/Tutorial.html>
 - » <http://www.slideshare.net/gwenshap/kafka-for-dbas>
- 



CREDITS

Content of the slides:

- » Big Data Tools - ITBA

Images:

- » Big Data Tools - ITBA
- » obtained from: commons.wikimedia.org

Special thanks to all the people who made and released these awesome resources for free:

- » Presentation template by [SlidesCarnival](https://slidescarnival.com)
 - » Photographs by [Unsplash](https://unsplash.com)
- 