ADVANCED DATABASES

TERM PROJECT

STREAM DATA PLATFORMS – APACHE KAFKA

KANATOV, KARİM

ZEYBEK, ALİ BAHADIR

TABLE OF CONTENTS

ABSTRACT

Throughout this paper, one of the emerging concepts in the database world – Stream Data Platforms and what are they useful for will be discussed. Moreover, Apache Kafka – a publish/subscribe messaging rethought – will be demonstrated to better understand specific use cases of Stream Data Platforms.

INTRODUCTION

60 seconds. Let us think for a while what might happen on internet within that short period of time. According to DOMO[1], each minute of every day the following happens on the Internet:

- 347,222 Tweets sent!
- More than 300 Hours of content uploaded to YouTube!
- Up to 2 Billion photos liked in Instagram.
- 4.1 BILLION Facebook user likes posts!
- Amazon receives 4,310 unique visitors.
- Apple users download 51,000 apps

All of these happen only within one minute.

According to Gartner's, Inc. forecasts from 2014[2], we will have around 5 billion connected things (Internet of Things) used in 2015. In other words, those things will generate more data and the data will be streamed in real time. More interestingly, the majority of data that is used in internet today are created by individual users via social media. It is never ending feed of information.

Globally 3.2 billion people are using the Internet by end 2015, of which 2 billion are from developing countries[3], which is 45% of worldwide population. It means there will be more data in coming years. Those data will grow exponentially and will require to be processed in real time.

[1] https://www.domo.com/blog/2015/08/data-never-sleeps-3-0/

[2] https://www.gartner.com/newsroom/id/2905717

[3] https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2015.pdf

Now let us switch the topic to a different perspective. In most of the traditional business applications, there are certain changes of events that are needed to be captured and act accordingly. Even a simple shopping site has different events such as user login – add to the cart – payment – shipment – etc. The question here is that how much of those event changes can be captured?

Normally, what most of the applications using traditional RDBMS's is changing the state of corresponding entry in the relation to the most up-to-date value. However, underlying procedures that builds up to the final state are usually overridden.

Stream Data Platform, is basically good for two main concepts:

- **Data Integration:** The stream data platform captures streams of events or data changes and feeds these to other data systems such as relational databases, key-value stores, Hadoop, or the data warehouse.
- **Stream Processing:** It enables continuous, real-time processing and transformation of these streams and makes the results available system-wide.

The demand for stream processing is increasing a lot these days. Often processing huge volumes of data is not enough. Data has to be processed fast, so that a firm can react to changing business conditions in real time. This is required for trading, fraud detection, system monitoring, and many other examples.

Generally, gathering the data depends on the data sources. Events coming from databases, machine-generated logs, and even sensors need to be cleaned, schematized and forwarded to a central place.

Secondly, one of the main goals on development of Kafka was collecting the streams in one central place. So, brokers collect stream of data, log and buffer them in a fault tolerant manner, as well as distribute them among the various consumers that are interested in the streams of data. Finally, analysing the stream of data, which can be performed well using other Apache solutions like Samza or Storm.

This paper is consisted of basic understanding of the stream processing and the role of Kafka on it. We will discuss not only theoretical, but also will have a look at practical use of existing solution.

Further we will discuss the use of Kafka in detail. We will look at configuration, design choice and implementation of Kafka. Also, the use of Kafka in stream processing will be discussed in detail.

STREAM PROCESSING, MESSAGING, DATA WAREHOUSE

Data Warehouses are well known for storing and analysing the structured data. Running simple query on top of terabytes of data is enough to get any historical data within seconds.

However the ETL processes often take too long. As business grows, shareholders want to query up-to-date information, since information from the day before is not sufficient enough to make precise decisions. This is where stream processing comes in and feeds all new data into the Data Warehouse immediately.

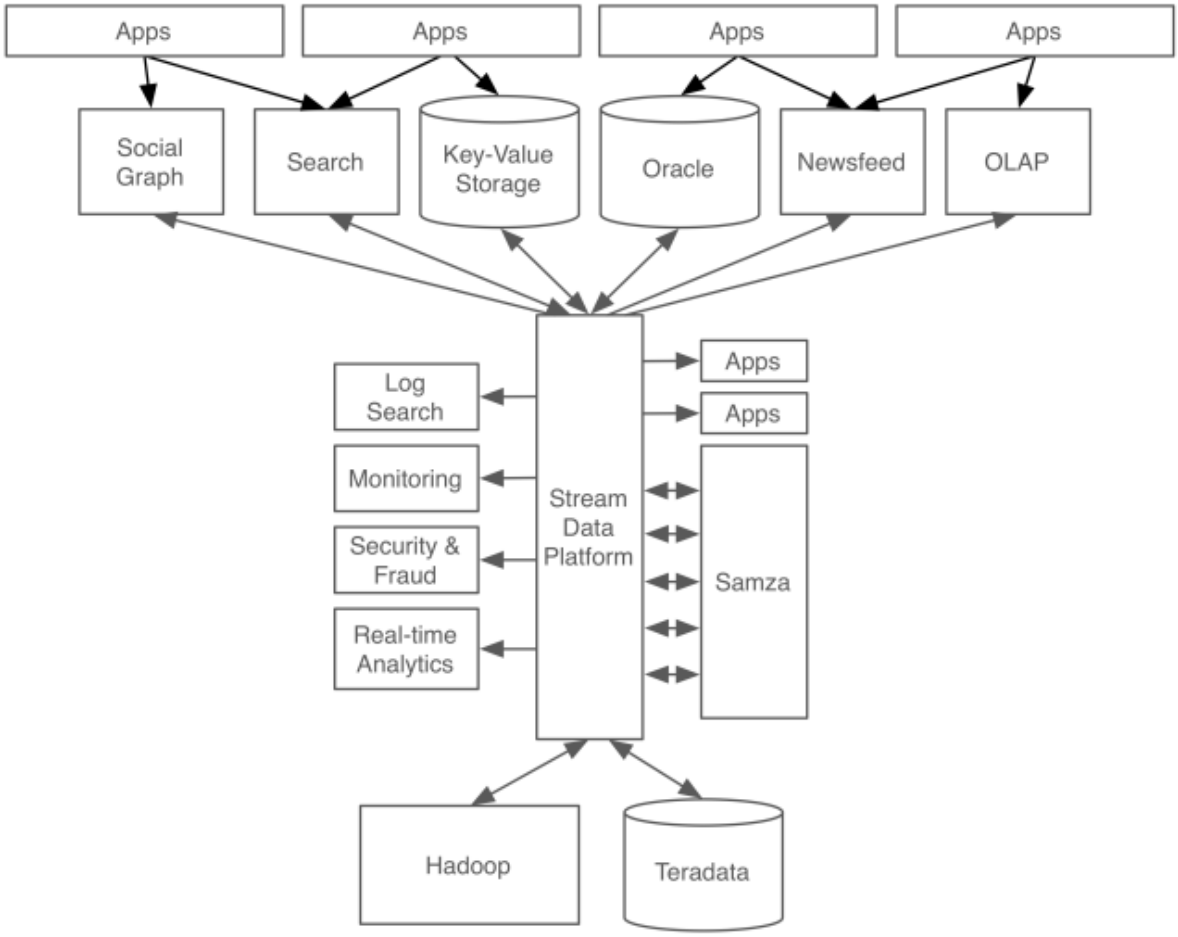


Figure 1. Stream Data Platform architecture build around Kafka

The problem of most big companies is the use of different data systems: relational OLTP databases, Hadoop, Teradata, a search system, monitoring systems, OLAP stores and so on. Therefore, each of these systems needs reliable feeds of data.

One of the main problems that Kafka solves is transportation of data between those systems. Once the data is in Kafka, consumers can do whatever is needed. Reliability and low latency is the key advantages of using Kafka.

,

USE CASES

One of the first commercial uses of stream processing was found in the finance industry, as stock exchanges moved from floor-based trading to electronic trading. Today, we can confidently say it is used almost in every industry - anywhere where stream of data are generated. The Internet of Things will increase volume, variety and velocity of data, leading to a dramatic increase in the applications for stream processing technologies.

Kafka has been developed by LinkedIn engineering team; so, it was the first place where the concept and real abilities of the system have been tested. As Kafka founder, Jay Kreps, states the existing monolithic, centralized database began facing its limits and there was a need to start the transition to a portfolio of specialized distributed systems[4]. So, Kafka was built to serve as a central repository of data stream. The most interesting is that indeed engineers underestimate the power of log and most of the time we do not accept logs as an innovation even though it has been around as long as computer.

Kafka can serve as a message broker, web activity tracker, log aggregation solutions. Nowadays all the stream processing solution such as Apache Storm and Samza goes together with Kafka as messaging stream and are well implemented with each other.

Kafka lets us see the data not as a data itself but accept the data as a stream of events. For instance, retail has events such as logistic, sales, orders, refunds and so on. Web sites have streams of clicks, impressions, searches, and so on. Big software systems have streams of requests, errors, machine metrics, and logs. So, the idea behind all of that are collecting, storing and the use of all those events for better understanding of users, customers' behaviour.

Most of the e-commerce websites do not only want to know the final result of customer purchase, but also they would like to see the logical path of user choice, which could be perceived as stream of events. Those streams of events help the company to improve their service and price policy while it is also good for marketing purpose.

---

[4]https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

Thus, much of what people refer to when they talk about "big data" is really the act of capturing these events that previously weren't recorded anywhere and putting them to use for analysis, optimization, and decision making. In some sense these events are the other half of the story the database tables don't tell: they are the story of what the business did.

COMPLEX EVENT PROCESSING

Kafka is well known as a system for high-throughput reliable event streams. So, on the processing side most of the time developers choose some stream processing frameworks like Samza, Storm or Spark Streaming.

On other hand, there is also some work on high-level query languages for stream processing, and Complex Event Processing is especially worth mentioning. It originated in 1990s research on event-driven simulation, and most CEP products are commercial, expensive enterprise software. Esper is the only free and open source solution, but it is limited to running on a single machine.

With CEP, queries or rules that match certain patterns in the events are written. In other words, CEP is about detecting and selecting the interesting events (and only them) from multiple sources, finding their relationships and deducing new data from them and their relationships. The goal of complex event processing is to identify meaningful events (such as opportunities or threats) and respond to them as quickly as possible

They are comparable to SQL queries (which describe what results you want to return from a database), except that the CEP engine continually searches the stream for sets of events that match the query and notifies you (generates a "complex event") whenever a match is found. This is useful for fraud detection or monitoring business processes, for example.

For use cases that can be easily described in terms of a CEP query language, such a high-level language is much more convenient than a low-level event processing API. On the other hand, a low-level API gives you more freedom, allowing you to do a wider range of things than a query language would let you do. Also, by focussing their efforts on scalability and fault tolerance, stream processing frameworks provide a solid foundation upon which query languages can be built.

Nowadays, as CEP is supposed to be one of the main topic in IT industry, many vendors offer CEP solutions. Many of the CEP tools on the market allow the creation of real-

time, event-driven applications. These applications might consumes data from multiple external sources, but they can also consume data from traditional database sources.

Most of these products include a graphical event flow language and support SQL. Key commercial vendors in this space are IBM with IBM Operational Decision Manager, Informatica with RulePoint, Oracle with its Complex Event Processing Solution, Microsoft's StreamInsights, and SAS DataFlux Event Stream Processing Engine, and TIBCO's CEP. Numerous start-ups are emerging in this market.

APACHE KAFKA

A Kafka cluster can accept tens of millions of writes per-second and persist them for days, months, or indefinitely. Each write is replicated over multiple nodes for fault-tolerance and a Kafka cluster can support thousands of concurrent readers and writers. This journal acts as the subscription mechanism for other systems to feed off of. The Kafka cluster can expand dynamically to handle greater load or retain more data.

Kafka is a distributed, partitioned, replicated commit log service[5]. In other words, it is publish-subscribe messaging implemented as a distributed commit log suitable not only for online, but also for offline message consumption. In general Kafka is made of three levels:

- Producers - the processes that publish messages to a Kafka topic.
- Broker - is Kafka cluster consisting of one or more servers.
- Consumers - the processes that subscribe to topics and process the feed of published messages.

Kafka maintains feeds of messages in categories called topics. For each topic Kafka maintains at least one partition block. Partition block contains messages in unchangeable sequence and each message is continually appended to a commit log. Each message is identified by unique sequence of ID called offset. All published messages are retained in the Kafka cluster for a configured period of time, after that those messages will be discarded t free up the space.

Partitions contain messages that are replicated over multiple servers, which intern form Kafka broker. This is done for a fault tolerance. Every partition has one server acting as a leader, the rest of them as followers. Reader handles all read/write requests for the partition. The follower passively replicates the leader. If the leader server fails, one of the follower servers becomes a leader by default making the broker more resilient against host failures.

[5] https://kafka.apache.org/documentation.html#messages

In the consumer side, the message can be broadcasted to all consumers or each message can be read one by one from the server.

QUICK START

- **Download the Code & Un-Tar it**

  Code Source:

  https://www.apache.org/dyn/closer.cgi?path=/kafka/0.9.0.0/kafka_2.11-0.9.0.0.tgz

  Un-Tar:

  > tar -xzf kafka_2.11-0.9.0.0.tgz
  > cd kafka_2.11-0.9.0.0

- **Start the Server**

  Kafka uses ZooKeeper so you need to first start a ZooKeeper server if you don't already have one. You can use the convenience script packaged with Kafka to get a quick-and-dirty single-node ZooKeeper instance.

  > bin/zookeeper-server-start.sh config/zookeeper.properties

  Now start the Kafka server:

  > bin/kafka-server-start.sh config/server.properties

- **Create a Topic**

  Create a topic named "test" with a single partition and only one replica:

  > bin/kafka-topics.sh --create --zookeeper localhost:2181
  --replication-factor 1 --partitions 1 --topic test

We can now see that topic if we run the list topic command:

> bin/kafka-topics.sh --list --zookeeper localhost:2181

test

Alternatively, instead of manually creating topics you can also configure your brokers to auto-create topics when a non-existent topic is published to.

- **Send Messages**

Kafka comes with a command line client that will take input from a file or from standard input and send it out as messages to the Kafka cluster. By default each line will be sent as a separate message.

Run the producer and then type a few messages into the console to send to the server.

> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test

Hello Kafka

This is an event that is streamed…

- **Consume Messages**

Kafka also has a command line consumer that will dump out messages to standard output.

> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning

Hello Kafka

This is an event that is streamed…

If you have each of the above commands running in a different terminal then you should now be able to type messages into the producer terminal and see them appear in the consumer terminal.

All of the command line tools have additional options; running the command with no arguments will display usage information documenting them in more detail.

- **Setting Up a Multi-Broker Cluster**

So far we have been running against a single broker, but that's no fun. For Kafka, a single broker is just a cluster of size one, so nothing much changes other than starting a few more broker instances. But just to get feel for it, let's expand our cluster to three nodes (still all on our local machine).

First we make a config file for each of the brokers:

> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties

Now edit these new files and set the following properties:

config/server-1.properties:
broker.id=1
port=9093
log.dir=/tmp/kafka-logs-1

config/server-2.properties:
broker.id=2
port=9094
log.dir=/tmp/kafka-logs-2

The broker.id property is the unique and permanent name of each node in the cluster. We have to override the port and log directory only because we are running these all on the same machine and we want to keep the brokers from all trying to register on the same port or overwrite each other's' data.

We already have Zookeeper and our single node started, so we just need to start the two new nodes:

> **bin/kafka-server-start.sh config/server-1.properties &**
> **bin/kafka-server-start.sh config/server-2.properties &**

Now create a new topic with a replication factor of three:

> **bin/kafka-topics.sh --create --zookeeper localhost:2181**
--replication-factor 3 --partitions 1 --topic my-replicated-topic

Okay but now that we have a cluster how can we know which broker is doing what? To see that run the "describe topics" command:

> **bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic**
my-replicated-topic

```
Topic:my-replicated-topic    PartitionCount:1
ReplicationFactor:3          Configs:
Topic: my-replicated-topic   Partition: 0        Leader: 1
Replicas: 1,2,0      Isr: 1,2,0
```

Here is an explanation of output. The first line gives a summary of all the partitions, each additional line gives information about one partition. Since we have only one partition for this topic there is only one line.

- "leader" is the node responsible for all reads and writes for the given partition. Each node will be the leader for a randomly selected portion of the partitions.
- "replicas" is the list of nodes that replicate the log for this partition regardless of whether they are the leader or even if they are currently alive.
- "isr" is the set of "in-sync" replicas. This is the subset of the replicas list that is currently alive and caught-up to the leader.

Note that in this example node 1 is the leader for the only partition of the topic.

We can run the same command on the original topic we created to see where it is:

> **bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test**

```
Topic:test          PartitionCount:1
ReplicationFactor:1         Configs:
        Topic: test         Partition: 0        Leader: 0
        Replicas: 0         Isr: 0
```

So there is no surprise there—the original topic has no replicas and is on server 0, the only server in our cluster when we created it.

Let's publish a few messages to our new topic:

> **bin/kafka-console-producer.sh --broker-list localhost:9092 –topic my-replicated-topic**

Who is listening to this
May the force be with you

Now let's consume these messages:

> **bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my-replicated-topic**

Who is listening to this
May the force be with you

Now let's test out fault-tolerance. Broker 1 was acting as the leader so let's kill it:

> **ps | grep server-1.properties**
7564 ttys002 0:15.91
> **kill -9 7564**

Leadership has switched to one of the slaves and node 1 is no longer in the in-sync replica set:

> **bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic**

Topic:my-replicated-topic PartitionCount:1
ReplicationFactor:3 Configs:
Topic: my-replicated-topic Partition: 0
Leader: 2 Replicas: 1,2,0 Isr: 2,0

But the messages are still be available for consumption even though the leader that took the writes originally is down:

> **bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my-replicated-topic**

Who is listening to this

May the force be with you

HOW TO PUT DATA INTO KAFKA

- **Producer API** in the org.apache.kafka.clients package

This client is production tested and generally both fast and fully featured. You can use this client by adding a dependency on the client jar using the following example maven co-ordinates (you can change the version numbers with new releases):

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>0.9.0.0</version>
</dependency>
```

- **Kafka Connect** in the org.apache.kafka.connect

Kafka Connect is a tool included with Kafka that imports and exports data to Kafka. It is an extensible tool that runs *connectors*, which implement the custom logic for interacting with an external system. Kafka Connect allows developers to write custom producer - SOURCE procedures in Java and run it with Kafka Connect. How it is done is more deeply explained in the example application.

HOW TO GET DATA FROM KAFKA

- **Consumer API** in org.apache.kafka.clients

  This client enables developers to write personalized consumers in Java, that connects to Kafka cluster, listens to a topic and retrieves messages. However, both Producer & Consumer API's seems to be more complicated and confusing than Kafka Connect which is the better version of importing/exporting data.

- **Kafka Clients**

  There are various numbers of Kafka Clients for consuming data with different programming languages written as open-source projects by others. Kafka project lists them on their clients web page (https://cwiki.apache.org/confluence/display/KAFKA/Clients). Some of them include:

  Python

  .NET

  Node.js

  HTTP REST, etc.

  Among them, Node.js seems to be interesting since Node.js is designed to be a data-driven web framework and would be nice to integrate Kafka as a communication middleware between a Node.js applications.

- **Kafka Connect**

  Kafka Connect is a tool included with Kafka that imports and exports data to Kafka. It is an extensible tool that runs connectors, which implement the custom logic for interacting with an external system. Kafka Connect allows developers to write custom consumer - SINK procedures in Java and

run it with Kafka Connect. How it is done is more deeply explained in the example application.

DESIGN

- **Persistency**

Kafka stores and caches its messages mostly in the filesystem. Instead of the common sense that is disk operations are very slow compared to memory operations – which is actually not false – it is still depends on how those operations are done and how the disk structure is designed. Random accesses are indeed very slow. On the other hand, throughput of hard drives has been diverging from the latency of a disk seek for the last decade.

As a result, the performance of linear writes on a JBOD configuration with six 7200rpm SATA RAID-5 array is about 600MB/sec but the performance of random writes is only about 100k/sec—a difference of over 6000X. These linear reads and writes are the most predictable of all usage patterns which are heavily optimized by the operating system. A modern operating system provides read-ahead and write-behind techniques that prefetching data in large block multiples and group smaller logical writes into large physical writes.

This suggests a design which is very simple: rather than maintain as much as possible in-memory and flush it all out to the filesystem in a panic when a program runs out of space, Kafka inverts that. All data is immediately written to a persistent log on the filesystem without necessarily flushing to disk. In effect this just means that it is transferred into the kernel's page cache.

Mostly, a BTree or other general-purpose access data structures are being used as the persistent data structure in the messaging systems. This structure has the advantage of all the operations are being O(1) and I/O operations do not block each other.

- **Efficiency**

Since one of the primary use cases of Kafka is to handle web activity data such as click sourcing – which are in huge amounts –, each page view may produce dozens of writes. By being very fast, Kafka helps ensure that it does not tip-over under load before the application. This is a pretty important property since one of the Kafka's aims is to become a centralized hub to serve to dozens or hundreds of applications on a centralized cluster as changes in usage patterns are close to daily-occurrence.

In general, other than disk access cost, there are two common causes of inefficiency in this type of system, that are too many small I/O operations and excessive byte copying.

To avoid lots of small I/O operations, Kafka abstracts group messages around a message set abstraction. Hence allowing network requests to group messages together and amortize the overhead of the network roundtrip. Even with this little optimization, results in a speed up of the whole system an undeniable amount.

Moreover, most of the data transferred from page cache of the OS which serves very fast and eliminates the cost of copying data to memory first. Secondly, transfer of data from page cache to sockets, which are then published to the outside world, done by sendfile system call which is a highly optimized code path for transferring data to sockets.

Taking all of those into consideration, Kafka does a lot of improvements under the hood to become one of the messaging systems available on the market.

- **Log Compaction**

Log compaction ensures that Kafka will always retain at least the last known value for each message key within the log of data for a single topic partition. It addresses use cases and scenarios such as restoring state after application crashes or system failure, or reloading caches after application restarts during operational maintenance.

Log compaction gives us a more granular retention mechanism so that we are guaranteed to retain at least the last update for each primary key. By doing this we guarantee that the log contains a full snapshot of the final value for every key not just keys that changed recently. This means downstream consumers can restore their own state off this topic without us having to retain a complete log of all changes.

Log compaction is a mechanism to give finer-grained per-record retention, rather than the coarser-grained time-based retention. The idea is to selectively remove records where we have a more recent update with the same primary key. This way the log is guaranteed to have at least the last state for each key.

This retention policy can be set per-topic, so a single cluster can have some topics where retention is enforced by size or time and other topics where retention is enforced by compaction.

,

APPLICATION – Converting a traditional RDBMS

to Stream of Changes Using Kafka

In order to demonstrate the use of Apache Kafka, we have took into account an OLTP system, specifically the one given as an assignment in the Data Warehouses course – ING Bank. This database contains Customer and Account information that are versioned when loaded into the Data Warehouse. Traditionally, snapshot of a database is captured periodically and given to the ETL procedure. Customer and Account changes are captured through a 'lastupdated' field which represents when the last change happened to a specific entry. What is missing in this scenario is, only the latest update during the snapshot period is taken into consideration. Moreover, for every snapshot, all of the database's content is copied and then changed entries are extracted from it.

Instead, we have developed a Kafka Connect Source Task that periodically – which can be very small – checks for changed entries and puts them into apache Kafka as a message for consumption. By doing so, we guarantee that only the changed entries are processed while loading to database with more granular change history. This Kafka Connect Source Task run by Kafka Connect with specific configurations results in the ease of all the ETL process.

MORE ON KAFKA CONNECT

Kafka Connect is a tool for scalably and reliably streaming data between Apache Kafka and other systems. It makes it simple to quickly define *connectors* that move large collections of data into and out of Kafka. Kafka Connect can ingest entire databases or collect metrics from all your application servers into Kafka topics, making the data available for stream processing with low latency. An export job can deliver data from Kafka topics into secondary storage and query systems or into batch systems for offline analysis. Kafka Connect features include:

- **A common framework for Kafka connectors** - Kafka Connect standardizes integration of other data systems with Kafka, simplifying connector development, deployment, and management
- **Distributed and standalone modes** - scale up to a large, centrally managed service supporting an entire organization or scale down to development, testing, and small production deployments
- **REST interface** - submit and manage connectors to your Kafka Connect cluster via an easy to use REST API
- **Automatic offset management** - with just a little information from connectors, Kafka Connect can manage the offset commit process automatically so connector developers do not need to worry about this error prone part of connector development
- **Distributed and scalable by default** - Kafka Connect builds on the existing
- **Streaming/batch integration** - leveraging Kafka's existing capabilities, Kafka Connect is an ideal solution for bridging streaming and batch data systems

Running Kafka Connect

Kafka Connect currently supports two modes of execution: standalone (single process) and distributed. In standalone mode all work is performed in a single process. This configuration is simpler to setup and get started with and may be useful in situations where only one worker makes sense (e.g. collecting log files), but it does not benefit from some of the features of Kafka Connect such as fault tolerance. You can start a standalone process with the following command:

> bin/connect-standalone.sh config/connect-standalone.properties connector1.properties [connector2.properties ...]

The first parameter is the configuration for the worker. This includes settings such as the Kafka connection parameters, serialization format, and how frequently to commit offsets. The provided example should work well with a local cluster running with the default configuration provided by config/server.properties. It will require tweaking to use with a different configuration or production deployment. The remaining parameters are connector

configuration files. You may include as many as you want, but all will execute within the same process (on different threads).

Connectors And Tasks

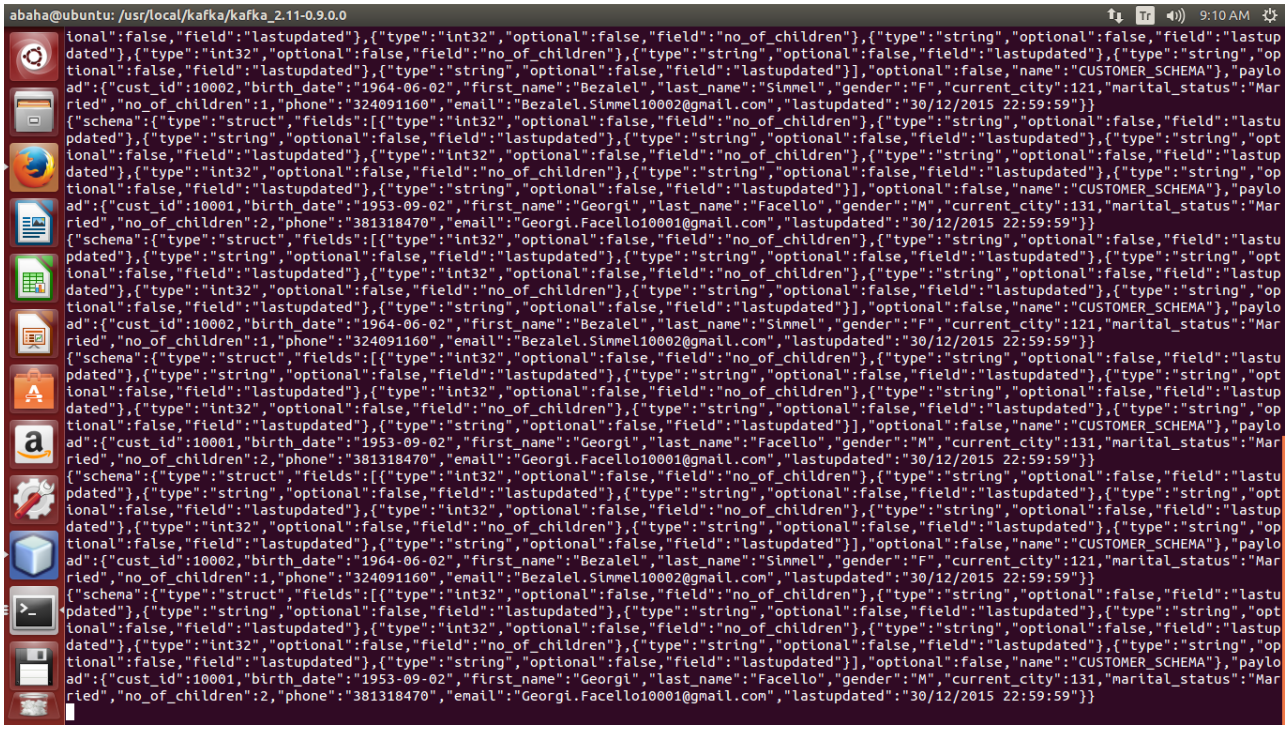
To copy data between Kafka and another system, users create a Connector for the system they want to pull data from or push data to. Connectors come in two flavors: SourceConnectors import data from another system (e.g. JDBCSourceConnector would import a relational database into Kafka) and SinkConnectors export data (e.g. HDFSSinkConnector would export the contents of a Kafka topic to an HDFS file). Connectors do not perform any data copying themselves: their configuration describes the data to be copied, and the Connector is responsible for breaking that job into a set of Tasks that can be distributed to workers.

These Tasks also come in two corresponding flavors: SourceTaskand SinkTask. With an assignment in hand, each Task must copy its subset of the data to or from Kafka. In Kafka Connect, it should always be possible to frame these assignments as a set of input and output streams consisting of records with consistent schemas. Sometimes this mapping is obvious: each file in a set of log files can be considered a stream with each parsed line forming a record using the same schema and offsets stored as byte offsets in the file. In other cases it may require more effort to map to this model: a JDBC connector can map each table to a stream, but the offset is less clear. One possible mapping uses a timestamp column to generate queries incrementally returning new data, and the last queried timestamp can be used as the offset.

Application Development & Running

While developing the application, we have only turned Customer relation into stream for testing purposes. For this task, two java classes have been implemented namely, ‘SourceJDBCConnectorCustomer.java’ & ‘SourceJDBCTaskCustomer.java’ that implements ‘SourceConnector’ and ‘SourceTask’ interfaces of Kafka Connect package. As the nature of SourceConnector interface, ‘SourceJDBCConnectorCustomer.java’ class have the corresponding task configurations and deals with task partitioning for multiple threads. In our application, only one SourceTask thread will be run. In the ‘SourceJDBCTaskCustomer.java’, poll() function continuously checks for customer changes

by checking the ‘lastupdated’ column with check interval. For every run of poll, check interval is incremented as desired. Then, all the changes in the customer relation are put into Kafka topic ‘customer-pipe’. One thing to note here is that, while putting data into Kafka through custom Kafka Connect Source Connector and Source Task, there are only few possible schema options such as Boolean, String, Integer. In order to ease consumption of the messages – database events, customer changes in this case – custom Schema corresponding to the Customer Relation Schema has been generated. Currently, every message in this topic is type of this Schema. Below, is the screenshot of Kafka consumer client, listening to the topic ‘customer-pipe’.



In order to run custom Kafka Connect Source Connector and Source Task, project should be built and its ‘.jar’ file should be put into to Kafka’s libs folder with the dependency folders included. Moreover, in order to run the Connector, Configuration file is placed into the Kafka’s ‘config’ folder and specified while running the Kafka Connect through terminal.

Configuration file and its content are as follows:

- **name**=etl-source-branch --- Name of the configuration class
- **connector.class**=kafkaconnectetl.SourceJDBCConnectorCustomer --- Name of the Source Connector Class
- **tasks.max**=1 --- Maximum number of task threads that should be run in parallel
- **driver**=org.postgresql.Driver --- Name of the driver
- **db**=jdbc:postgresql://localhost/ING_OLTP --- Database Adress
- **dbuser**=postgres --- Database User Name
- **dbpassword**=postgrespassword --- Database Password
- **topic**=customer-pipe --- Which topic to publish

Among those configurations, **name**, **connector.class**, **tasks.max** and **topic** are necessary. Other configurations are for ease of readability of implementation and used internally by Source Connector and Source Task classes, not by Kafka.

By doing such, we have just put Customer table changes into a stream in Kafka topic. These messages can be easily used by any custom Kafka Connect classes that implements Sink Connector and Sink Task. Moreover, those can be put into different targets such as HDFS, OLAP system, Key-Value storage, etc. Moreover, since the Schema that is used by Source Task class can be modified to hold some extra values, it is possible to pre calculate some ETL procedures before putting them into Kafka or any scenario can be implemented internally. Source code for 'SourceJDBCConnectorCustomer.java' and 'SourceJDBCTaskCustomer.java' are included with the report.

CONCLUSION

Stream processing is required when data has to be processed fast and / or continuously, i.e. reactions have to be computed and initiated in real time. This requirement is coming more and more into every vertical. Many different frameworks and products are available on the market already, however the number of mature solutions with good tools and commercial support is small today. Apache Storm is a good, open source framework; however custom coding is required due to a lack of development tools and there's no commercial support right now.

Products such as IBM InfoSphere Streams or TIBCO StreamBase offer complete products, which close this gap. You definitely have to try out the different products, as the websites do not show you how they differ regarding ease of use, rapid development and debugging, and real-time streaming analytics and monitoring. Stream processing complements other technologies such as a DWH and Hadoop in a big data architecture - this is not an "either/or" question.

Stream processing has a great future and will become very important for most companies. Big Data and Internet of Things are huge drivers of change.