Data Streaming with Apache Kafka & MongoDB

September 2016

mongoDB

Table of Contents

Introduction

In today's data landscape, no single system can provide all of the required perspectives to deliver real insight. Deriving the full meaning from data requires mixing huge volumes of information from many sources.

At the same time, we're impatient to get answers instantly; if the time to insight exceeds 10s of milliseconds then the value is lost – applications such as high frequency trading, fraud detection, and recommendation engines can't afford to wait. This often means analyzing the inflow of data before it even makes it to the database of record. Add in zero tolerance for data loss and the challenge gets even more daunting.

As the number, variety, and velocity of data sources grow, new architectures and technologies are needed.

Apache Kafka and data streams are focused on ingesting the massive flow of data from multiple fire-hoses and then routing it to the systems that need it – filtering, aggregating, and analyzing en-route.

Enterprise messaging systems are far from new, but even frameworks from the last 10 years (such as ActiveMQ and RabbitMQ) are not always up to the job of managing modern data ingestion and routing pipelines. A new generation of technologies is needed to consume and exploit today's data sources. This paper digs into these technologies (Kafka in particular) and how they're used. The paper also examines where MongoDB fits into the data streaming landscape and includes a deep dive into how it integrates with Kafka.

Apache Kafka

How Kafka Works & What it Provides

Kafka provides a flexible, scalable, and reliable method to distribute streams of event data from one or more **producers** to one or more **consumers**. Examples of **events** (or **messages**) include:

- A periodic sensor reading such as the current temperature
- A user adding an item to the shopping cart in an online store
- A Tweet being sent with a specific hashtag

- A log entry generated for each click in a web application

Streams of Kafka events are organized into **topics**. A producer chooses a topic to send a given event to and consumers select which topics they pull events from. For example, a financial application could pull NYSE stock trades from one topic, and company financial announcements from another in order to look for trading opportunities.

The consumers typically receive streamed events in near real-time, but Kafka actually persists data internally, allowing consumers to disconnect for hours and then catch up once they're back online. In fact, an individual consumer can request the full stream of events from the first event stored on the broker. Administrators can choose to keep the full history for a topic, or can configure an appropriate deletion policy. With this system, it's possible for different consumers to be processing different chunks of the sequence of events at any given time. Each event in a topic is assigned an offset to identify its position within the stream. This offset is unique to the event and never changes.

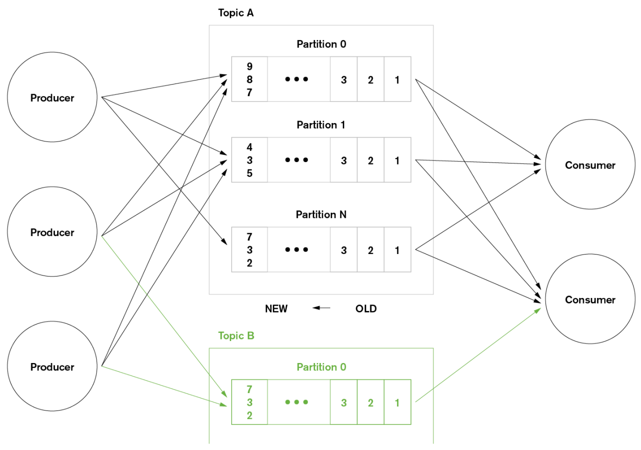

Figure 1: Kafka Topics & Partitions

The ability to reliably replay events that have already been received, in the exact order in which they were received, provides a number of benefits such as:

- Newly added consumers can catch up on everything that's happened
- When the code for an application is updated, the application can process the full stream of events again, applying the new logic to each one

In Kafka, topics are further divided into **partitions** to support scale out. As each message is produced, the producer determines the correct partition for a message (depending on its topic and message key), and sends it to an appropriate **broker** for that partition. In this way, the processing and storage for a topic can be linearly scaled across many brokers. Similarly, an application may scale out by using many consumers for a given topic, with each pulling events from a discrete set of partitions.

A consumer receives events from a topic's partition in the order that they were added by the provider, but the *order is not guaranteed between different partitions*. While this appears troublesome, it can be mitigated by controlling how events are assigned to partitions. By default, the mapping of events to partitions is random but a **partitioning key** can be defined such that 'related' events are placed in the same partitions. In our financial application example, the stock symbol could be used as the partition key so that all events for the same company are written to the same partition – if the application then has just one consumer pulling from that partition then you have a guarantee that all trades for that stock are processed in the correct order.

Multiple consumers can be combined to form a **consumer group**. Each consumer within the group processes a subset of events from a specific topic – at any instant, messages from one partition can only be received by one consumer within the group. Different applications (e.g., fraud detection and product recommendation) can be represented by their own consumer groups.

High Availability can be implemented using multiple Kafka brokers for each topic partition. For each partition there is a single broker that acts as the **leader** in addition to one or more **followers**. The leader handles all reads and writes for the topic partition and the followers each **replicate** from the leader. Should the leader fail, one of the followers is automatically promoted to be the new leader. Typically, each broker acts as the leader for some partitions and as a follower for others. This replication approach prevents the loss of data when a broker fails and increases Kafka's availability,

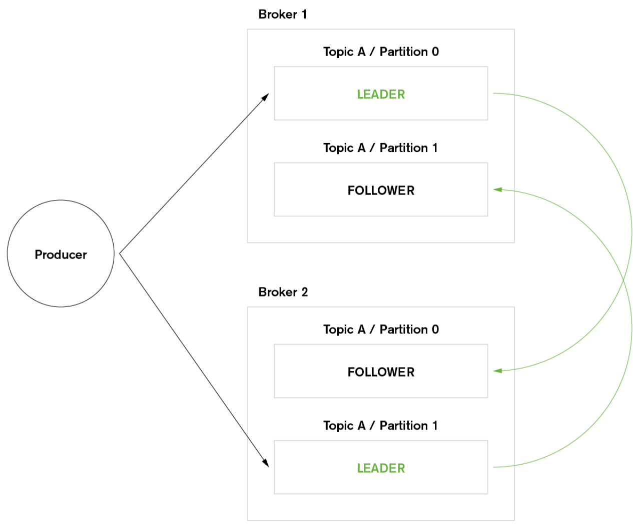

Figure 2: Kafka Replication

An event is considered **committed** when all replicas for that topic partition have applied it. A producer has the option of blocking until the write has been committed or continuing right away. Consumers only ever read committed events.

Each Kafka broker stores all of the data for its topic partitions on disk in order to provide persistence. Because the data is immutable and append-only, each broker is able to handle a large number of writes, and the cost of reading the most recent message remains constant as the volume of data stored grows. In spite of Kafka's ability to work efficiently with large data volumes, it is often desirable to remove old or obsolete data; Kafka users can choose between two two different algorithms for managing space: retention policies and log compaction:

- **Retention policy**: You can choose, on a per-topic basis, to delete log files after they reach a certain age, or the number of bytes in the topic exceeds a certain size
- **Log compaction**: Log Compaction is an optional Kafka feature that reduces the storage size for keyed data (e.g., change logs for data stored in a database). Log compaction allows you to retain only the most recent message with a given key, and delete older messages with the same key. This can be useful for operations like database updates, where you only care about the most recent message.

Kafka Use Cases

Log Aggregation: This has traditionally involved the collection of physical log files from multiple servers so that they can be stored in a central location – for example, HDFS. Analyzing the data would then be performed by a periodic batch job. More modern architectures use Kafka to combine real-time log feeds from multiple sources so that the data can be constantly monitored and analyzed – reducing the interval between an event being logged and its consequences being understood and acted upon.

Website Activity Tracking: This was the original use case for Kafka. Site activity such as pages visited or adverts rendered are captured into Kafka topics – one topic per data type. Those topics can then be consumed by multiple functions such as monitoring, real-time analysis, or archiving for offline analysis. Insights from the data are then stored in an operational database such as MongoDB where they can be analyzed alongside data from other sources.

Event Sourcing: Rather than maintaining and storing the latest application state, event sourcing relies on storing all of the *changes* to the state (e.g., `[x=150, x++, x+=12, x-=2]`) in the original order so that they can be replayed to recreate the final state. This pattern is often used in financial applications. Kafka is well suited to this design approach as it can store arbitrarily long sequences of events, and quickly and efficiently provide them in the correct sequence to an application.

Microservices: Microservice architectures break up services into small, discrete functions (typically isolated within containers) which can only communicate with each other through well defined, network-based APIs. Examples include eCommerce applications where the service behind the 'Buy Now' button must communicate with the inventory service. A typical application contains a large number of microservices and containers. Kafka provides the means for containers to pass messages to each other – multiple containers publishing and subscribing to the same topics so that each container has the data it needs. Since Kafka persists the sequences of messages, when a container is rescheduled it is able to catch up on everything it has missed; when a new container is added (e.g., to scale out) it can bootstrap itself by requesting prior event data.

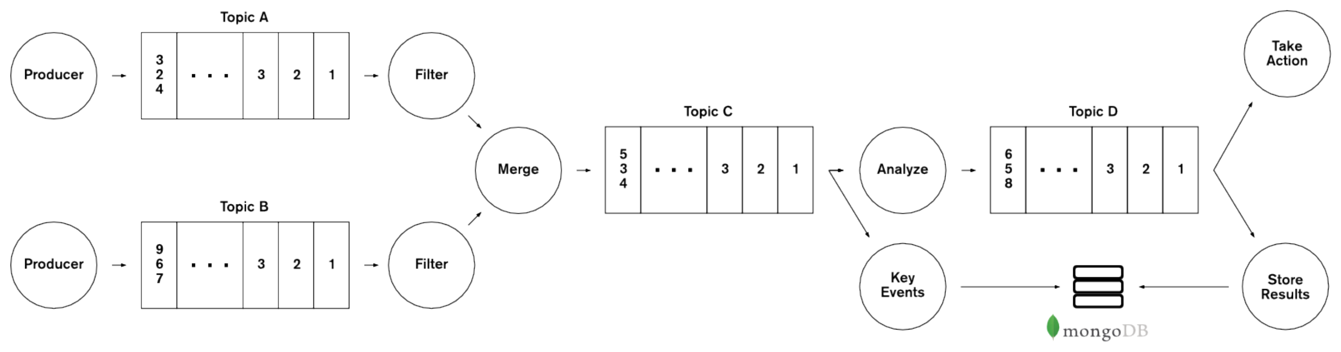

Figure 3: Stream Processing with Kafka & MongoDB

More information on Microservices can be found in the Microservices: The Evolution of Building Modern Applications white paper as well as in Enabling Microservices: Containers & Orchestration Explained.

Stream Processing: stream processing involves filtering, manipulating, triggering actions, and deriving insights from the data stream as it passes through a series of functions. Kafka passes the event messages between the processing functions, merging and forking the data as required. Technologies such as Apache Storm, Samza, Spark Streaming, Apache Flink, and Kafka Streams are used to process events as they pass through, while interesting events and results are written to a database like MongoDB where they're used for analysis and operational decisions. This is the pattern used for the Indian smart housing project described later – where MongoDB stores aggregated energy sensor data which is used for billing and energy performance benchmarking between properties.

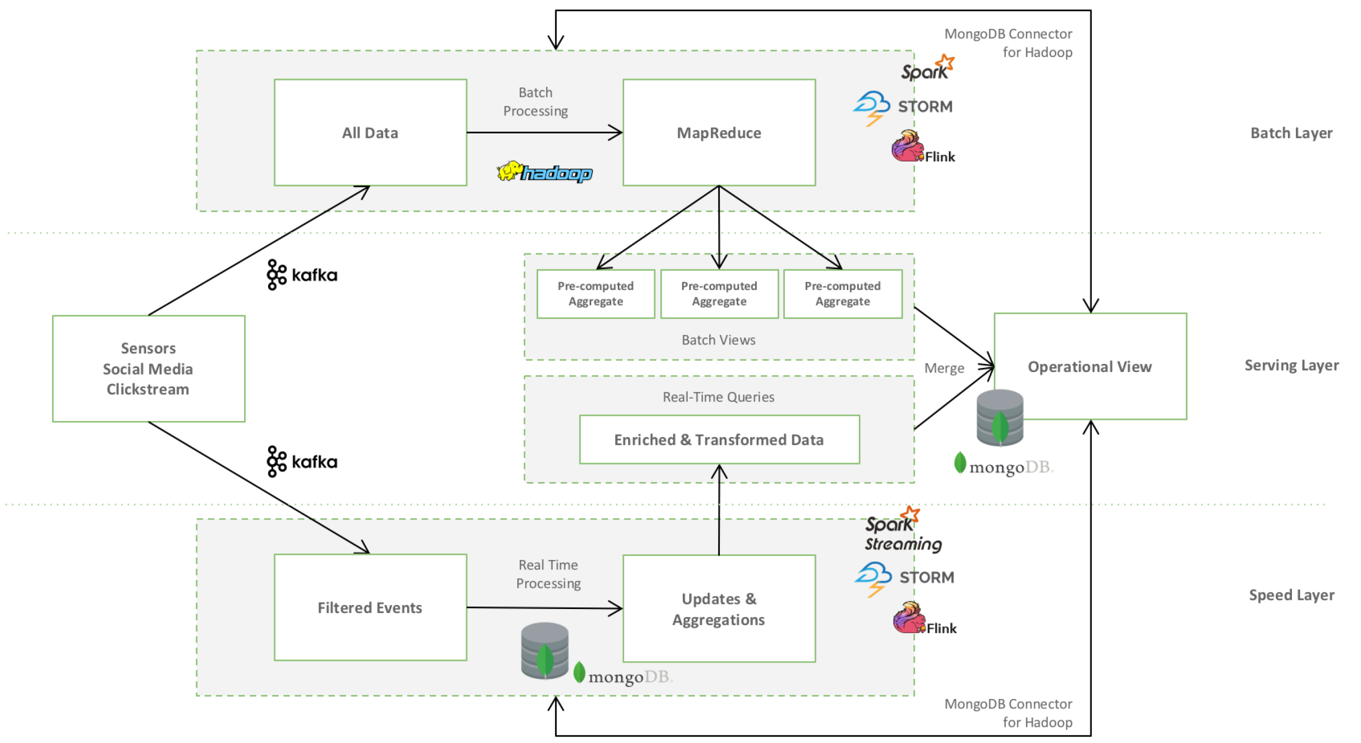

Figure 4: Lambda Architecture

Lambda Architecture: Applications following the Lambda Architecture (Figure 4) augment data produced by stream processing on recent events (the *Speed Layer*) with views built from batch processing jobs run on the full, historical data set (the *Batch Layer*).

The Lambda Architecture is coming under increasing scrutiny due to the operational complexity of managing two distributed systems which must implement the same logic. A more contemporary, scalable, and less complex solution is described below in *Operationalizing the Data Lake with MongoDB*.

Internet of Things (IoT): IoT applications must cope with massive numbers of events being generated by a multitude of devices. Kafka plays a vital role in providing the *fan-in* and real-time collection of all of that sensor data. A common use case is telematics, where diagnostics from a vehicle's sensors must be received and processed back at base.

Once captured in Kafka topics, the data can be processed in multiple ways, including stream processing or Lambda architectures. It is also likely to be stored in an operational database such as MongoDB, where it can be combined with other stored data to perform real-time analytics and support operational applications such as triggering personalized offers.

Implementation Recommendations for Kafka

Great value can be derived from combining multiple sources of data – some combinations might be obvious when setting up the Kafka architecture, but others only come to mind later when an application developer or analyst sees everything that's available. For this reason, it's recommended to design on the side of openness and extensibility:

- Build a minimal number of Kafka clusters.
- Make your Kafka topics accessible to all.
- Use common data formats between topics to reduce barriers in parsing and combining multiple data sources.

Some data may still need to be kept in silos due to security concerns, but it's worth asking the question about which topics can be made available to all internal users. Kafka 0.9 supports secure authentication and per-topic ACLs, which allow administrators to control access to individual topics.

JSON is a suitable common data format for standardization, and it's also worth considering **Apache Avro**, which allows schemas to be enforced on any given topic. Avro also has the advantage that it's more compact than JSON. It's very simple to map between JSON and Avro, which helps when integrating with MongoDB.

Each topic should use a common schema, but it's also important to recognize that schema evolve over time; consider including a schema version identifier in each message.

Kafka is extremely good at ingesting and storing massive numbers of events in real-time; this makes it a great buffer to smooth out bursty traffic. An example could be the results produced at the end of a customer classification Hadoop batch job, which must then be propagated into personalization and marketing applications.

As in any distributed system, poor network performance is the enemy of low latency and high throughput. Where possible, keep processing local and replicate data between Kafka clusters in different data centers.

Kafka Limitations

By default, the replication of messages from the leader to the followers is asynchronous, which can result in lost messages. In many applications, that is a worthwhile trade off (e.g., for log aggregation) as the performance benefits outweigh the potential loss of a small number of events. When greater safety is required, a topic can be configured to require 1 or more followers to acknowledge receipt of the event before it's committed.

Writes are immutable – once an event has been written to a Kafka topic, it cannot be removed or changed, and at some point, it will be received and acted upon by all subscribed consumers. In many cases, it may be possible to undo the downstream effects of an event by creating a new, compensating event – e.g. if you want to undo an event to reduce a balance by $100 then a second event can be written that increases the same balance by $100. There are other scenarios where it's harder to roll back the clock; consider a rogue temperature sensor event from a data center that triggers cutting power and a halon dump.

Kafka is designed for high performance writes and sequential reads but if random access is a requirement then the data should be be persisted and read from a database such as MongoDB.

Alternate Messaging Systems

The idea of moving data between systems is nothing new and frameworks have been available for decades. Two of the most popular frameworks are described here.

RabbitMQ: Implements the Advanced Message Queuing Protocol (AMQP) to act as a message broker. RabbitMQ focuses on the delivery of messages to consumers with complex routing and per-message delivery guarantees. Kafka is more focused on ingesting and persisting massive streams of updates and ensuring that they're delivered to consumers in the correct sequence (for a given partition).

Apache ActiveMQ: A Java based message broker which include a Java Message Service (JMS) client. As with RabbitMQ, the advantage offered by Kafka is the ingesting and persisting of massive streams of data.

Related Technologies

A number of technologies complement Kafka; some of the more notable ones are described here.

Kafka Connect: A standard framework to reliably stream data between Kafka and other systems. It makes it simple to implement **connectors** that move large sets of data into and out of Kafka. Kafka Connect can run in either standalone or distributed modes – providing scalability.

Apache Zookeeper: Provides distributed configuration, synchronization, and naming for large distributed systems. Kafka uses Zookeeper to share configuration information across a cluster.

Apache Storm: Provides batch, distributed processing of streaming data. Storm is often used to implement Stream Processing, where it can act as both a Kafka consumer of raw data and producer of derived results.

Apache Heron: The successor to Apache Storm, produced by Twitter after their Storm deployment ran into issues when scaled to thousands of nodes. It is more resource efficient and maintains API compatibility with Storm, while delivering an order of magnitude greater performance.

Apache Spark: Performs data analytics using a cluster of nodes. Spark is used for many batch workloads that would previously have been run with Apache Hadoop – Spark's ability to store the data in memory rather than on disk results in much faster execution. More information can be found in Apache Spark and MongoDB – Turning Analytics into Real-Time Action.

Apache Spark Streaming: Adds the ability to perform streaming analytics, reusing the same application analytics code used in Spark batch jobs. As with Storm it can act as both a Kafka consumer of raw data and producer of derived results – it may be preferred if you already have Spark batch code that can be reused with Spark Streaming.

Kafka Streams: A new library included with Apache Kafka that allows you to build a modern stream processing system with your existing Kafka Cluster. Kafka Streams makes it easy to join, filter, or aggregate data from streams using a high level API.

Apache Apex: Enables batch and streaming analytics on Hadoop – allowing the same application Java code to be used for both. Through Apache Apex **Malhar**, reference data can be pulled in from other sources and results can be pushed to those same data stores – including MongoDB. Malhar is also able to act as a Kafka consumer, allowing Apex to analyze streams of topic data.

Apache Flume: A distributed, highly available service for aggregating large volumes of log data, based on streaming data flows. Kafka is more general purpose and has a couple of advantages over Flume:

- Kafka consumers pull data and so cannot be overwhelmed by the data stream.

- Kafka includes replication so events are not lost if a single cluster node is lost.

Apache Flink: A framework for distributed big data analytics using a distributed data flow engine. Any data storage is external to Flink; e.g., in MongoDB or HDFS. Flink often consumes its incoming data from Kafka.

Apache Avro: A data serialization system, very often used for event messages in Kafka. The schema used is stored with the messages and so serialization and deserialization is straight-forward and efficient. Avro schemas are defined in JSON, making it simple to use with languages, libraries, and databases such as MongoDB designed to work with JSON.

A Diversion – Graphically Build Data Streams Using Node-RED

Node-RED provides a simple, graphical way to build data pipelines – referred to as **flows**. There are community provided **nodes** which act as connectors for various devices and APIs. At the time of writing, there are almost 500 nodes available covering everything from Nest thermostats and Arduino boards, to Slack and Google Hangouts, to MongoDB. Flows are built by linking together nodes and by adding custom JavaScript code. Node-RED is a perfect tool to get small IoT projects up and running quickly or to run at the edge of the network to collect sensor data.

Installing Node-Red together with the node for MongoDB and then starting the server is as simple as:

```
sudo npm install -g node-red
sudo npm install -g node-red-node-mongodb
node-red
```

Once running, everything else is setup by browsing `http://127.0.0.1:1880/`. Building a new flow is intuitive:

- Drag nodes onto the canvas.
- Double click to configure a node.
- Drag lines between nodes to connect them.
- Add snippets of JavaScript to manipulate the data as it passes through a function node.

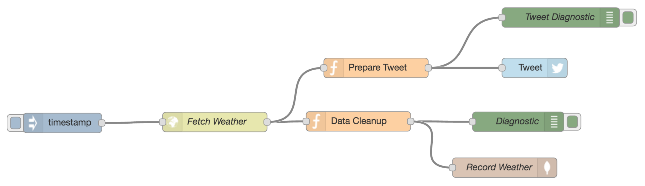

Figure 5: Example Node-RED Flow

The flow shown in Figure 5 implements a weather publishing system:

- The *timestamp* node triggers the flow every five minutes.
- The *Fetch Weather* node makes a web API call to retrieve the current weather and forecast data for our location; that data is then used in two ways:
  - The summary of the current weather is extracted by the *Prepare Tweet* JavaScript node and then the "Tweet" node Tweets out that summary.
  - The *Data Cleanup* JavaScript node extracts the current weather and reformats the timestamp before sending it to the *Record Weather* node which writes it to a MongoDB collection.

The Javascript for *Prepare Tweet* is:

```
var weather = JSON.parse(msg.payload);
msg = {};
payload=weather.currently.summary;
msg.payload=payload;
return msg;
```

and for *Data Cleanup*:

```
var weather = JSON.parse(msg.payload);
msg = {};
payload = {};
payload.weatherNow=weather.currently;
payload.weatherNow.time=new Date(
        weather.currently.time * 1000);
msg.payload=payload;
return msg;
```

After executing the flow, the Tweet gets sent and the data is stored in MongoDB:

```
db.weather.findOne()
{
    "_id" : ObjectId("571e2e9865177b7a1f40ddb5"),
    "weatherNow" : {
        "time" : ISODate("2016-04-25T14:49:58Z"),
        "summary" : "Mostly Cloudy",
        "icon" : "partly-cloudy-day",
        "nearestStormDistance" : 0,
        "precipIntensity" : 0.0033,
        "precipIntensityError" : 0.0021,
        "precipProbability" : 0.21,
        "precipType" : "rain",
        "temperature" : 51.51,
        "apparentTemperature" : 51.51,
        "dewPoint" : 39.57,
        "humidity" : 0.64,
        "windSpeed" : 13.41,
        "windBearing" : 314,
        "visibility" : 10,
        "cloudCover" : 0.7,
        "pressure" : 1007.32,
        "ozone" : 387.18
    }
}
```

To recreate this flow, simply import this definition.

Operationalizing the Data Lake with MongoDB

The traditional Enterprise Data Warehouse (EDW) is straining under the load, overwhelmed by the sheer volume and variety of data pouring into the business, and being able to store it in a cost-efficient way. As a result many organizations have turned to Hadoop as a centralized repository for this new data, creating what many call a data lake.

Figure 6 presents a design pattern for integrating MongoDB with a data lake:

- Data streams are ingested into one or more Kafka topics, which route all raw data into HDFS. Processed events that drive real-time actions, such as personalizing an offer to a user browsing a product page, or alarms for vehicle telemetry, are routed to MongoDB for immediate consumption by operational applications.
- Distributed processing libraries such as Spark or MapReduce jobs materialize batch views from the raw data stored in the Hadoop data lake.
- MongoDB exposes these models to the operational processes, serving queries and updates against them with real-time responsiveness.
- The distributed processing libraries can re-compute analytics models, against data stored in either HDFS or MongoDB, continuously flowing updates from the operational database to analytics views.

The details and motivation for this solution may be found in the white paper Unlocking Operational Intelligence from the Data Lake.

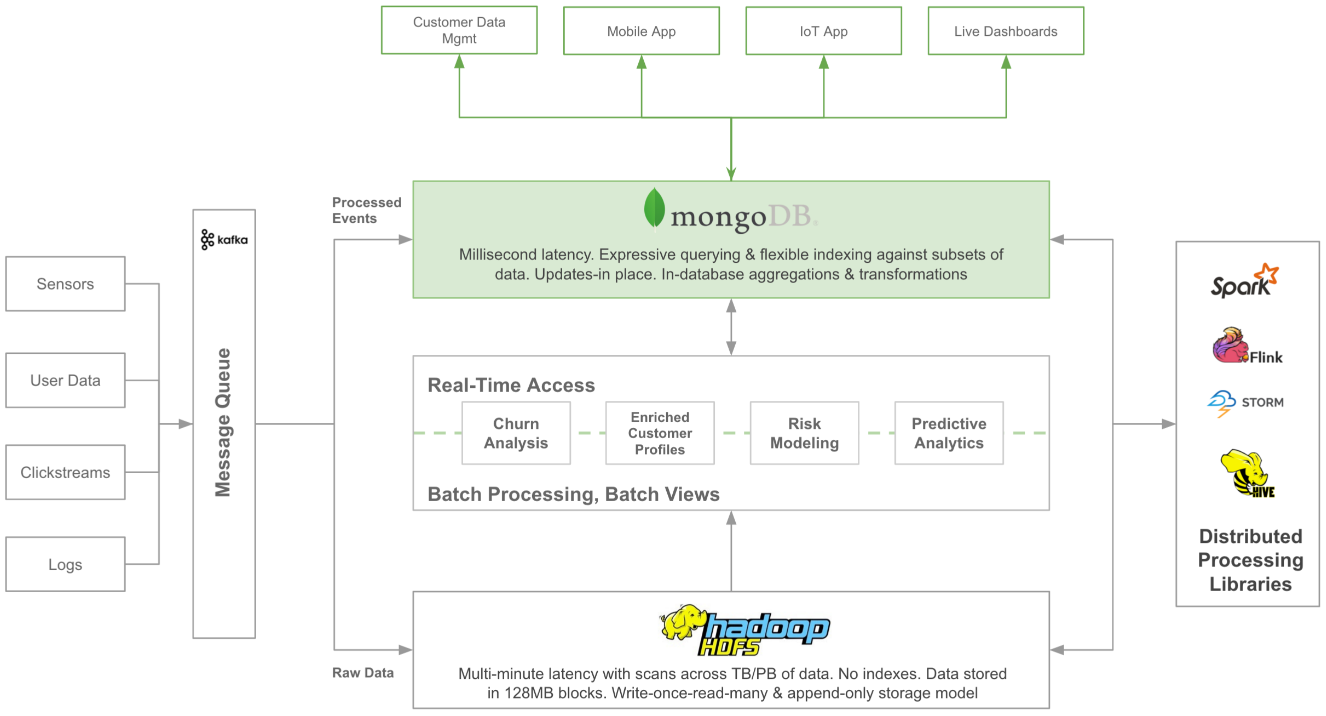
Customer Data Mgmt
Mobile App
IoT App
Live Dashboards
Processed Events
mongoDB
Millisecond latency. Expressive querying & flexible indexing against subsets of data. Updates-in place. In-database aggregations & transformations
kafka
Message Queue
Sensors
User Data
Clickstreams
Logs
Real-Time Access
Churn Analysis
Enriched Customer Profiles
Risk Modeling
Predictive Analytics
Batch Processing, Batch Views
Raw Data
hadoop HDFS
Multi-minute latency with scans across TB/PB of data. No indexes. Data stored in 128MB blocks. Write-once-read-many & append-only storage model
Spark
Flink
STORM
HIVE
Distributed Processing Libraries

Figure 6: Design pattern for operationalizing the data lake

MongoDB As a Kafka Producer – Tailing the Oplog

When using any database as a producer, it's necessary to capture all database changes so that they can be written to Kafka. With MongoDB this can be achieved by monitoring its oplog.

The **oplog** (operations log) is a special capped collection that keeps a rolling record of all operations that modify the data stored in your database.

Tailable cursors, have many uses, such as real-time notifications of all the changes to your database. A tailable cursor is conceptually similar to the Unix `tail -f` command. Once you've reached the end of the result set, the cursor will not be closed, rather it will continue to wait forever for new data and when it arrives, return that too.

MongoDB replication is implemented using the oplog and tailable cursors; the primary records all write operations in its oplog. The secondary members then asynchronously fetch and then apply those operations. By using a tailable cursor on the oplog, an application receives all changes that are made to the database in near real-time.

A producer can be written to propagate all MongoDB writes to Kafka by tailing the oplog in the same way. The logic is more complex when using a sharded cluster:

- The oplog for each shard must be tailed.
- The MongoDB shard balancer occasionally moves documents from one shard to another; causing *deletes* to be written to the originating shard's oplog and *inserts* to that of the receiving shard. Those internal operations must be filtered out.

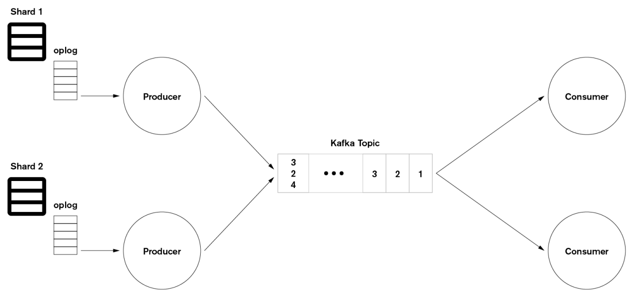

Figure 7: Kafka Producer Tailing the MongoDB oplog

This blog post details how to use tailable cursors on the oplogs in a sharded cluster.

MongoDB As a Kafka Consumer

In order to use MongoDB as a Kafka consumer, the received events must be converted into BSON documents before they are stored in the database. It's a simple and automatable process to convert the received JSON or Avro message payload into a Java object and apply any required business logic on that object before encoding it as a BSON document which is then written to MongoDB.

This blog post shows a worked Java example of a MongoDB Kafka consumer.

MongoDB and Kafka in the Real World

Man AHL: The Man Group is one of the largest hedge fund investors in the world. AHL is a subsidiary, focused on systems trading. The data for up to 150,000 ticks per second is received from multiple financial sources and written to Kafka. Kafka provides both the consolidation and buffering of the events before they are stored in MongoDB, where the data can be analyzed. Details on Man AHL's use case can be found in this presentation.

State: State is an intelligent opinion network; connecting people with similar beliefs who want to join forces and make waves. User and opinion data is written to MongoDB and then the oplog is tailed so that all changes are written to topics in Kafka where they are consumed by the user recommendation engine. Details on State's use of MongoDB and Kafka can be found in this presentation.

Josh Software: Part of a project near Mumbai, India that will house more than 100,000 people in affordable smart homes. Data from millions of sensors, is pushed to Kafka and then processed in Spark before the results are written to MongoDB. MongoDB Connector for Hadoop is used to connect the operational and analytical data sets. More details on this project and its use of Kafka, MongoDB and Spark can be found in this blog post.

Comparethemarket.com: One of the UK's leading price comparison providers, and one of the country's best known household brands. Comparethemarket.com uses MongoDB as the default operational database across its microservices architecture. Its online comparison systems need to collect customer details efficiently and then securely send them to a number of different providers. Once the insurers' systems respond, Comparethemarket.com can aggregate and display prices for consumers. At the same time, MongoDB generates real-time analytics to personalize the customer experience across the company's web and mobile properties.

As Comparethemarket.com transitioned to microservices, the data warehousing and analytics stack were also modernized. While each microservice uses its own MongoDB database, the company needs to maintain synchronization between services, so every application event is written to a Kafka topic. Event processing runs against the topic to identify relevant events that can then trigger specific actions – for example customizing customer questions, firing off emails, presenting new offers and more. Relevant events are written to MongoDB, enabling the user experience to be personalized in real time as customers interact with the service.

Learn more about these use cases together with more details on using Kafka with MongoDB by watching the Data Streaming with Apache Kafka & MongoDB webinar replay.

Future Directions for Kafka

Kafka 0.10released in May 2016 with these key features:

- Kafka Streams is a Java library for building distributed stream processing apps using Kafka; in other words enabling applications to transform input Kafka topics into output Kafka topics and/or invoke external services or write to a database. Kafka Streams is intended to distinguish itself from the more analytics-focused frameworks such as Spark, Storm, Flink, and Samza by targeting core application functionality.
- Performance enhancements for compressed streams.
- Rack-Aware replica assignment can be used to ensure that a leader and its followers are run in different racks, improving availability.

We Can Help

We are the MongoDB experts. Over 2,000 organizations rely on our commercial products, including startups and more than a third of the Fortune 100. We offer software and services to make your life easier:

MongoDB Enterprise Advanced is the best way to run MongoDB in your data center. It's a finely-tuned package of advanced software, support, certifications, and other services designed for the way you do business.

MongoDB Atlas is a database as a service for MongoDB, letting you focus on apps instead of ops. With MongoDB Atlas, you only pay for what you use with a convenient hourly billing model. With the click of a button, you can scale up and down when you need to, with no downtime, full security, and high performance.

MongoDB Cloud Manager is a cloud-based tool that helps you manage MongoDB on your own infrastructure. With automated provisioning, fine-grained monitoring, and continuous backups, you get a full management suite that reduces operational overhead, while maintaining full control over your databases.

MongoDB Professional helps you manage your deployment and keep it running smoothly. It includes support from MongoDB engineers, as well as access to MongoDB Cloud Manager.

Development Support helps you get up and running quickly. It gives you a complete package of software and services for the early stages of your project.

MongoDB Consulting packages get you to production faster, help you tune performance in production, help you scale, and free you up to focus on your next release.

MongoDB Training helps you become a MongoDB expert, from design to operating mission-critical systems at scale. Whether you're a developer, DBA, or architect, we can make you better at MongoDB.

MongoDB has partnered with [Confluent] (www.confluent.io), the company founded by the creators of Apache Kafka. If you have questions or need additional Enterprise functionality for Kafka, Confluent may be able to help.

Resources

For more information, please visit mongodb.com or contact us at sales@mongodb.com.

Case Studies (mongodb.com/customers)
Presentations (mongodb.com/presentations)
Free Online Training (university.mongodb.com)
Webinars and Events (mongodb.com/events)
Documentation (docs.mongodb.com)
MongoDB Enterprise Download (mongodb.com/download)
MongoDB Atlas database as a service for MongoDB (mongodb.com/cloud)

mongoDB

New York • Palo Alto • Washington, D.C. • London • Dublin • Barcelona • Sydney • Tel Aviv
US 866-237-8815 • INTL +1-650-440-4474 • info@mongodb.com