FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

STREAMING ARCHITECTURE

eMag Issue 57 - Jan 2018

PRESENTATION

Exploring the Fundamentals of Stream Processing with the Dataflow Model and Apache Beam

PRESENTATION

Migrating Batch ETL to Stream Processing: A Netflix Case Study with Kafka and Flink

IN THIS ISSUE

FOLLOW US


facebook.com /InfoQ


@InfoQ


google.com /+InfoQ


linkedin.com company/infoq

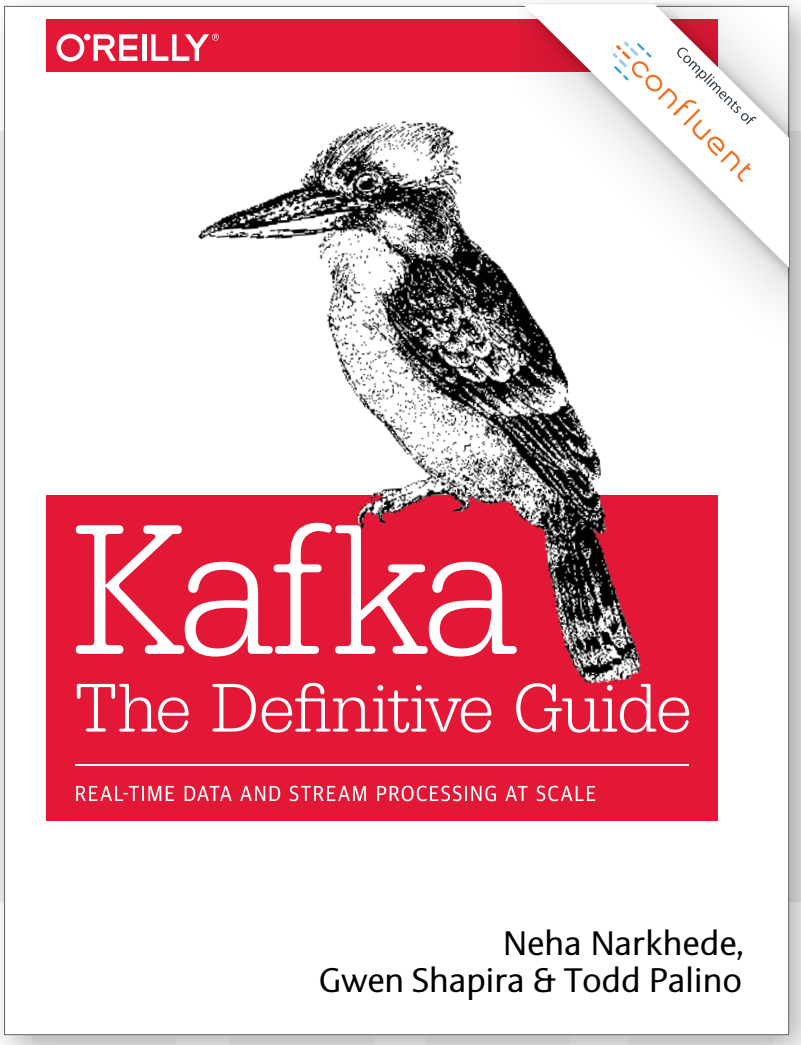
CONTACT US

GENERAL FEEDBACK feedback@infoq.com
ADVERTISING sales@infoq.com
EDITORIAL editors@infoq.com

confluent

Kafka: The Definitive Guide

Real-time data and stream processing at scale



- Deep dive into Apache Kafka's internal design
- Get best practices for developing applications
- Understand how to deploy Kafka in production
- Review detailed use-cases

Download the FREE Book

A LETTER FROM THE EDITOR

Daniel Bryant

With the rise of technologies like Apache Kafka, Apache Beam and Spark Streaming, the topic of stream processing is becoming increasingly popular. Commercial businesses are being formed around the associated open source technology, conference talks are filled with stories of migrations from batch Extract-Transform-Load (ETL) to stream processing, and blog posts and online discussions debate important concepts like if it is really possible to implement exactly once processing (as shown in the second article, the answer is yes, with caveats). This InfoQ emag aims to cut through some of the hype, and introduce you to core stream processing concepts like the log, the dataflow model, and implementing fault-tolerant streaming systems.

The first article summarises an excellent QCon San Francisco presentation by Frances Perry and Tyler Akidau, and explores the fundamentals of stream processing with the dataflow model and the corresponding Apache Beam implementation. The dataflow model encourages engineers to ask four questions in order to understand the approach required when processing data: what are you computing? Where in event time? When in processing time? and how do refinements relate? Apache Beam is the practical implementation of this model, and includes: the unified Beam Model (the what / where / when / how); SDKs for writing data processing pipelines using the Beam Model APIs; and "runners" for executing the data processing pipelines using existing distributed processing backends like Apache Flink or Apache Spark.

The stream processing paradigm is similar to many existing concepts, such as event stream processing and reactive processing. At the core of many of the implementations of these concepts is a distributed transaction log. A log -- an append only, totally ordered data structure -- is a powerful primitive for building distributed systems. Many RDBMS use change logs (or "write ahead logs") to improve performance, for point-in-time-recovery (PITR) after a crash, and also for distributed replication. The second article in this emag explores how the Amazon DynamoDB team exposed the transaction/change log of the DynamoDB NoSQL service to end-user engineers as "DynamoDB Streams" -- a Kinesis Data Stream. Akshat Vig and Khawaja Shams discussed how understanding the concepts of ordering, deduplication, and checkpointing is vital for building correct systems, particularly distributed (streaming-) based systems.

Apache Kafka is an open source stream processing platform that is designed as a "massively scalable pub/sub message queue architected as a distributed transaction log". At QCon San Francisco Neha Narkhede argued that "logs unify batch and stream processing", and platforms like Kafka can be used to create the next generation of ETL systems. This concept is explored within the third article of the series. Traditional approaches to data integration often end up "looking like a mess", with custom scripts, ESBs, MQs, custom middleware, and Hadoop deployments being woven together to provide a bespoke solution that was focused on batch processing. Kafka enables the building of real-time streaming data pipelines

from "source" to "sink" by providing the Kafka Connect API -- a series of pluggable data input/output connectors for the 'E' and 'L' in ETL; and the Kafka Streams API -- a fluent DSL for stream processing with operators such as join, map, filter and windowed aggregates, which is effectively the 'T' in ETL.

The fourth article presents a case study of how Shirya Arora and the Data Engineering and Analytics team at Netflix migrated an existing batch ETL data processing system to a real-time stream processing system using Apache Flink. Arora cautioned that there are many decisions and tradeoffs that must be made when moving from batch to stream data processing - engineers should not "stream all the things" just because stream processing technology is popular. This case study demonstrated that there were clear business wins for using stream processing, including the opportunity to train machine learning algorithms with the latest data, and the creation of opportunities for new kinds of machine learning algorithms. There were also technical wins for implementing stream processing, such as the ability to save on storage costs, faster turnaround time on error correction, and integration with other real-time systems.

The final article in this series summarises Anton Gorshov's recent QCon New York presentation, where he explains in detail how the Core Front Office Platform team at Goldman Sachs provision and operate a large-scale on-premise Apache Kafka cluster. The team has invested significant resources into preventing data loss, and this includes providing tape backup, nightly batch replication, and synchronous disk level replication. There has also been significant investment in creating tooling to support their infrastructure, including a REST-like service and web application to provide insight into the Kafka cluster, and the creation of a comprehensive metrics capture component. The core takeaway is that failure will occur, and engineers must plan to handle this. The approach that has been adopted at GS is to run everything with high-availability ("belt and suspenders"), and be transparent in all of the approaches and trade-offs made to ensure resilience.

CONTRIBUTORS

Daniel Bryant

is leading change within organisations and technology. His current work includes enabling agility within organisations by introducing better requirement gathering and planning techniques, focusing on the relevance of architecture within agile development, and facilitating continuous integration/delivery. Daniel's current technical expertise focuses on 'DevOps' tooling, cloud/container platforms and microservice implementations.

Akshat Vig

works in DynamoDB (Amazon Web Services) as Senior Software engineer with 7 years of experience on distributed systems.

Anton Gorshkov

is a managing director at Goldman Sachs Asset Management where he runs a global Core Platform team, focusing on GSAM's data strategy and real-time services.

Frances Perry

is a software engineer at Google. After many years working on Google's internal data processing stack, she joined the Cloud Dataflow team to make this technology available to external cloud customers.

Neha Narkhede

is co-founder and CTO at Confluent, a company backing the popular Apache Kafka messaging system. Prior to founding Confluent, Neha led streams infrastructure at LinkedIn.

Khawaja Shams

is the Head of Engineering for NoSQL at Amazon Web Services. Prior to Amazon, he led the Data Services team at NASA Jet Propulsion Laboratory.

Shriya Arora

is a senior data engineer at Netflix.

Tyler Akidau

is a Staff Software Engineer at Google where he's spent six years working on massive-scale streaming data processing systems.

View Full Presentation

KEY TAKEAWAYS

Data captured within modern systems has become increasingly "big", and may be generated as an unordered and (effectively) infinite stream. Data may also be captured with unknown delays, particularly if it is collected via an unreliable (distributed) network

The Google Dataflow Model -- and corresponding Apache Beam implementation -- encourages users to ask four questions in order to understand the approach required when processing data: what are you computing? Where in event time? When in processing time? and how do refinements relate?

The Apache Beam project includes three things: The conceptual unified Beam Model (the what / where / when / how); SDKs for writing data processing pipelines using the Beam Model APIs; and "runners" for executing the data processing pipelines using existing distributed processing backends like Apache Flink or Apache Spark

EXPLORING THE FUNDAMENTALS OF STREAM PROCESSING WITH THE DATAFLOW MODEL AND APACHE BEAM

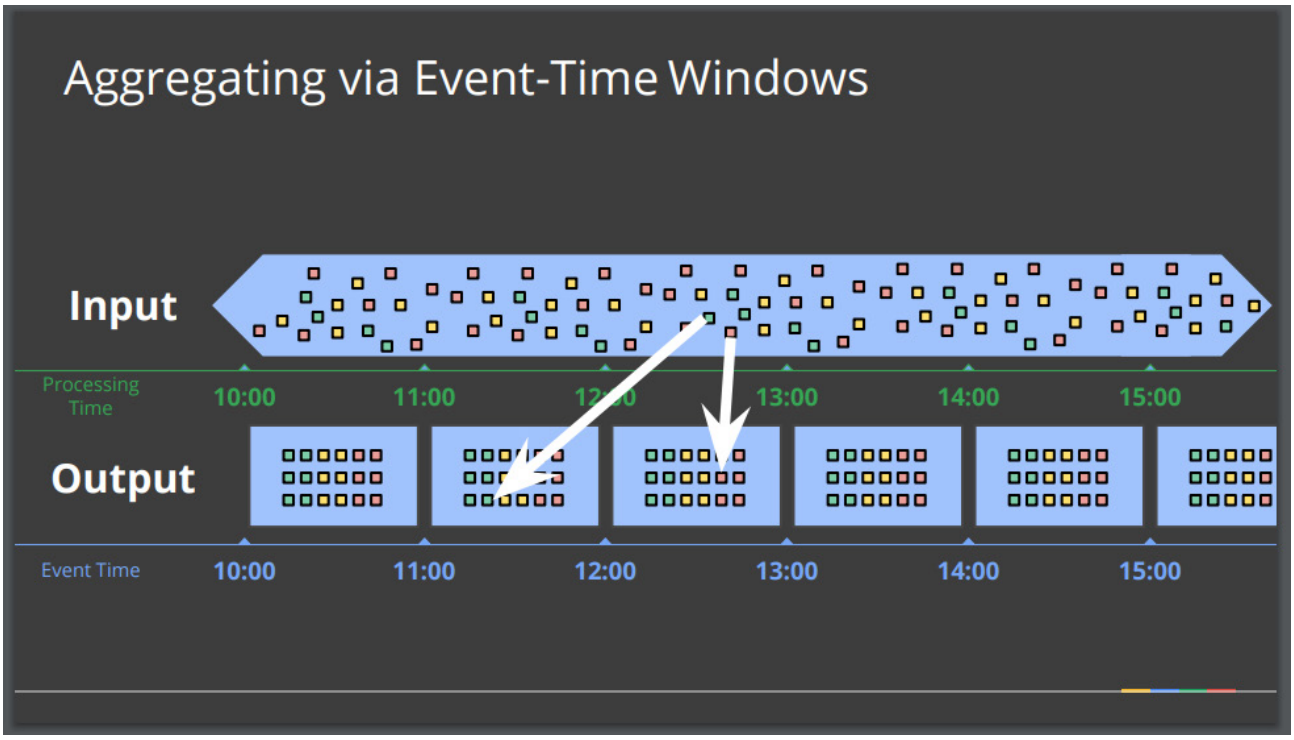
At QCon San Francisco 2016, Frances Perry and Tyler Akidau presented "Fundamentals of Stream Processing with Apache Beam".

Perry and Akidau, both senior staff engineers at Google, began the talk with a discussion of how data captured within modern systems has become increasingly "big", and may be generated as an unordered and (effectively) infinite stream. This can make it challenging for data-processing systems and end users to extract meaningful and timely results and insight for the business. For example, capturing ongoing player scores from a mobile-game application results in a continual stream of data, and the business may want to mine this data in order to understand and improve player retention or "stickiness". In addition to being unordered, data may also be captured with unknown delays, particularly if it is collected via an unreliable network: data may arrive delayed by a few seconds due to a network glitch, a few minutes due to loss of signal, or potentially hours (or days) delayed if the player continues to play the game aboard a transatlantic flight without mobile reception until they land.

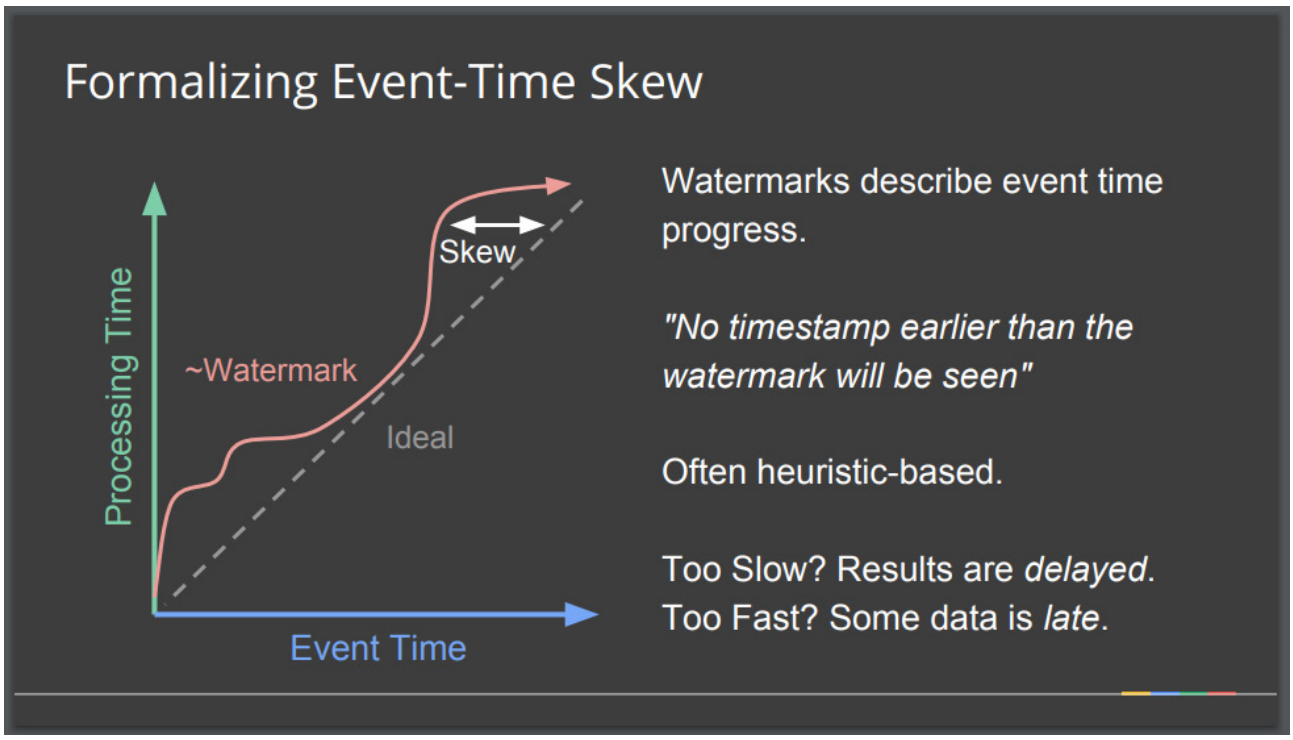
Some data processing is relatively straightforward — for example, element-wise transformations like parsing, translating, or filtering. However, a large amount of data processing requires aggregation operations such counting and joining, and this means that the stream of data must be chopped up in finite chunks before the aggregation can occur and a result be emitted. The logical approach to this woulda be to divide the stream into processing time windows — for example, two-minute or one-hour chunks — but the challenge with this approach is the potential late arrival of data. This can lead to processing data out of context, where the processing time is significantly different than the original event time, which may be a problem for some algorithms. Somehow, late arriving data needs to be shuffled back into the appropriate time window and context from which it originated.

More on this

APACHE BEAM FOUNDER TYLER AKIDAU DISCUSSES STREAMING SYSTEM AND THEIR COMPLEXITIES



Although this reshuffling of late-arriving data makes conceptual sense, it can be challenging to implement. In an idealized world, the event data would be processed as it was generated, but in reality there is a variable skew between the event generation and processing time for which a formal method needs to account and compensate. The solution is to use a watermark to describe event-time progress. A watermark is essentially a timestamp, and when the processing system receives a watermark, it assumes that it is not going to see any message older than that timestamp. A watermark can be perfect — for example, with data taken from a static set of sequentially increasing log files — or heuristic, where the system has to best guess about when all events for a given time window have arrived.



The Apache Beam project includes three things: The unified Beam Model; SDKs for writing data processing pipelines; and "runners" for executing pipelines using backends like Apache Flink or Apache Spark

If a watermark is too slow, the system waits for late data to arrive, and the computational results of the stream-processing operation may be delayed. If a watermark is too fast, then some data arrives late, and an early (speculative) result that was emitted may have to be updated. The reality is that many modern systems will be processing infinite streams or unordered data that is collected via a distributed system and so the data-processing system must account for these issues.

The bulk of the talk explored the challenges of modern stream processing and used the Dataflow model alongside the corresponding practical implementation of this model in the Apache Beam API in order to ask four questions in order to understand the approaches required when processing data:

- What are you computing?
- Where in event time?
- When in processing time?
- How do refinements relate?

For the question "What are you computing?", the answer may be element-wise (single-element) processing, perhaps a translation or filter — this is effectively the map part of the popular MapReduce paradigm — or it may be aggregating, such as a join or a count — this can be thought of as the reduce part of MapReduce. The answer to this question could also involve composite operations: these are operations that are made up of primitives, but we would like to think of them as conceptually simpler higher-level operations. The talk presented an example of using the Apache Beam API and pseudo Java code that computes integer sums from the exemplar mobile-phone game app. The `PCollection` abstraction represents a potentially distributed, multi-element data set: we can think of a `PCollection` as pipeline data, and Beam transforms use `PCollection` objects as inputs and outputs.

For the next question, "Where in event time?", the typical approach is to use windowing to divide data into event-time-based

What: Computing Integer Sums

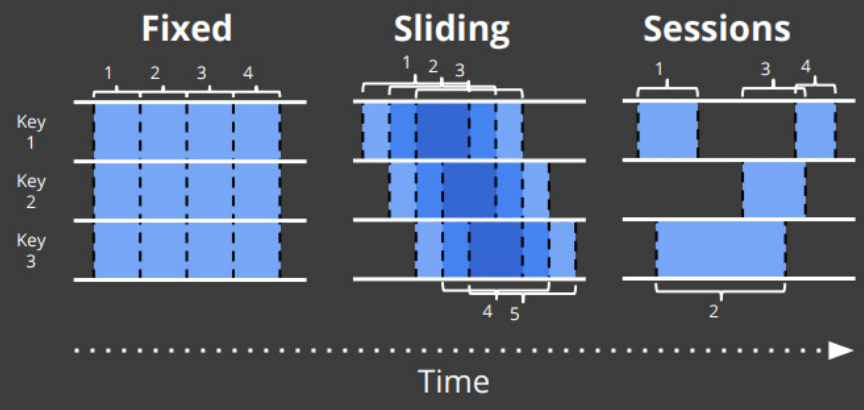
```
// Collection of raw log lines
PCollection<String> raw = IO.read(...);

// Element-wise transformation into team/score pairs
PCollection<KV<String, Integer>> input =
      raw.apply(ParDo.of(new ParseFn());

// Composite transformation containing an aggregation
PCollection<KV<String, Integer>> scores =
      input.apply(Sum.integersPerKey());
```

What Where When How

finite chunks. These windows can be fixed (e.g., every five minutes), sliding (e.g., the previous 24 hours for every hour) or session-based (e.g., an application-specific burst of activity). Windowing is very similar to the concept of using a composite key to group data within batch processing.

Where in event time?

Windowing divides data into event-time-based finite chunks.



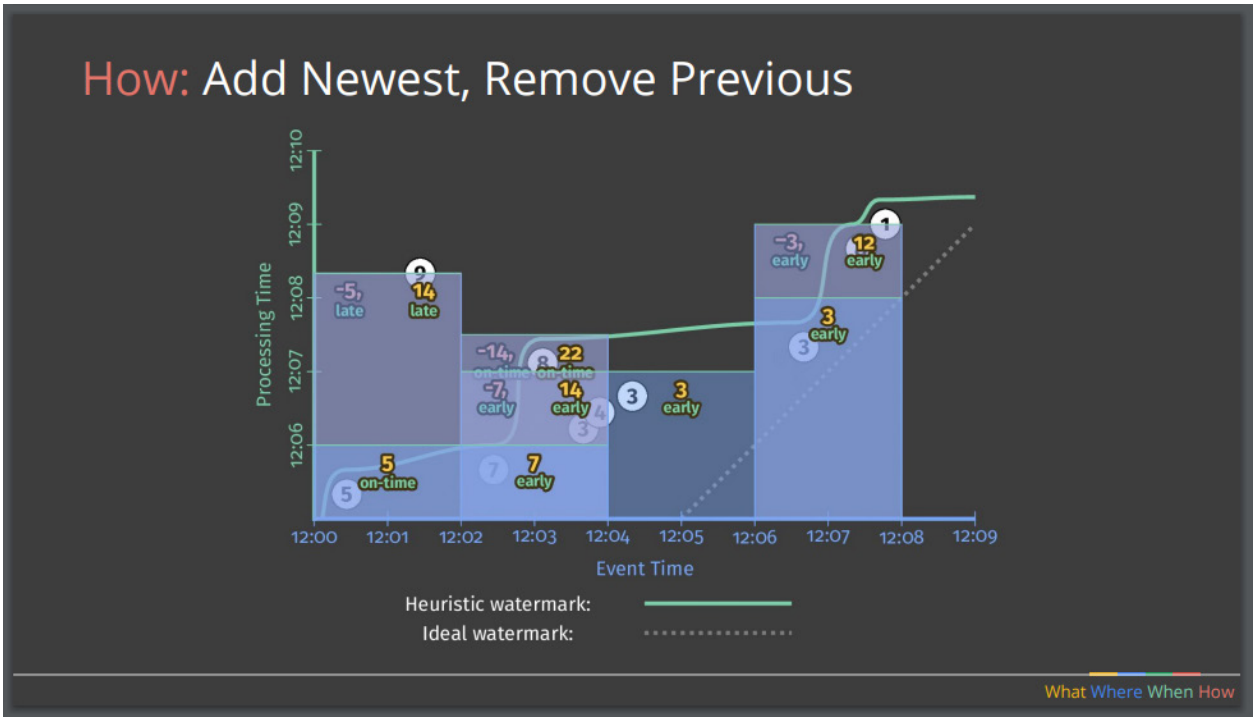
Often required when doing aggregations over unbounded data.

What Where When How

The third question, "When in processing time?", relates to the requirements for triggers that control when results are emitted. Triggers are often relative to the watermark — when the watermark is seen, we believe that all the results for this event-time have been seen, and therefore the computation result can be emitted. If a perfect watermark is used, this can lead to the late emission of results as the system waits for all data elements to be processed before emitting any result. If a heuristic watermark is used, the results can be emitted in a more timely fashion, but the processing of late elements can provide additional challenges.

The issues with defining triggers can be mitigated by using early and late firings. An early firing can provide a speculative result at specific time periods — i.e., after every minute — and late firing allows results to be updat-

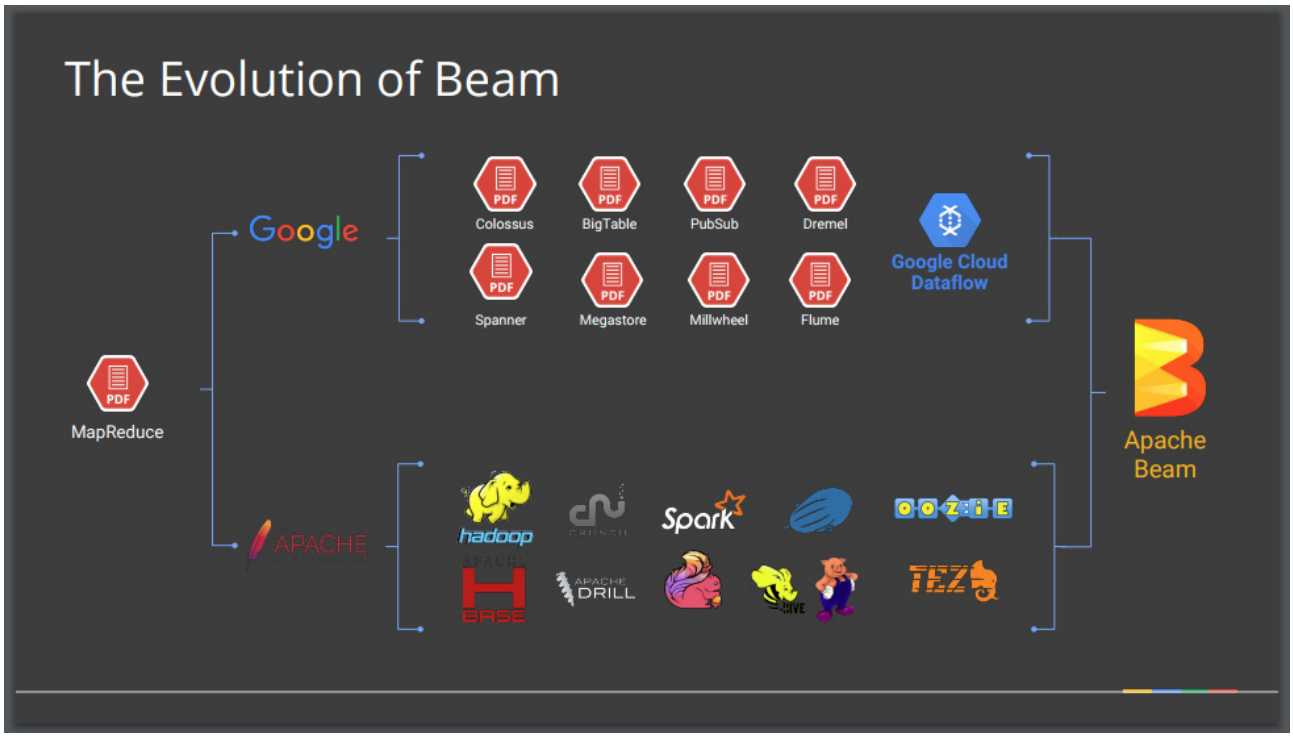
ed as late arriving data is processed. A key consideration when using early and late triggers relates to the fourth question, "How to make refinements (or updates) on speculative results?" Three tactics are available: discarding, simply throwing away any previous speculative results; accumulating, updating the result emitted for every update; or accumulating and retracting, updating the result emitted for every update but also issuing a retraction on the previous result. There are tradeoffs with each approach between the ease of implementation and the correctness of last observed and total observed values. This must be taken into account if, for example, a downstream service is performing multiple aggregations within a distributed pipeline — without the issuing of a retraction accompanying an update, the wrong result may be computed.



The final section of the talk examined properties that make the Apache Beam model of stream processing "awesome". The first property discussed was correctness. It has historically been challenging to achieve correctness in a distributed stream-processing system; the very nature of a distributed system means that data will arrive with variable latency. However, modern stream-processing systems provide primitives to allow engineers to make tradeoffs between accuracy and result-emission latency. The second property is power. The Apache Beam API provides an engineer with powerful primitives and abstractions that can be implemented with relative ease. For example, the fluent-style Beam DSL allows easy swapping of approaches and algorithms, such as changing from a fixed window-aggregation strategy to a session-based one. This also relates to the third useful property of composability, as it is easy to compose new pipelines within the Beam API in order to test new hypothesis or experiment with the data. The final two properties, flexibility and modularity, allow various approaches to processing the data with minimal (and easily understandable) changes to code.

The concepts for Apache Beam evolved from the original 2004 MapReduce paper, which were in turn further refined at Google through the creation of internal systems like Bigtable, Dremel, Spanner, and MillWheel. Although Google focused on satisfying internal requirements with these systems, company engineers published a series of papers at conferences and within academic journals, and this led to the creation of a vibrant open-source ecosystem formed around these ideas.

This in turn led to the creation of numerous successful open-source Apache projects like Hadoop, Drill, Spark, and Tez. In 2014, Google Cloud Platform began to offer Cloud Dataflow, which is a fully managed stream and batch data-processing service created based on the years of experience working on the internal data-processing systems. There were two parts to Cloud Dataflow: the Dataflow programming model discussed previously and the SDK, a "no knobs" managed service for executing the models. Google ultimately contributed the Dataflow model and SDK to the open-source community as the Apache Beam project.



The Apache Beam project includes three things:

1. The conceptual Beam model

- The what/where/when/how model presented in the talk

2. SDKs for writing Beam pipelines

- Java SDK
- Python SDK

3. Runners for existing distributed processing back ends

- Apache Flink
- Apache Spark
- Google Cloud Dataflow
- Apache Apex
- Apache Gearpump (incubating)
- Direct Runner for local development and testing

Fundamentally, the Beam model attempts to generalize the semantics of this modern style of data processing and provides three core levels of abstraction for various personas within the data-processing community: end users who simply want to write data pipelines or transform libraries in a language that is either familiar to them or that their organization has invested in, SDK writers who want to make Beam concepts available in new languages, and runner writers who have a distributed processing environment and want to support Beam pipelines. It is worth noting that since not all runners offer the same capabilities (although many are converging), the Apache Beam project has created a series of runner capability matrices that provide further details.

Additional information on the topic of streaming fundamentals can be found in Akidau's articles "The world beyond batch: Streaming 101" and "The world beyond batch: Streaming 102". The Apache Beam website also contains many useful references and tutorials, and the Beam community offers user and developer mailing lists at user-subscribe@beam.apache.org and dev-subscribe@beam.apache.org.

The complete video for the talk Perry and Akidau presented at QCon SF, "Fundamentals of Stream Processing with Apache Beam", can be found on InfoQ.

View Full Presentation

KEY TAKEAWAYS

A log -- an append only, totally ordered data structure -- is a powerful primitive for building distributed systems. Many RDBMS use change logs (or "write ahead logs") to improve performance, for point-in-time-recovery (PITR) after a crash, and also for distributed replication

The Amazon DynamoDB team exposed the underlying DynamoDB change log to end-user engineers as "DynamoDB Streams" -- a Kinesis Data Stream

Understanding ordering and the effects of message duplication are vital for building correct systems, particularly distributed systems

It is not possible to obtain exactly-once end-to-end delivery within a distributed system, although systems can be designed to provide exactly-once processing. This is typically implemented using checkpointing and deduplication/idempotency filters

"Checkpointing" consists of maintaining a pointer that specifies the latest transaction that has been read or processed within a log, which indicates the current state of processing

DEMYSTIFYING DYNAMODB STREAMS: AN INTRODUCTION TO ORDERING, DEDUPLICATION AND CHECKPOINTING

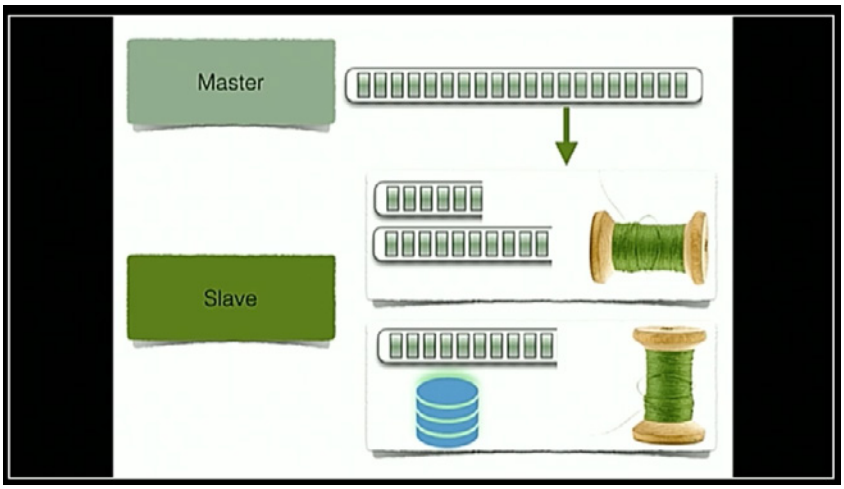
At QCon San Francisco 2016, Akshat Vig and Khawaja Shams presented "Demystifying DynamoDB Streams".

The core takeaway from the talk was that an append-only, totally ordered log data structure is a powerful primitive for building a distributed system, but engineers using this technology must understand the key principles of ordering, deduplication, and checkpointing. They explored these concepts in depth and provided practical examples using DynamoDB Streams, which is effectively an end-user service that exposes the underlying change log of the Amazon DynamoDB data-store technology.

Shams, VP of engineering at AWS Elemental, began the talk by discussing the longstanding relationship between relational-database technology and the log data structure. Many RDBMSs use change logs — such as MySQL's binary log or PostgreSQL's write-ahead log — to improve performance, provide point-in-time recovery (PITR) after a crash, and implement replication. A database change log also effectively allows the creation of a distributed database via replication of data on additional external hosts. For example, MySQL replicates data between a master and associated slave hosts through its binary log. Each slave maintains two threads for the process of replication: one to continually write a copy of the master binary log on the local slave's disk and one to sequentially read through the log and apply each transaction to the local replicated copy of the database.

In order to build something even as simple as a master-slave replication, there are several primitives to understand. The first and foremost is ordering. Imagine if two transactions were to be applied sequentially to a database — the first writes a new entry and the second deletes this entry, which ultimately results in no data persisting in the database — but if the ordering is not guaranteed, the delete transaction could be processed first (causing no effect) and then the write transaction applied, which results in data incorrectly persisting in the database. The second core primitive is duplication: each single transaction should appear exactly once within the log. Failure to enforce ordering or prevent duplication within a log can result in the master and slave becoming inconsistent.

Shams posed the question of why these core primitives are important, given that not many engineers will be writing crash-recovery tooling or implementing distributed databases. He quickly pointed out that many engineers are indeed building distributed systems — such as those based on the microservices architecture — and often rely on log processing and event-driven architecture to share data, for example, via



Apache Kafka or Amazon Kinesis. Accordingly, an awareness of these fundamental log-processing concepts is essential.

Vig, senior software engineer at AWS, discussed the concept of checkpointing. As the log is being processed, a pointer must be kept in order to specify the latest transaction that has been read or processed. In essence, this indicates the current state of processing. If the processing of the log is interrupted — for example, by a crash in the consuming system — the current checkpoint can be examined and processing resumed from the indicated transaction. This not only prevents extra work but also attempts to ensure accuracy by preventing the reprocessing of transactions that have already been seen.

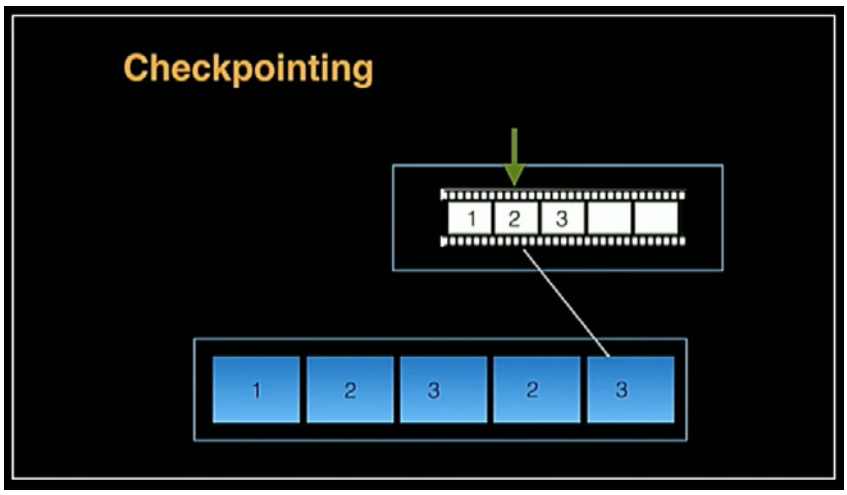
There are multiple strategies to checkpointing, each of which is a trade-off between specificity and throughput. For example, checkpointing after every transaction is processed provides a specific pointer to the last read item, but is high cost in that it requires a checkpoint write for every item. Checkpointing after every, say, 10 reads is less specific to the exact item that has most recently been processed but requires an order of magnitude less cost in checkpoint writes.

On the other hand, the larger the number of transactions that occur between checkpoints, the more transactions that have to

be read (and processed) if a crash does occur. Not only does this take time but, Shams cautioned, in some systems it may be unacceptable to reprocess transactions (which could give the impression of going back through time) and therefore the only strategy in this case is to checkpoint every processed transaction.

Vig presented a series of trends that the AWS team has identified within current software architectures. The first, optimizing global latency, occurs when engineers attempt to provide a consistent experience across different geographic regions — for example, by implementing cross-region replication of data. There are multiple approaches to implementing this; for example, engineers can build the geo-replication logic directly into the application, or transactions can be written to a distributed queue before being processed independently within each geographic region. However, none of these implementations is trivial, and the complexity increases as the transaction throughput and number of supported geographic regions increases.

The second trend, protection of logical corruption, is an attempt by engineers to prevent application-level issues that are not physical in nature — for example, if a new release of an application incorrectly mutates data by accident, the system should be able to recover. Typical approaches to this include using PITR snapshots or delayed replicas that maintain a live copy of a data store at various points in time (for example, multiple live standby databases of one running one hour in the past, a second running two hours in the past, etc.). There is obviously an operational cost to implementing and maintaining both of these approaches.



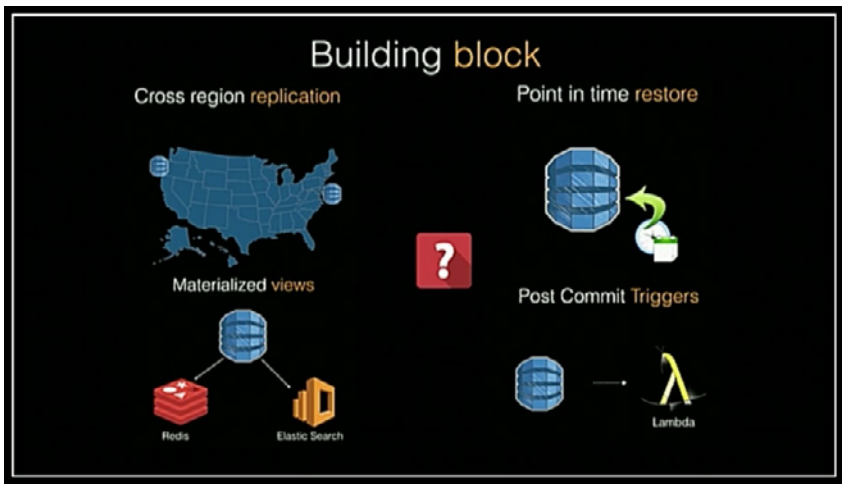
The third trend AWS detected within modern architectures is the requirement for flexible querying of data: engineers frequently want both online transaction processing (OLTP) and online analytical processing (OLAP) queries on data within a system. Storing the single source of truth within a log allows the processing of data multiple times, and the data can also materialize in a variety of ways. This can often be seen within event sourcing (ES) and command-query responsibility segregation (CQRS) architectures, by which data materializes using a data-store technology most appropriate to each query use case — for example, using a graph database for queries involving connectedness or shortest-path algorithms in combination with a RDMS for relational queries and a key-value store for direct identifier-driven lookups.

The fourth trend Vig discussed was the rise in popularity of event-driven architectures (EDA), in which engineers are increasingly building systems that process streams of events, potentially in parallel. An example of this style of architecture can be seen within function-as-a-service (FaaS) serverless applications, in which engineers write functions that are triggered by an event, such as a write to an object store or a request from an API gateway.

The presentation shifted gears in the second half as Shams and Vig discussed how the DynamoDB team at AWS attempted to provide building blocks that end-user engineers could use to implement solutions that address the trends identified above. The solution was AWS DynamoDB Streams, which essentially exposes the change log of DynamoDB to engineers as an Amazon Kinesis Stream. Following the principles discussed earlier in the presentation, DynamoDB Streams are highly available, durable, ordered and deduplicated. It is worth noting that at the 2017 AWS re:Invent conference, Amazon announced end-user services for DynamoDB cross-region replicated Global Tables and automated on-demand DynamoDB backup, presumably built using these primitives.

Shams and Vig presented a sample voting application that was built using DynamoDB Streams. Optimistic concurrency can be implemented within systems built upon DynamoDB by using the `put if not exists` command. In the voting application, this could be used to prevent duplicate votes, perhaps those occurring by accident via a logical error within the application. If the `put if not exists` command detects an attempt to make a duplicate write then this write

would fail and no corresponding DynamoDB Stream entry would be generated. A conditional `put` could be used to allow a voter to change their vote within the application, where the condition specified would be a different candidate ID. If the condition is satisfied, then the new write (the vote update) would succeed, and a DynamoDB Stream entry would be generated that contains both the updated `NewImage` and the previous `OldImage` data.

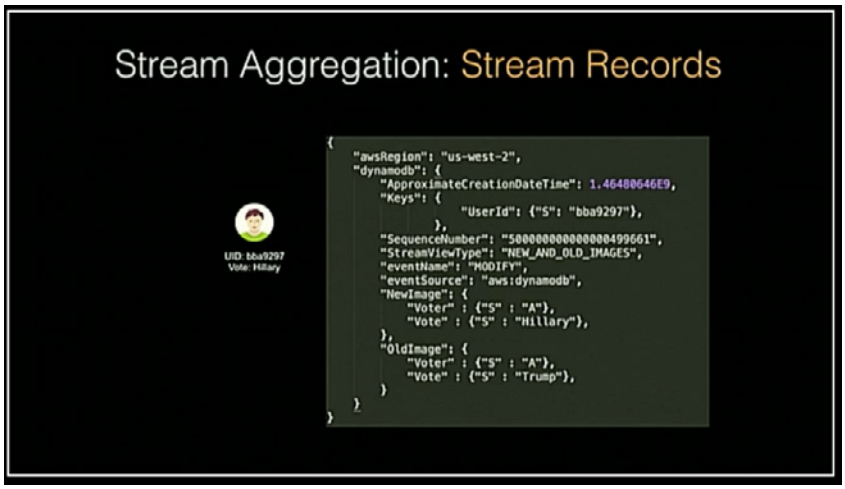


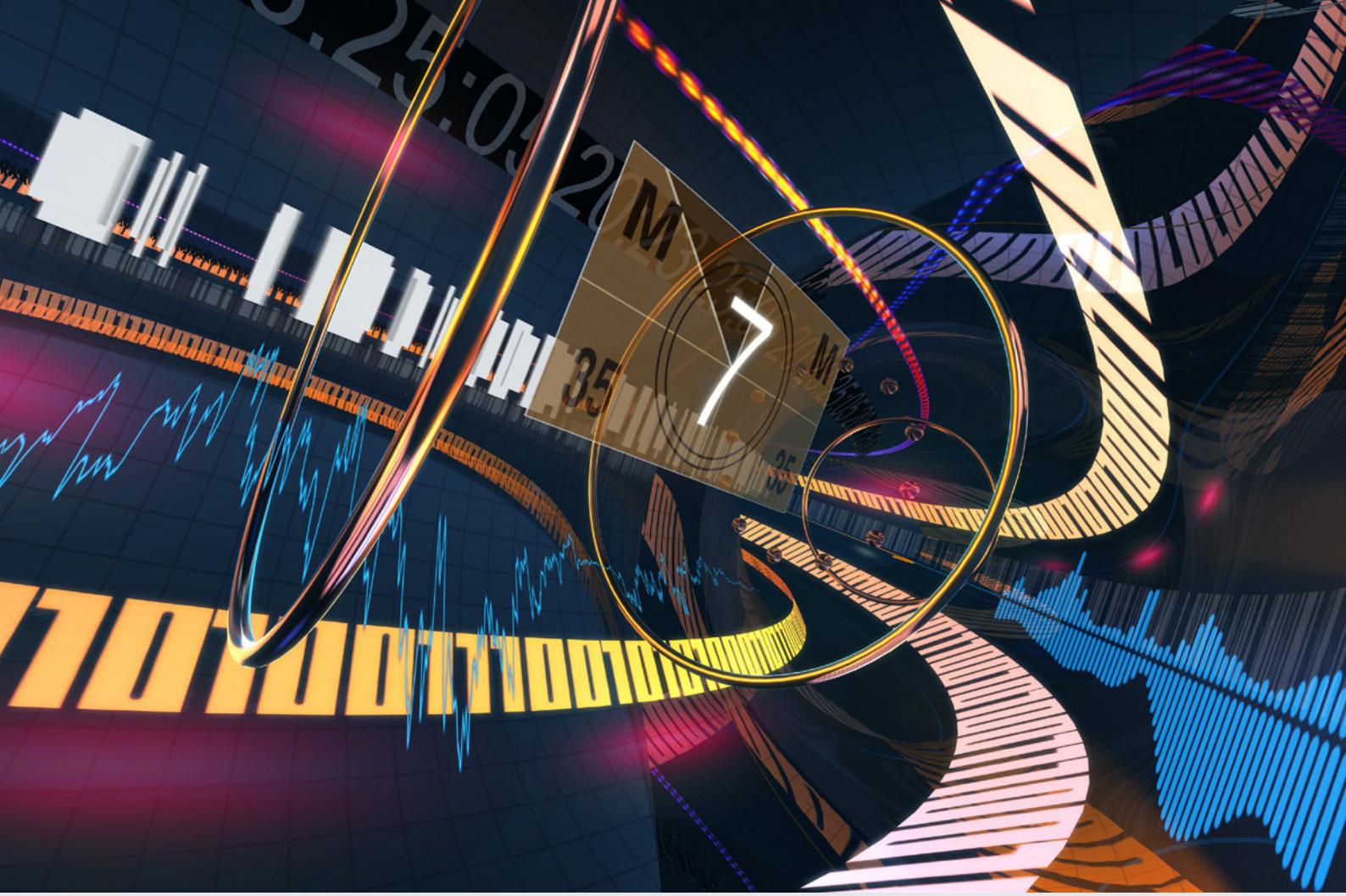
When processing a DynamoDB Stream using Kinesis, the application should checkpoint the logs using the Kinesis client library. As Shams and Vig had noted earlier, this can be used to prevent re-processing of data. Shams cautioned that implementing global ordering within a distributed system is only possible if a single process within the system generates a unique sequence number, and this typically limits throughput. An application can attempt to implement global ordering using timestamps, but this assumes that all processes within the system have access to a reliable and uniformly configured clock. This is generally considered an unreasonable assumption, but it is worth noting that AWS have recently released the Amazon Time Sync Service, which provides a "highly accurate and reliable time reference that is natively accessible from Amazon EC2 instances".

Implementing partial ordering within a system — ordering for each individual item within a DynamoDB table — is, however, relatively simple, as mutations for an individual item are written to the same shard within Kinesis. As long as the application processes data in order within a shard, this will be sufficient.

Vig concluded the talk by stating that it is possible to consume DynamoDB Streams using at-most once or at-least once semantics, but not exactly once. It is not possible to obtain exactly once end-to-end delivery within a distributed system, although systems can be implemented to provide exactly once processing. This is typically implemented using deduplication or idempotency filters.

The full video for Shams and Vig's QCon SF 2016 presentation "Demystifying DynamoDB Streams" can be found on InfoQ.





View Full Presentation

KEY TAKEAWAYS

Several recent data trends are driving a dramatic change in the old-world batch Extract-Transform-Load (ETL) architecture: data platforms operate at company-wide scale; there are many more types of data sources; and stream data is increasingly ubiquitous

Enterprise Application Integration (EAI) was an early take on real-time ETL, but the technologies used were often not scalable. This led to a difficult choice with data integration in the old world: real-time but not scalable, or scalable but batch.

Apache Kafka is an open source streaming platform that was developed seven years ago within LinkedIn

Kafka enables the building of streaming data pipelines from "source" to "sink" through the Kafka Connect API and the Kafka Streams API

Logs unify batch and stream processing. A log can be consumed via batched "windows", or in real time by examining each element as it arrives

IS BATCH ETL DEAD, AND IS APACHE KAFKA THE FUTURE OF DATA PROCESSING?

At QCon San Francisco 2016, Neha Narkhede presented "ETL is Dead; Long Live Streams", and discussed the changing landscape of enterprise data processing.

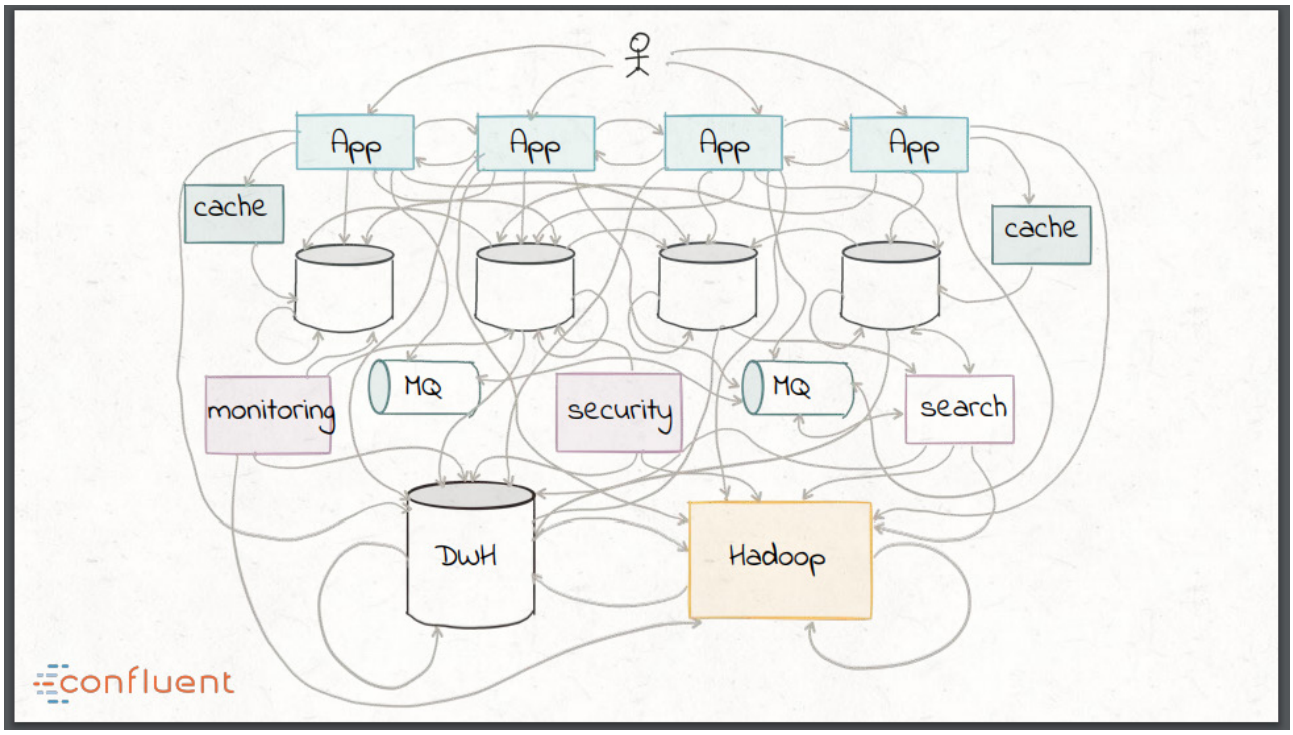
A core premise of the talk was that the open-source Apache Kafka streaming platform can provide a flexible and uniform framework that supports modern requirements for data transformation and processing.

Narkhede, co-founder and CTO of Confluent, began the talk by stating that data and data systems have significantly changed within the past decade. The old world typically consisted of operational databases providing online transaction processing (OLTP) and relational data warehouses providing online analytical processing (OLAP). Data from a variety of operational databases was typically batch-loaded into a master schema within the data warehouse once or twice a day. This data integration process is commonly referred to as extract-transform-load (ETL).

Several recent data trends are driving a dramatic change in the old-world ETL architecture:

- Single-server databases are being replaced by a myriad of distributed data platforms that operate at company-wide scale.
- There are many more types of data sources beyond transactional data: e.g., logs, sensors, metrics, etc.
- Stream data is increasingly ubiquitous, and there is a business need for faster processing than daily batches.

The result of these trends is that traditional approaches to data integration often end up looking like a mess, with a combination of custom transformation scripts, enterprise middleware such as enterprise service buses (ESBs) and message-queue (MQ) technology, and batch-processing technology like Hadoop.

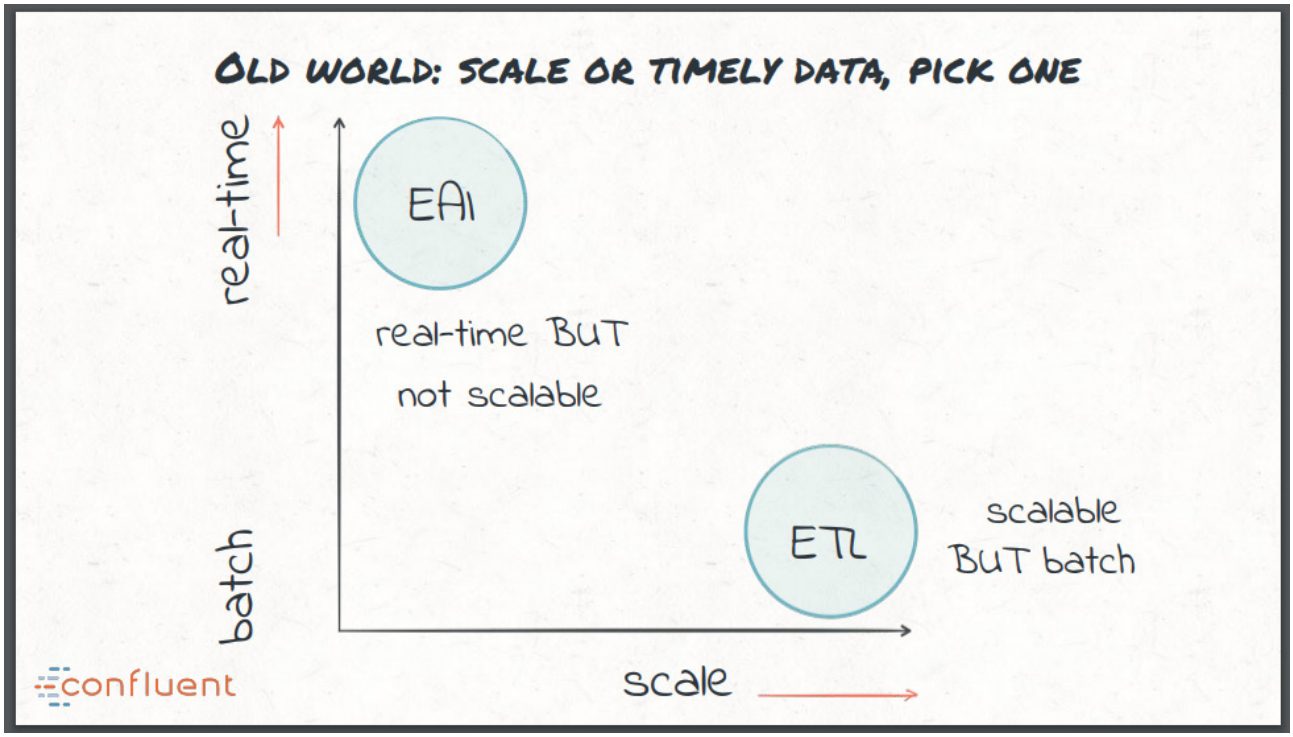


Before exploring how transitioning to modern streaming technology could help to alleviate this issue, Narkhede dove into a short history of data integration. Beginning in the 1990s within the retail industry, businesses became increasingly keen to analyze buyer trends with the new forms of data now available to them. Operational data stored within OLTP databases had to be extracted, transformed into the destination warehouse schema, and loaded into a centralized data warehouse. As this technology has matured over the past two decades, however, the data coverage within data warehouses remains relatively low due to the drawbacks of ETL:

- There is a need for a global schema.
- Data cleansing and curation is manual and fundamentally error prone.
- The operational cost of ETL is high: it is often slow and time and resource intensive.
- ETL tools were built to narrowly focus on connecting databases and the data warehouse in a batch fashion.

Enterprise application integration (EAI) was an early take on

real-time ETL, and used ESBs and MQs for data integration. Although effective for real-time processing, these technologies could often not scale to the magnitude required. This led to a difficult choice with data integration in the old world: real time but not scalable, or scalable but batch.



Narkhede argued that the modern streaming world has new requirements for data integration:

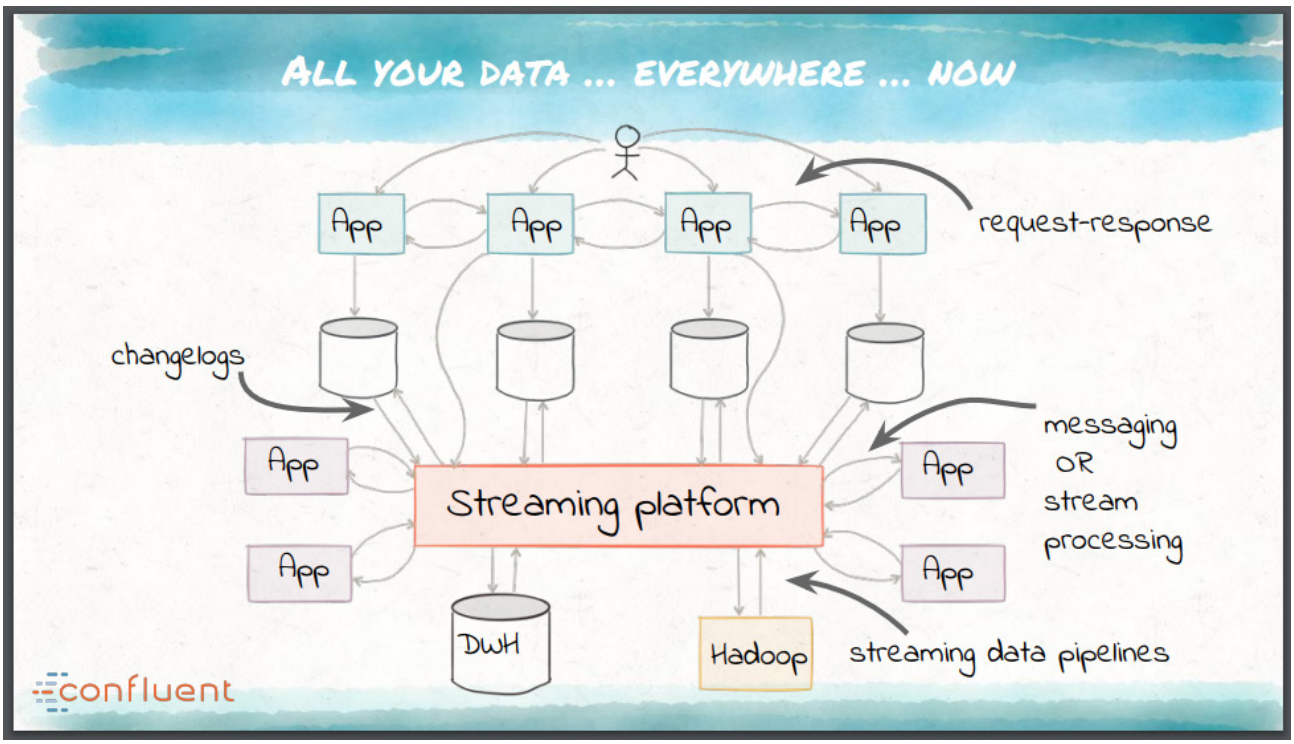
- The ability to process high-volume and high-diversity data.
- A platform must support real-time from the ground up, which drives a fundamental transition to event-centric thinking.
- Forward-compatible data architectures must be enabled and must be able to support the ability to add more applications that need to process the same data differently.

These requirements drive the creation of a unified data-integration platform rather than a series of bespoke tools. This platform must embrace the fundamental principles of modern architecture and infrastructure, and should be fault tolerant, be capable of parallelism, support multiple delivery semantics, provide effective operations and monitoring, and allow schema management. Apache Kafka, which was developed seven years ago within LinkedIn, is one such open-source streaming platform and can operate as the central nervous system for an organization's data in the following ways:

- It serves as the real-time, scalable messaging bus for applications, with no EAI.
- It serves as the source-of-truth pipeline for feeding all data-processing destinations.
- It serves as the building block for stateful stream-processing microservices.

Apache Kafka currently processes 14 trillion message a day at LinkedIn, and is deployed within thousands of organizations worldwide, including Fortune 500 companies such as Cisco, Netflix, PayPal, and Verizon. Kafka is rapidly becoming the storage of choice for streaming data, and it offers a scalable messaging backbone for application integration that can span multiple data centers.

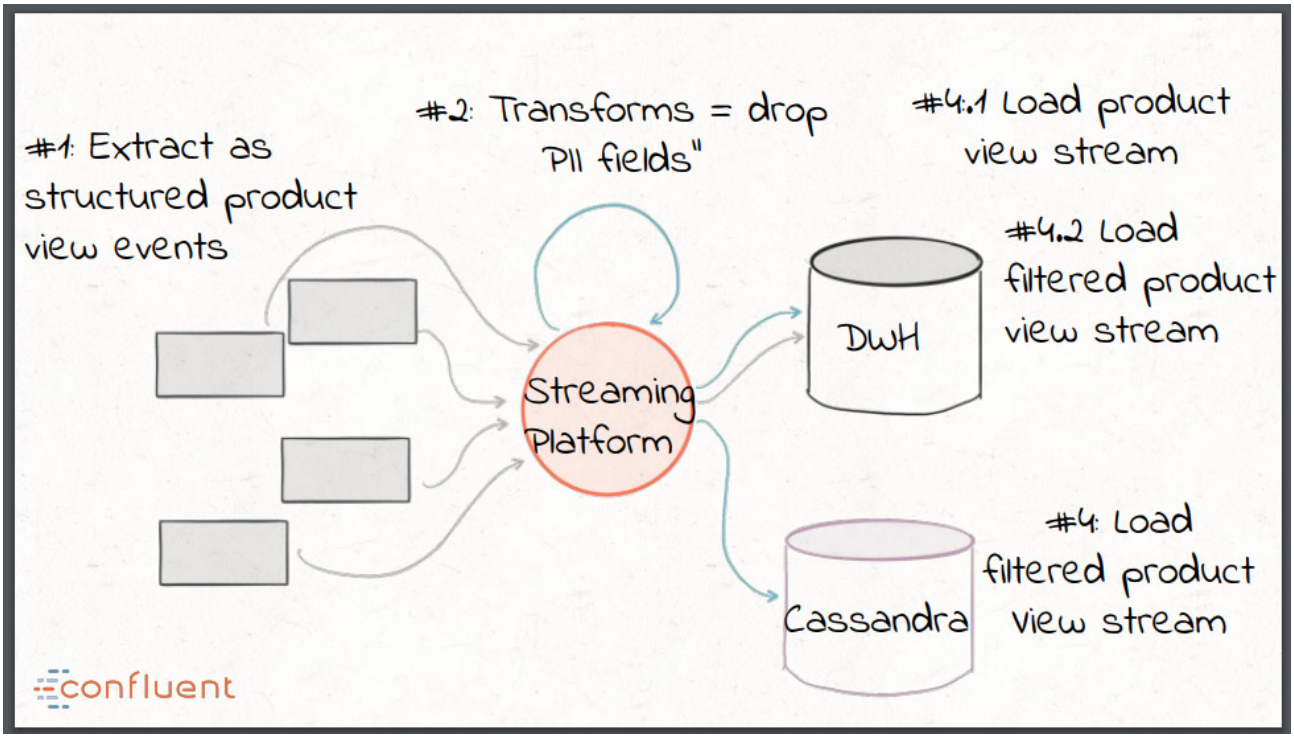
Fundamental to Kafka is the concept of the log; an append-only, totally ordered data structure. The log lends itself to publish-subscribe (pubsub) semantics, as



a publisher can easily append data to the log in immutable and monotonic fashion, and subscribers can maintain their own pointers to indicate current message processing.

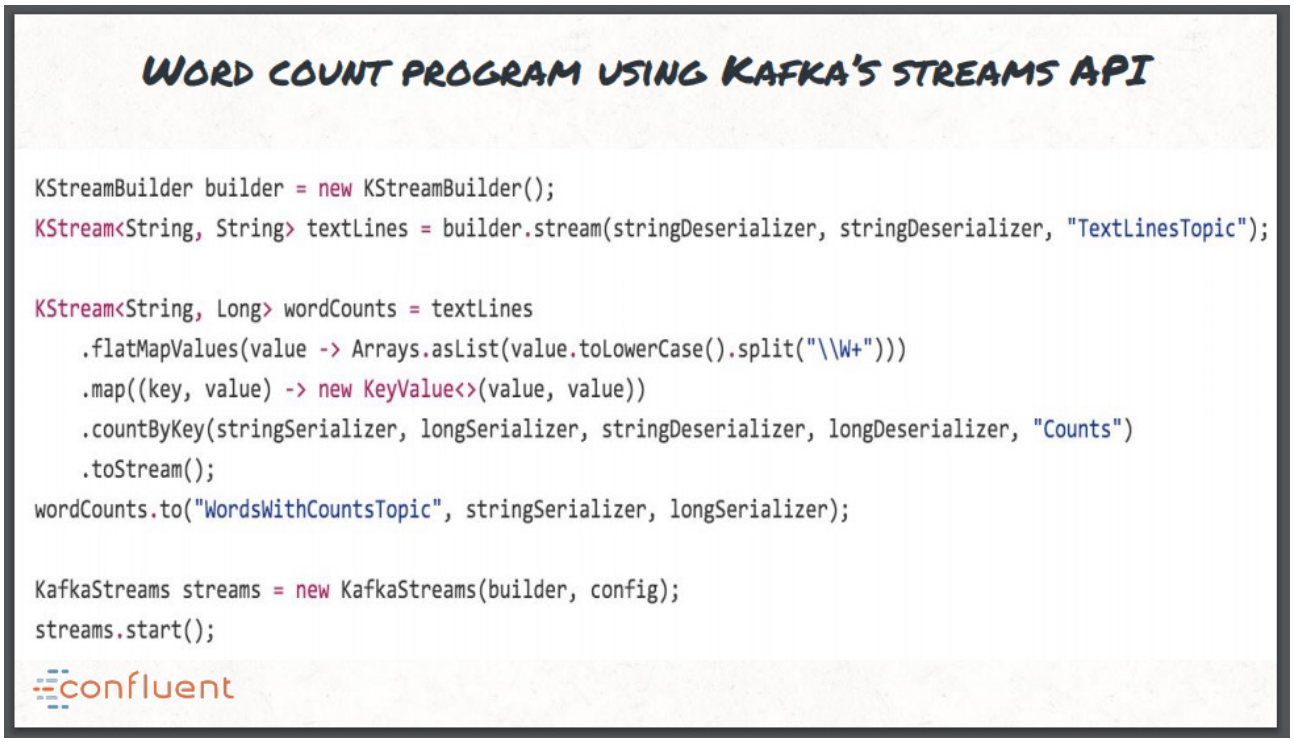
Kafka enables the building of streaming data pipelines — the E and L in ETL — through the Kafka Connect API. The Connect API leverages Kafka for scalability, builds upon Kafka's fault-tolerance model, and provides a uniform method to monitor all of the connectors. Stream processing and transformations can be implemented using the Kafka Streams API — this provides the T in ETL. Using Kafka as a streaming platform eliminates the need to create (potentially duplicate) bespoke extract, transform, and load components for each destination sink, data store, or system. Data from a source can be extracted once as a structured event into the platform, and any transforms can be applied via stream processing.

In the final section of her talk, Narkhede examined the concept of stream processing — transformations on stream data — in more detail, and presented two



competing visions: real-time MapReduce versus event-driven microservices. Real-time MapReduce is suitable for analytic use cases and requires a central cluster and custom packaging, deployment, and monitoring. Apache Storm, Spark Streaming, and Apache Flink implement this. Narkhede argued that the event-driven microservices vision — which is implemented by the Kafka Streams API — makes stream processing accessible for any use case, and only requires adding an embedded library to any Java application and an available Kafka cluster.

The Kafka Streams API provides a convenient fluent DSL, with operators such as `join`, `map`, `filter`, and window aggregates.



This is true event-at-a-time stream processing — there is no micro-batching — and it uses a dataflow-style windowing approach based on event time in order to handle late-arriving data. Kafka Streams provides out-of-the-box support for local state, and supports fast stateful and fault-tolerant processing. It also supports stream reprocessing, which can be useful when upgrading applications, migrating data, or conducting A/B testing.

Narkhede concluded the talk by stating that logs unify batch and stream processing — a log can be consumed via batched windows or in real time by examining each element as it arrives — and that Apache Kafka can provide the "shiny new future of ETL".

The full video of Narkhede's QCon SF talk "ETL Is Dead; Long Live Streams" can be found on InfoQ.

HANDLING GDPR WITH APACHE KAFKA: HOW DOES A LOG FORGET?

By Ben Stopford

If you follow the press around Apache Kafka you'll probably know it's pretty good at tracking and retaining messages, but sometimes removing messages is important too. GDPR is a good example of this as, amongst other things, it includes the right to be forgotten. This raises a very obvious question: how do you delete arbitrary data from Kafka? After all, its underlying storage mechanism is an immutable log.

As it happens, Kafka is a pretty good fit for GDPR. The regulatory regime specifies not only that users have the right to be forgotten, but also have the right to request a copy of their personal data. Companies are also required to keep detailed records of what data is used for—a requirement for which recording and tracking the messages that move from application to application is a boon.

How do you delete (or redact) data from Kafka?

The simplest way to remove messages from Kafka is to simply let them expire. By default, Kafka will keep data for two weeks, and you can tune this to an arbitrarily large (or small) period of time. There is also an Admin API that lets you delete messages explicitly if they are older than some specified time or offset. But businesses increasingly want to leverage Kafka's ability to keep data for longer periods of time, say for Event Sourcing architectures or as a source of truth. In such cases it's important to understand how to make long lived data in Kafka GDPR compliant. For this,compacted topics are the tool of choice, as they allow messages to be explicitly deleted or replaced via their key.

Data isn't removed from compacted topics in the same way as in a relational database. Instead, Kafka uses a mechanism closer to those used by Cassandra and HBase where records are marked for removal then later deleted when the compaction process runs. Deleting a message from a compacted topic is as simple as writing a new message to the topic with the key you want to delete and a null value. When compaction runs the message will be deleted forever.

```
//Create a record in a compacted topic in
kafka
producer.send(new
ProducerRecord(CUSTOMERS_TOPIC,
“Customer123”, “Donald Duck”));
//Mark that record for deletion when
compaction runs
producer.send(new
ProducerRecord(CUSTOMERS_TOPIC,
“Customer123”, null));
```

If the key of the topic is something other than the CustomerId, then you need some process to map the two. For example, if you have a topic of Orders, then you need a mapping of Customer to OrderId held somewhere. Then, to 'forget' a customer, simply look-up their Orders and either explicitly delete them from Kafka, or alternatively redact any customer information they contain. You might roll this into a process of your own, or you might do it using Kafka Streams if you are so inclined.

There is a less common case, which is worth mentioning, where the key (which Kafka uses for ordering) is completely different to the key you want to be able to delete by. Let's say that you need to key your Orders by ProductId. This choice of key won't let you delete Orders for individual customers, so the simple method above wouldn't work. You can still achieve this by using a key that is a composite of the two: make the key [ProductId][CustomerId], then use a custom partitioner in the Producer (see the Producer Config: "partitioner.class") that extracts the ProductId and partitions only on that value. Then you can delete messages using the mechanism discussed earlier using the [ProductId][CustomerId] pair as the key.

Click here to read the full article

View Full Presentation

KEY TAKEAWAYS

There are many decisions and tradeoffs that must be made when moving from batch ETL to stream data processing. Engineers should not "stream all the things" just because stream processing technology is popular

The Netflix case study presented here migrated to Apache Flink. This technology was chosen due to the requirements for real-time event-based processing and extensive support for customisation of windowing

Many challenges were encountered during the migration, such as getting data from live sources, managing side (metadata) inputs, handling data recovery and out of order events, and increased operational responsibility

There were clear business wins for using stream processing, including the opportunity to train machine learning algorithms with the latest data

There were also technical wins for implementing stream processing, such as the ability to save on storage costs, and integration with other real-time systems

MIGRATING BATCH ETL TO STREAM PROCESSING: A NETFLIX CASE STUDY WITH KAFKA AND FLINK

At QCon New York, Shriya Arora presented "Personalising Netflix with Streaming Datasets" and discussed the trials and tribulations of a recent migration of a Netflix data processing job from the traditional approach of batch-style ETL to stream processing using Apache Flink.

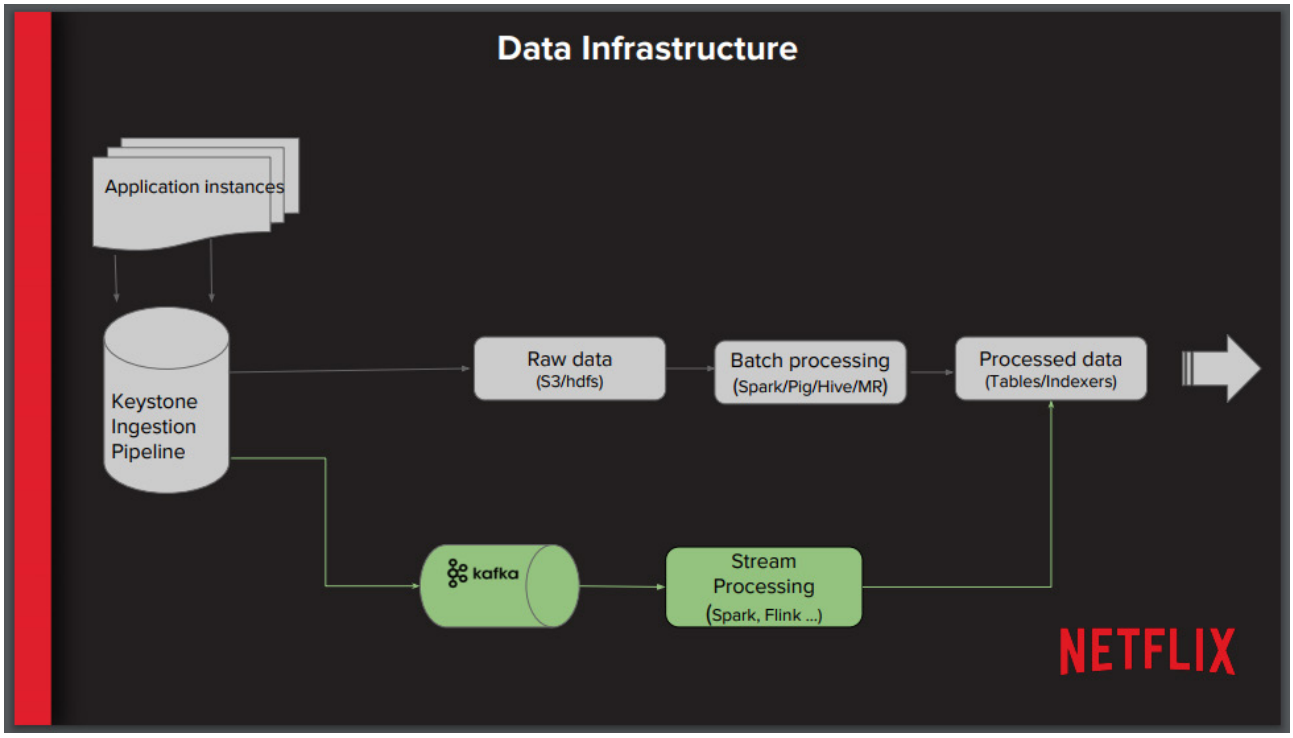
Arora, a senior data engineer at Netflix, began by stating that the key goal of the presentation was to help the audience decide if a stream-processing data pipeline would help resolve problems they may be experiencing with a traditional extract-transform-load (ETL) batch processing job. In addition to this, she discussed core decisions and tradeoffs that must be made when moving from batch to streaming. Arora was clear to stress that "batch is not dead", and although there are many stream-processing engines, there is no single best solution.

Netflix's core mission is to entertain customers by allowing them to watch personalized video content anywhere at anytime. In the course of providing this personalized experience, Netflix processes 450 billion unique events daily from 100+ million active members in 190 different countries who view 125 million hours of content per day. The Netflix system uses the microservice architectural style and services communicate via remote procedure call (RPC) and messaging. The production system has a large Apache Kafka cluster with 700+ topics deployed that manages messaging and also feeds the data-processing pipeline.

Within Netflix, the Data Engineering and Analytics (DEA) team and Netflix Research are responsible for running the personalization systems. At a high level, microservice application instances emit user and system-driven data events that are collected within the Netflix Keystone data pipeline — a petabyte-scale real-time event streaming-processing system for business and product analytics. Traditional batch data processing is conducted by storing this data within a Hadoop Distributed File System (HDFS) running on the Amazon S3 object storage service and processing with Apache Spark, Pig, Hive, or Hadoop. Batch-processed data is stored within tables or indexers like Elasticsearch for consumption by the research team, downstream systems, or dashboard applications. Stream processing is also conducted by using Apache Kafka to stream data into Apache Flink or Spark Streaming.

Before discussing her team's decision to convert a long-running batch ETL job into a streaming process, Arora cautioned the audience against "streaming all the things". There are clear business wins for using stream processing, including the opportunity to train machine-learning algorithms with the latest data, provide innovation in the marketing of new launches, and create opportunities for new kinds of machine-learning algorithms. There are also technical wins, such as the ability to save on storage costs (as raw data does not need to be stored in its original form), faster turnaround time on error correction (long-running batch jobs can incur significant delays when they fail), real-time auditing on key personalization metrics, and integration with other real-time systems.

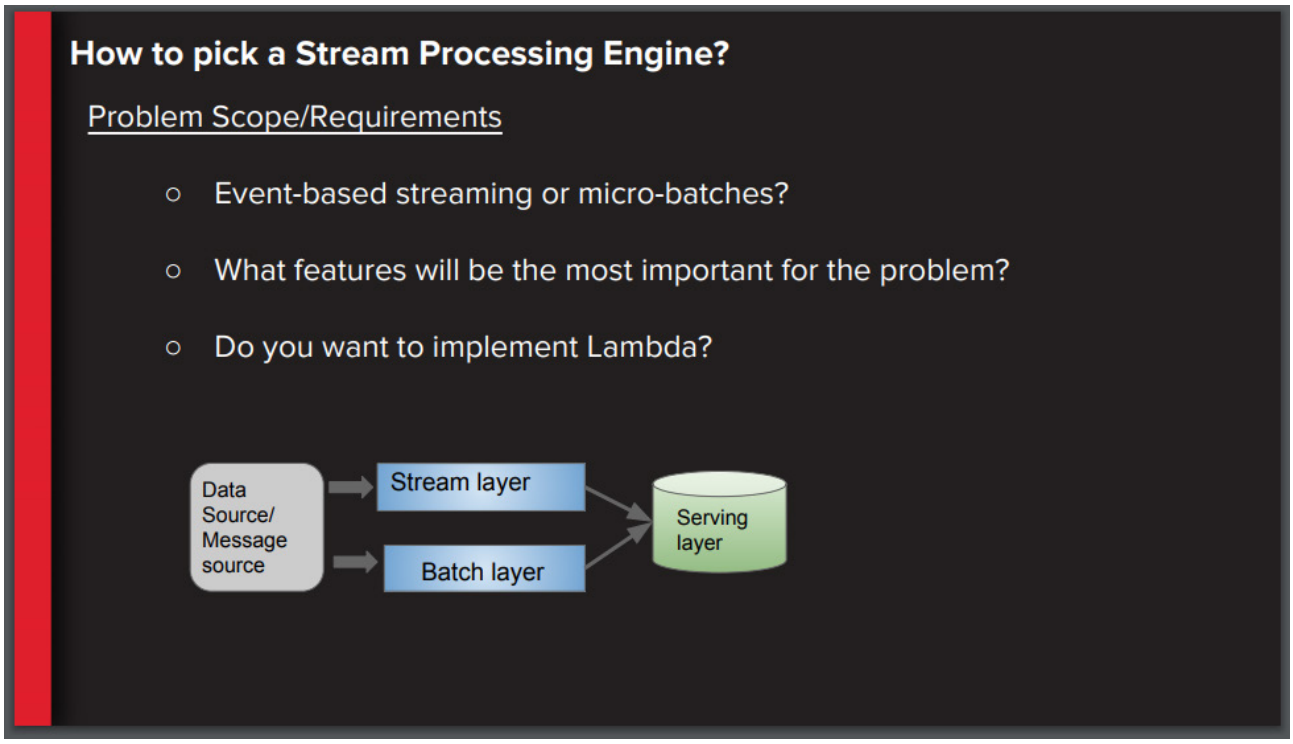
A core challenge when implementing stream processing is



picking an appropriate engine. The first key question to ask is will the data be processed as an event-based stream or in micro-batches. In Arora's opinion, micro-batching is really just a subset of batch processing — one with a time window that may be reduced from a day in typical batch processing to hours or minutes — but a process still operating on a corpus of data rather than actual events. If results are simply required sooner than currently provided, and the organization has already invested heavily in batch, then migrating to micro-batching could be the most appropriate and cost-effective solution.

The next challenge in picking a stream-processing engine is to ask what features will be most important in order to solve the problem being tackled. This will most likely not be an issue that is solved in an initial brainstorming session — often a deep understanding of the problem and data only emerge after an in-depth investigation. Arora's case study required "sessionization" (session-based windowing) of event data. Each engine supports this feature to varying degrees with varying mechanisms. Ultimately, Netflix chose Apache Flink for Arora's batch-job migration as it provided excellent support for customization of windowing in comparison with Spark Streaming (although it is worth mentioning that new APIs supporting Spark Structured Streaming and advanced session handling have become stable as of Apache Spark 2.2.0, which was released in July 2017, after this presentation was delivered).

Another question to ask is whether the implementation requires the lambda architecture. This architecture is not to be confused with AWS Lambda or serverless technology in general — in the data-processing domain, the lambda architecture is designed to handle massive quantities of data by taking advantage of both batch-processing and stream-processing methods. This approach to architecture attempts to balance latency, throughput, and fault-tolerance by creating a batch layer that provides a comprehensive and accurate "correct" view of batch data, while simultaneously implementing a speed layer for real-time stream processing to provide potentially incomplete, but timely, views of online data. It may be the case that an existing batch job simply needs to be augmented with a speed layer, and if this is the case then choosing a data-processing engine that supports both layers of the lambda architecture may facilitate code reuse.



Several additional questions to ask when choosing a stream-processing engine include:

- What are other teams using within your organization? If there is a significant investment in a specific technology, then existing implementation and operational knowledge can often be leveraged.
- What is the landscape of the existing ETL systems within your organization? Will a new technology easily fit in with existing sources and sinks?
- What are your requirements for learning curve? What engines do you use for batch processing, and what are the most widely adopted programming languages?

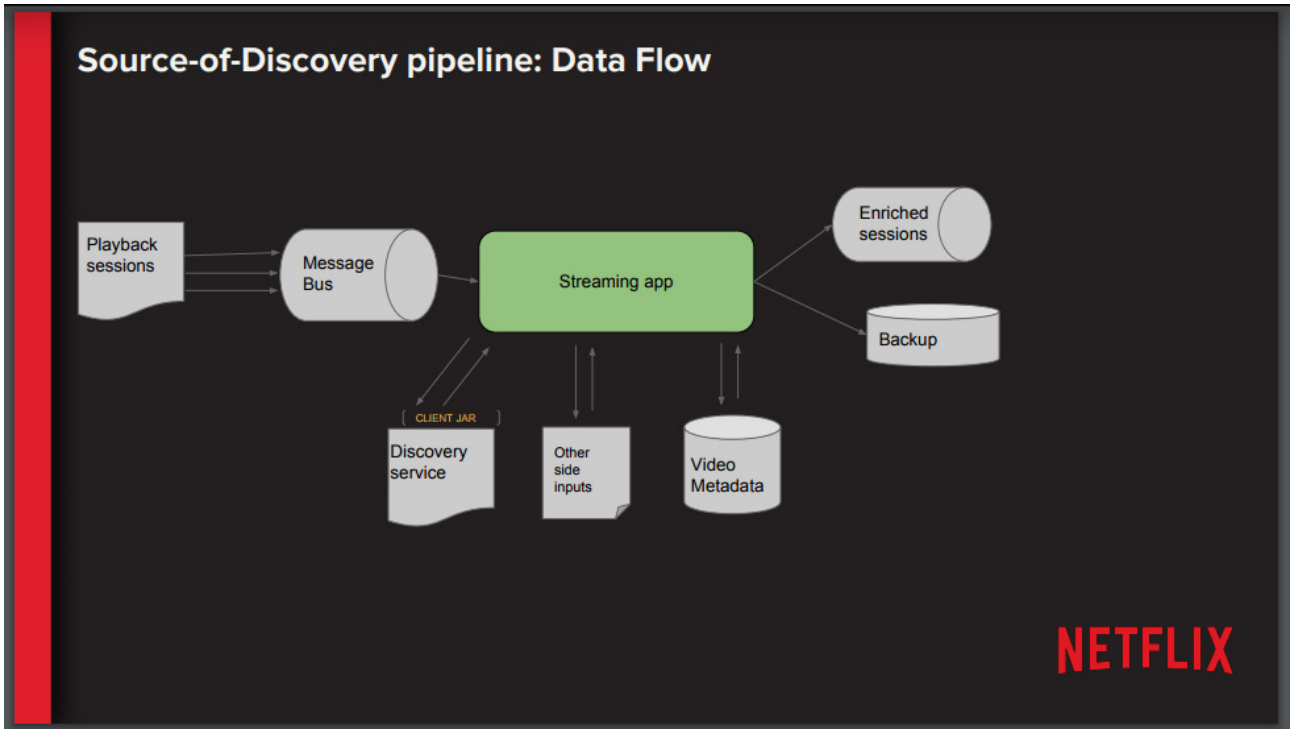
The penultimate section of the talk examined the migration of a Netflix batch ETL job to a stream-processing ETL process. The Netflix DEA team previously analyzed sources of play and sources of discovery within the Netflix application using a batch-style ETL job that can take longer than eight hours to complete. Sources of play are the locations from the Netflix application homepage from which users initiate playback. Sources of discovery are the locations on the homepage where users discover new content to watch. The ultimate goal of the DEA team was to learn how to optimize the homepage to maximize discovery of content and playback for users, and to improve the overly long 24-hour latency between occurring events and analysis. Real-time processing could shorten this gap between action and analysis.

Examining the "source of discovery" problem in more depth revealed to Netflix that the stream-processing engine to choose had to be able to: handle a high throughput of data (users across the globe currently generate ~100 million discovery/playback events per day); communicate to live microservices via thick (RPC-style) clients in order to enrich the initial events; integrate with the Netflix platform ecosystem such as, for example, service discovery; have centralized log management and alerting; and allow side inputs of slowly changing data (e.g., a dimension or metadata table containing film metadata or country demographics).

Ultimately, Arora and her team chose Apache Flink with an ensemble cast of supporting technology:

- Apache Kafka acting as a message bus;
- Apache Hive providing data summarization, query, and analysis using an SQL-like interface (particularly for metadata in this case);
- Amazon S3 for storing data within HDFS;
- the Netflix OSS stack for integration into the wider Netflix ecosystem;
- Apache Mesos for job scheduling and execution; and
- Spinnaker for continuous delivery.

An overview of the complete source of discovery pipeline can be seen below.



Arora outlined the implementation challenges that the DEA team faced with the migration process:

Getting data from live sources:

- The job being migrated required access to the complete viewing history of the user of every playback initiation event.
- This was conceptually easy to implement with stream processing, as the integration with the Netflix stack and real-time nature of the data processing meant that a simple RPC-like call was required for each event as it was processed.
- However, because the Apache Flink stream-processing application was written using the Java API and the Netflix OSS stack is also written using Java, it was sometimes challenging to ensure compatibility between libraries within both applications (managing so-called "JAR hell").

Side inputs:

- Each item of metadata required within the stream-processing job could have been obtained by making a call in the same fashion as getting data from live sources.
- However, this would require many network calls, and ultimately be a very inefficient use of resources.
- Instead the metadata was cached into memory for each stream-processing instance, and the data refreshed every 15 minutes.

Data recovery:

- When a batch job fails due to an infrastructure issue, it is easy to rerun the job, as the

There are many decisions and tradeoffs that must be made when moving from batch ETL to stream data processing. Engineers should not "stream all the things" just because stream processing technology is popular.

data is still stored within the underlying object store — i.e., HDFS. This is not necessarily the case with stream processing, as the original events can be discarded as they are processed.

- Within the Netflix ecosystem, the TTLs of the message bus (Kafka) that stores the original events can be relatively aggressive — due to the volume, as little as four to six hours. Accordingly, if a stream-processing job fails and this is not detected and fixed within the TTL time limit, data loss can occur.
- The solution for this issue was to additionally store the raw data in HDFS for a finite time (one to two days) in order to facilitate replay.

Out-of-order events:

- In the event of a pipeline failure, the data-recovery process (and reloading of events) will mean that "old" data will be mixed in with real-time data.
- The challenge is that late-arriving data must be attributed correctly to the event time at which it was generated.
- The DEA team chose to implement time windowing and also post-process data to ensure that the results are emitted with the correct event-time context.

Increased monitoring and alerts:

- In the event of a pipeline failure, the team must be notified as soon as possible.
- Failure to trigger a timely alert can result in data loss.
- Creating an effective monitoring, logging, and alerting implementation is vital.

Arora concluded the talk by stating that although the business and technical wins for migrating from batch ETL to stream processing were numerous, there were also many challenges and learning experiences. Engineers adopting stream processing should be prepared to pay a pioneer tax, as most conventional ETL is batch and training machine-learning models on streaming data is relatively new ground. The data processing team will also be exposed to high-priority operational issues — such as being on call and handling outages — as although "batch failures have to be addressed urgently, streaming failures have to be addressed immediately". An investment in resilient infrastructure must be made, and the team should also cultivate effective monitoring and alerting, and create continuous-delivery pipelines that facilitate the rapid iteration and deployment of the data-processing application.

The full video of Arora's QCon New York 2017 talk "Personalizing Netflix with Streaming Datasets" can be found on InfoQ.

View Full Presentation

KEY TAKEAWAYS

The Goldman Sachs Core Front Office Platform team run an on-premise Apache Kafka cluster on a virtualised on-premise infrastructure that handles ~1.5 Tb a week of traffic

The team has invested significant resources into preventing data loss, and with data centers in the same (or very close) metro area, the multiple centers can effectively be treated as a single redundant data center for disaster recovery and business continuity (DRBC) purposes

The Core Front Office Platform team have invested significantly in creating tooling to support their infrastructure, including a REST service to provide insight into the Kafka cluster, and the creation of a comprehensive metrics capture component

Failure will occur, and engineers must plan to handle this. The approach that has been adopted at GS is to run everything with high-availability, and be transparent in all of the trade-offs made

WHEN STREAMS FAIL: IMPLEMENTING A RESILIENT APACHE KAFKA CLUSTER AT GOLDMAN SACHS

At QCon New York 2017, Anton Gorshkov presented “When Streams Fail: Kafka Off the Shore”. He shared insight into how a platform team at a large financial institution designs and operates shared internal messaging clusters like Apache Kafka, and also they plan for and resolve the inevitable failures that occur.

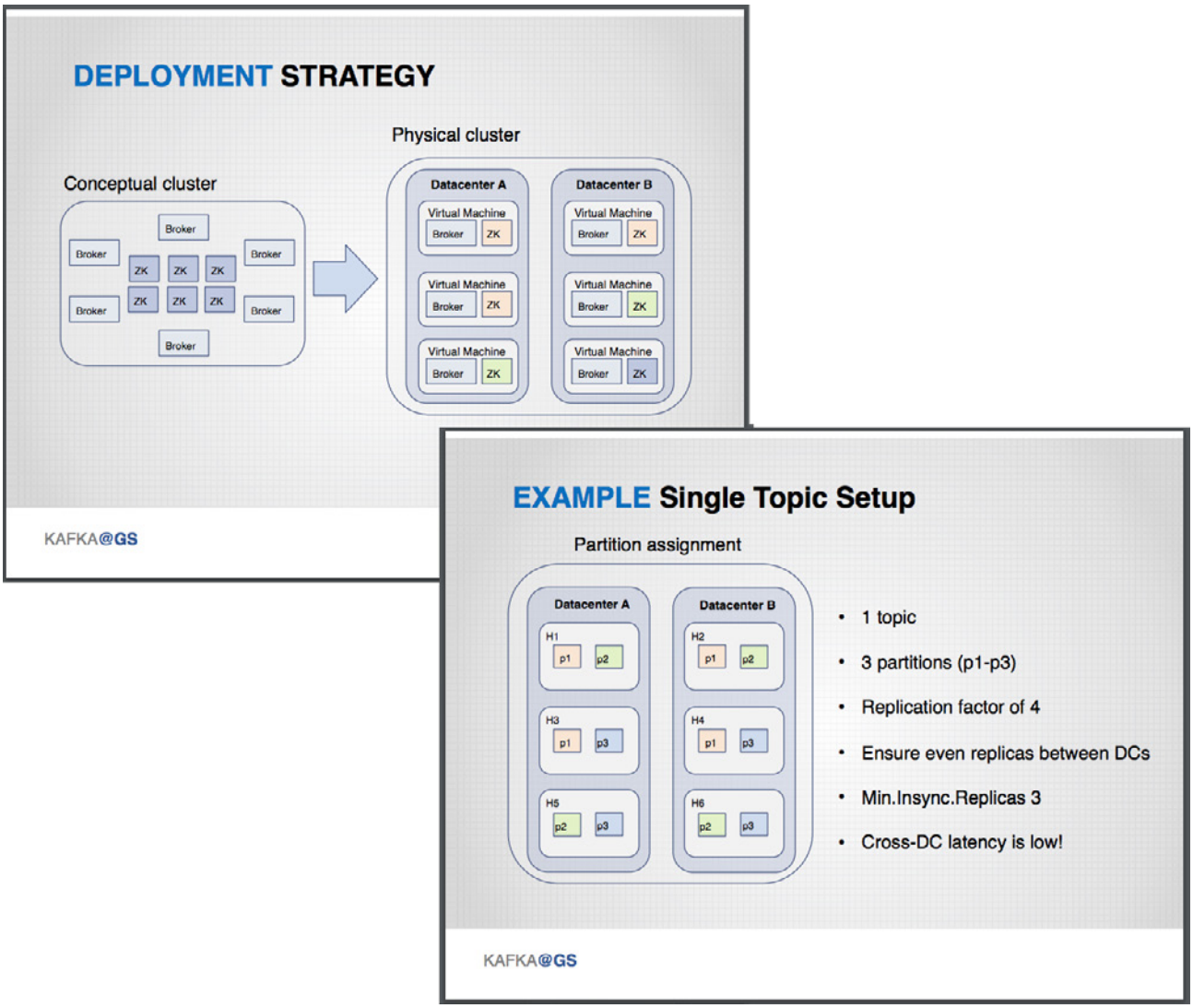
Gorshkov, managing director at Goldman Sachs, began by introducing Goldman Sachs and discussing the stream-processing workloads his division manages. The company's Investment Management Division has $1.4 trillion in assets under management, and the Core Platform team interfaces with many other internal teams to provide platforms and infrastructure to run Apache Kafka, Data Fabric, and Akka. The team operates an on-premise Apache Kafka cluster running on virtualized infrastructure that handles ~1.5 TB a week of traffic, and although the message count is relatively low — in the order of millions per week — at peak periods, Kafka can see about 1,500 messages produced per second.

The deployment goals of the Apache Kafka cluster are:

- no data loss, even in the event of a data-center outage;
- no notion of primary/backup;
- no failover scenarios; and
- to minimize outage time.

The team has invested significant resources into preventing fundamental data loss, and this includes providing tape backup, nightly batch replication, asynchronous replication, and synchronous replication (e.g., synchronous disk-level replication with Symmetrix Remote Data Facility). Gorshkov reminded the audience of latency numbers that every programmer should know, and stated that the speed of light dictates that a best-case network round trip from New York City to San Francisco takes ~60ms, Virginia to Ohio takes ~12ms, and New York City to New Jersey takes ~4ms. With data centers in the same metro area or otherwise close, multiple centers can effectively be treated as a single redundant data center for disaster recovery and business continuity. This is much the same approach as taken by modern cloud vendors like AWS, with infrastructure being divided into geographic regions, and regions being further divided into availability zones.

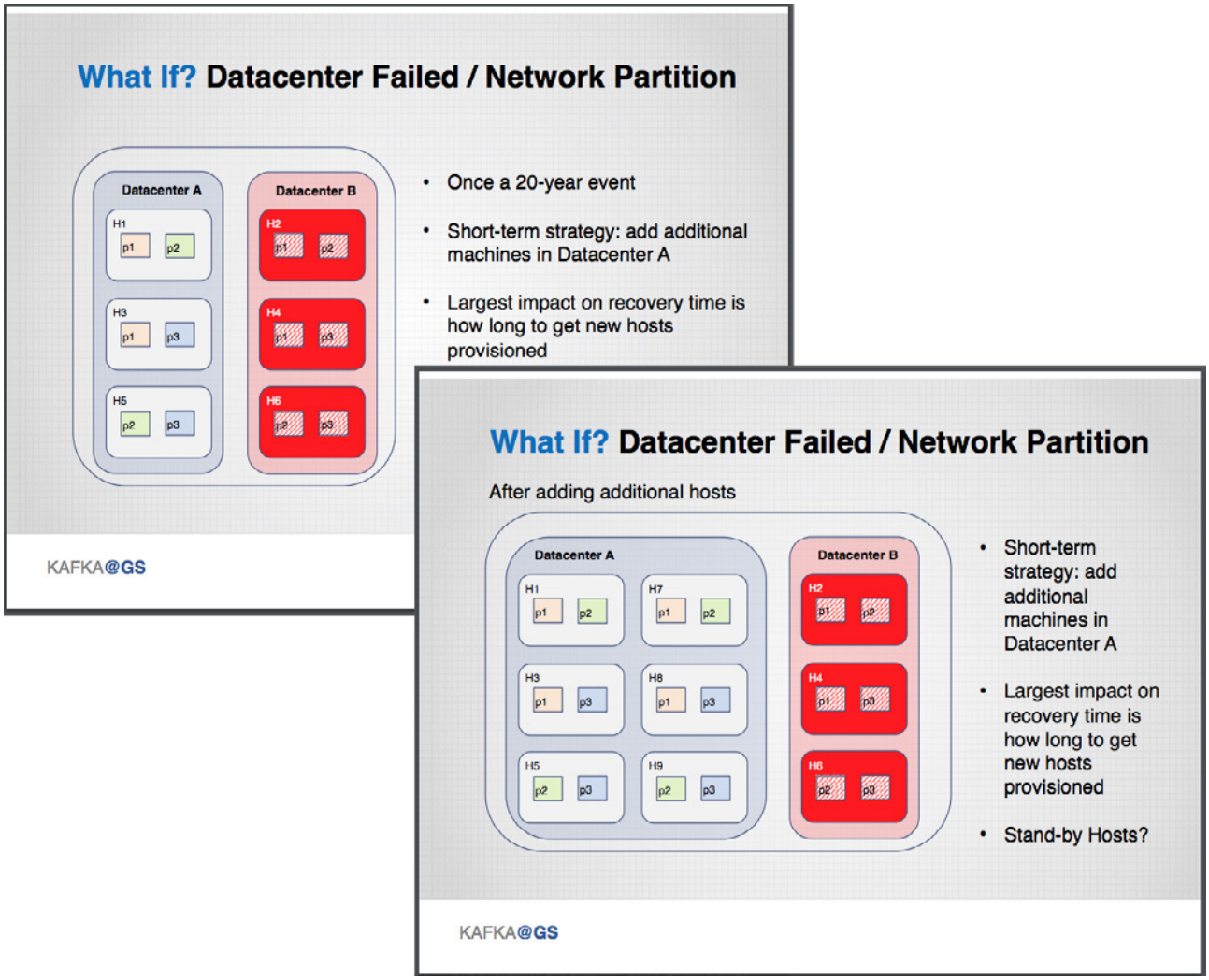
Allowing multiple data centers to be treated as one leads to an Apache Kafka cluster deployment strategy as shown on the diagram below, with a single conceptual cluster that spans multiple physical data centers.

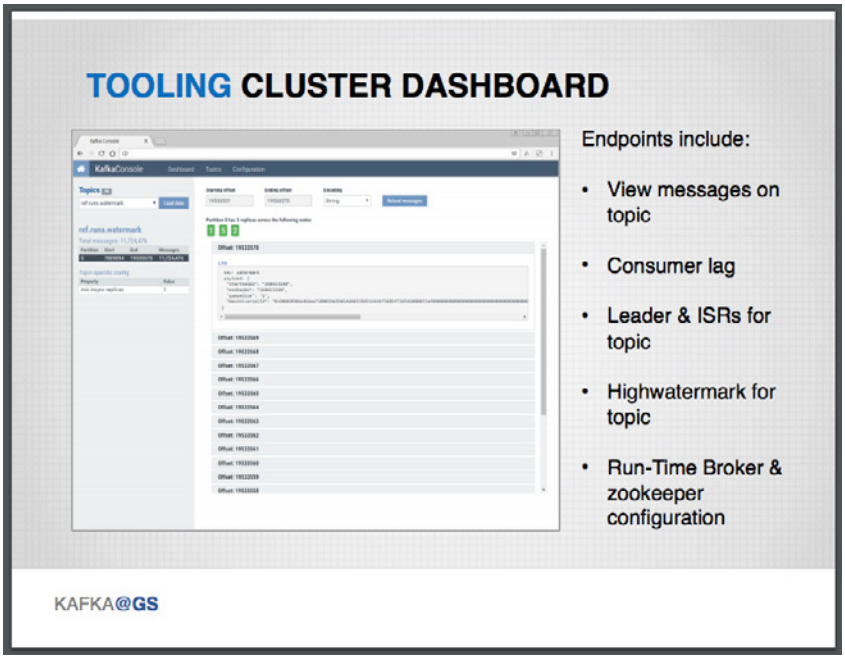


Gorshkov ran through a series of failure scenarios, starting with an exploration of what happens if a single virtual-machine (VM) host fails within a data center. This generally happens one to five times a year yet has no impact on Kafka producers or consumers, as the system is still able to satisfy the minimum required synchronization of at least three replicas. In this case, there is no manual recovery, beyond replacing the host. The occurrence of two hosts failing simultaneously occurs once a year or potentially more often if there is an underlying infrastructure or hypervisor failure. If this failure mode occurs, the processing for some Kafka topics will halt. The short-term fix is to add replicas for the affected partitions, and ultimately to replace the bad hosts. The Goldman Sachs compute infrastructure allows seamless VM replacement with no need to update DNS aliases or change Kafka configuration.

If three hosts within a data center fail then cluster processing immediately halts as this configuration can no longer satisfy the required number of in-sync replicas across the cluster. Fortunately, this only occurs once every few years. The fix is to replace the host as soon as possible. If a data center fails or a network partition occurs — which Gorshkov estimates is a "once a 20-year event" — then the short-term solution is to add additional hosts in the data center that is not affected. The largest impact on recovery time is how long it takes to provision new hosts, as data centers typically maintain spare capacity.

The Core Platform team has invested significantly in creating tooling to support their infrastructure, including a REST-like service and associated web application to provide insight into the Kafka cluster. The REST endpoints allow messages to be viewed on all topics, and core metrics like consumer lag and the number of in-sync replicas to be obtained. It is also possible to obtain information on ZooKeeper configuration, the process of leader election, and run-time broker metrics. The platform team has also created a component that records a multitude of metrics from the operation of the cluster at the application, JVM, and infrastructure levels. Metrics are sent to a





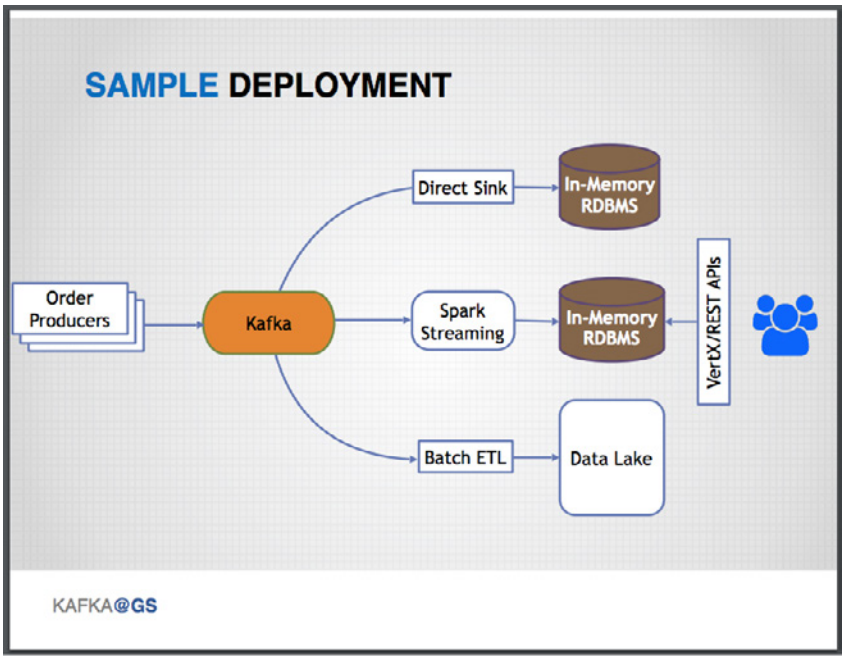
time-series database and are forwarded to a centrally managed Goldman Sachs alerting infrastructure. From here, alerts can be issued to on-call engineers.

A typical sample deployment includes an upstream service — e.g., a trade-orders service — acting as a message source and sending events based on an internal state change (which is also captured in a data store local to the service) to the Apache Kafka cluster. The Kafka Connect API is used to connect message sinks to the Kafka cluster, and downstream targets typically include a direct sink to an in-memory RDBMS that maintains a tabular version of all messages for troubleshooting purposes, a Spark Streaming job that outputs results to an in-memory RDBMS that is queried by end users via the associated Vert.x or REST APIs, and a batch ETL job that persists all events to a data lake for audit/governance purposes.

If a significant outage does occur and messages need to be resent, then the globally unique identifier that is added to every message by the upstream service makes this relatively easy to replay without processing duplicate messages or breaking idempotency guarantees. If the upstream system did not generate unique identifiers, then Gorshkov recommends exploring the new exactly-once processing semantics introduced to Apache Kafka by the Confluent team, and also researching into Kafka Improvement Proposal (KIP) "KIP-98 - Exactly Once Delivery and Transactional Messaging".

In the final section of the talk, Gorshkov stated that failure will always occur and that engineers must plan to handle this. The approach that his team has adopted is "belt and suspenders" for everything. Ultimately, a lot of the tradeoffs that are encountered for setting up resilient systems involves throughput versus reliability (versus cost). Apache Kafka has many configuration options — perhaps too many — and it can be best to hide some of the knobs from end users. For more details on configuring Kafka to run effectively, Gorshkov recommended the Confluent online talk series, "Best Practices for Apache Kafka in Production" by Gwen Shapira. He concluded the talk by stating the resilience must be implemented using a transparent approach, as this is the only way engineers will gain confidence in the system.

The video from Gorshkov's QCon NY talk "When Streams Fail: Kafka Off the Shore" can be found on InfoQ.



PREVIOUS ISSUES

56

Faster, Smarter DevOps

This DevOps eMag has a broader setting than previous editions. You might, rightfully, ask "what does faster, smarter DevOps mean?". Put simply, any and all approaches to DevOps adoption that uncover important mechanisms or thought processes that might otherwise get submerged by the more straightforward (but equally important) automation and tooling aspects.

Cloud Native

55

In this eMag, the InfoQ team pulled together stories that best help you understand this cloud-native revolution, and what it takes to jump in. It features interviews with industry experts, and articles on key topics like migration, data, and security.

Reactive JavaScript

54

This eMag is meant to give an easy-going, yet varied introduction to reactive programming with JavaScript. Modern web frameworks and numerous libraries have all embraced reactive programming. The rise in immutability and functional reactive programming have added to the discussion. It's important for modern JavaScript developers to know what's going on, even if they're not using it themselves.

Serverless Computing

53

In this InfoQ eMag, we curated some of the best serverless content into a single asset to give you a relevant, pragmatic look at this emerging space.