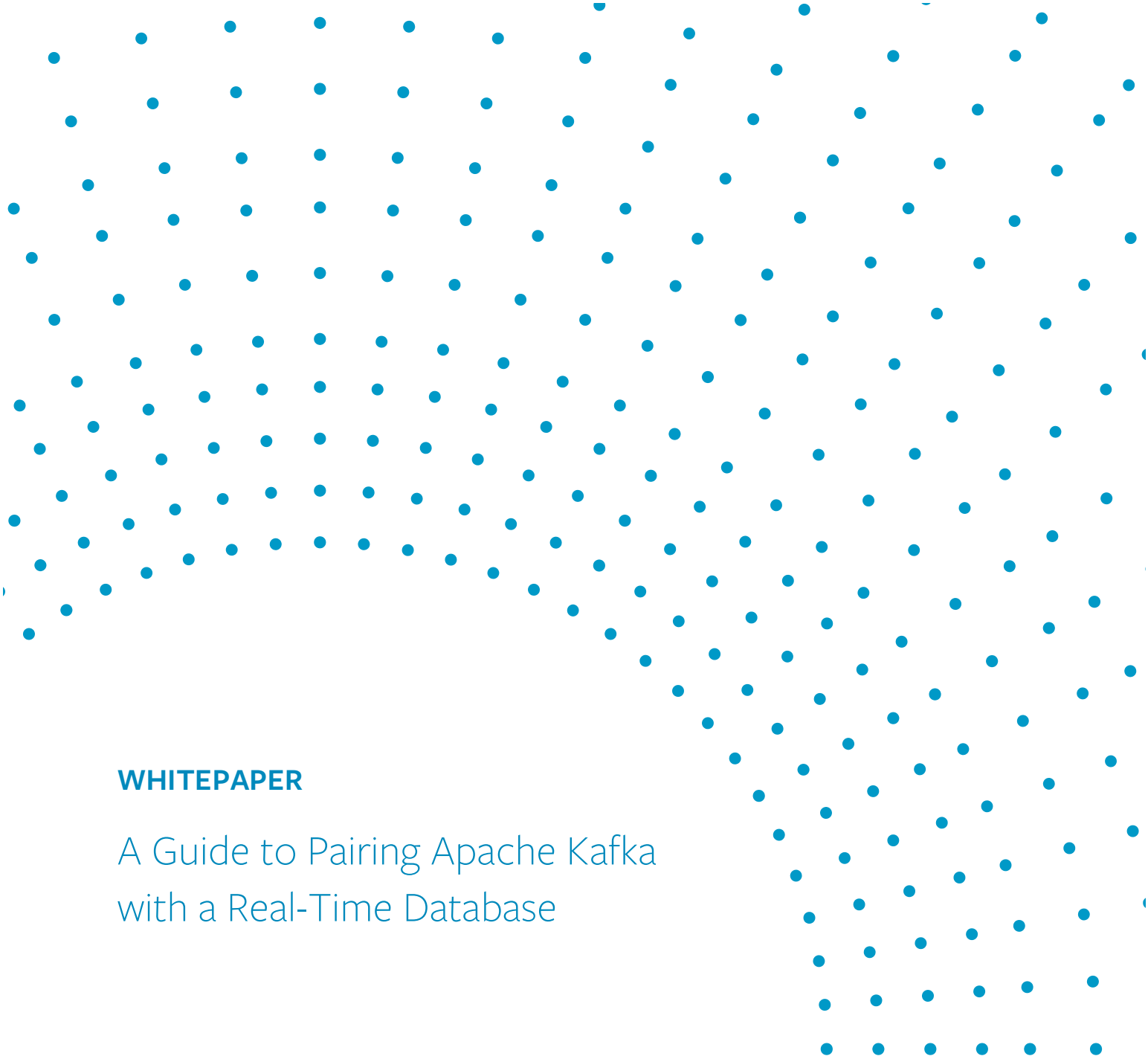


WHITEPAPER

A Guide to Pairing Apache Kafka with a Real-Time Database

MEMSQL

Introduction

The world is abuzz with Digital Transformation and at the heart of every connected device, all the way to enterprise applications, is a message. These messages fuel our digital evolution.

Rise of the Message Queue

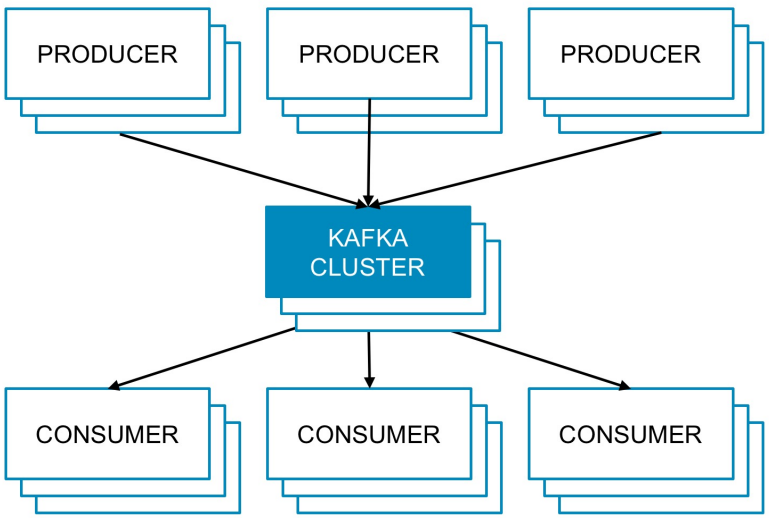
To keep pace with all of the inputs and outputs of digital messages, message queues have evolved. In the last few years, Apache Kafka has come to dominate the message queue landscape.

There are certainly other useful message queues like RabbitMQ, ZeroMQ, and of course AWS Kinesis. Some of the principles in this paper apply to other queues but our focus remains on Kafka.

Kafka basics include the ingest of data from one or multiple Producers, and the consumption of that data from one or multiple Consumers. Kafka acts as a central resource to manage the flow of messages between these end points.

Additionally, Apache Kafka is a distributed system, allowing it to scale horizontally to accommodate greater loads. Data growth combined with demand for real-time results let the distributed architecture of Kafka shine.

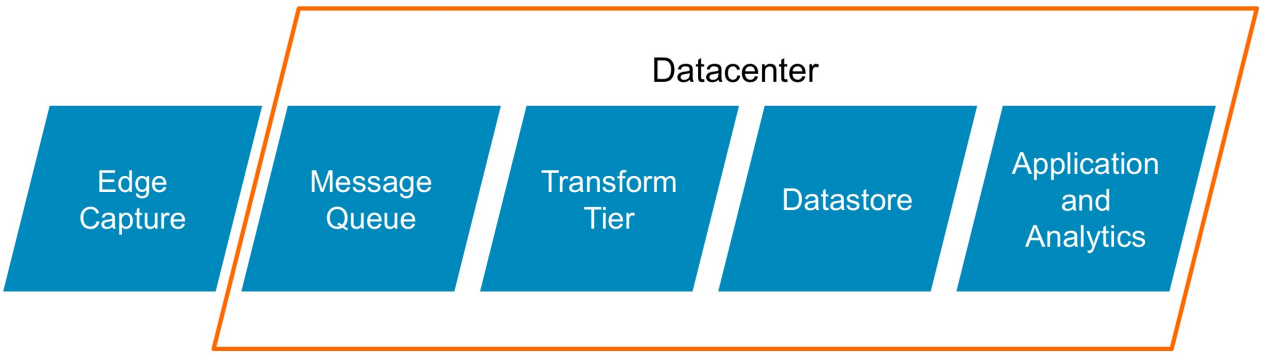
Kafka Inputs and Outputs



The Real-Time Infrastructure Data Pipeline

Beyond the message queue, a broader infrastructure exists for building real-time data pipelines all the way to applications and analytics.

Real-Time Infrastructure Data Pipeline



Edge Capture

This may be from mobile devices, factory automation equipment, or sensors located across the globe. Any type of data collection from users or machines is represented here. Now we shift to the data center.

Message Queue

The message queue, such as Apache Kafka, collects and distributes messages.

Transformation Tier

For data streams requiring persistence, a transformation tier ensures the right formatting takes place to store the data. This tier may also incorporate data cleansing, enrichment, or scoring based on machine learning models.

Datastore

A long term datastore, such as a database, allows for both real-time and historical data to be analyzed concurrently.

Application and Analytics

New data brings value when incorporated into applications and analytics.

Challenges with Real-Time Data Movement

Data movement has long been the process that everyone loves to hate. You need to do it, but nothing about moving data in large volumes or at high speed has been fun.

The crux is handling a large amount of rapidly changing data, sourced from a scaleable system, and the need to get that into a queryable database.

The scalable system could be a queue like Kafka or Kinesis, or it could be a large object or file store like AWS S3 or the Hadoop Distributed File System (HDFS).

The problem is that you cannot easily query the stream, and in the cases where some querying is available, it is generally relegated to a small window of data.

Timely and accurate analytics requires access to real-time data and full historical data, which requires loading the changing data into the database in real time.

This real-time transfer of data between the data source and database can be tricky because they typically have different sharding schemas. For example, the files in S3 might be sharded by time, but the database might be sharded by user ID. So queries will not be efficient if data is loaded straight in, as the time-based shard and user ID shard do not match.

Initial Approaches to Real-Time Data Movement

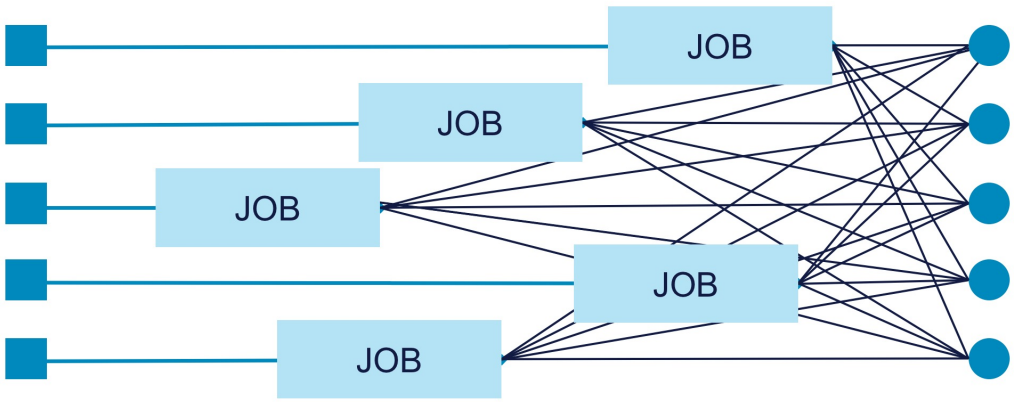
The need to move from one distributed system (the message queue) to another distributed system (the database), initially required yet another distributed system to transform data.

This “middleware” distributed system acts as a go between, pulling from the source and running LOAD DATA queries against the database. While simple in concept, it is a difficult approach in real world deployments.

However, in addition to the complexity of handing a range of transformations, it is further challenging to keep track of which files have been loaded or what offsets need to be loaded.

Overall this approach does not meet the most stringent scaling objectives.

Initial Real-Time Data Movement Approach



Consider the squares on the left as the nodes of the middleware solution, and the circles on the right as the database partitions and recognize that the sharding schemas are different.

The middleware solutions present a series of messages, or transactions, and a requirement to keep them separate. All of the jobs inevitably compete. For example, if one job is passing along a transaction to pay Sam $5, and another job is passing a long another transaction to pay Sam $10, the first job will take a lock on the row for Sam, while the 2nd has to wait.

Long story short, the coordination required is untenable at scale.

Databases and Native Pipeline Support

The good news is that with a properly architected distributed database, a middleware system is not required for real-time data movement including transformations. Specifically, a database with native pipelines support moves data in highly scalable transactional micro batches.

The database knows that all of the data movement is effectively one job, and it efficiently manages the operation by sharing resources smartly, including the ability to retain exactly-once semantics. Exactly-once semantics help ensure accuracy for real-time pipelines. For example, when importing data from S3 missing a file is a pain, but loading a file twice can cause much greater annoyance and troubleshooting.

Introducing CREATE PIPELINE

To make creating real-time data pipelines easy, MemSQL added the CREATE PIPELINE command to its SQL syntax. This provides native support for creating and managing data pipelines from capturing a stream through complete data persistence in a queryable datastore.

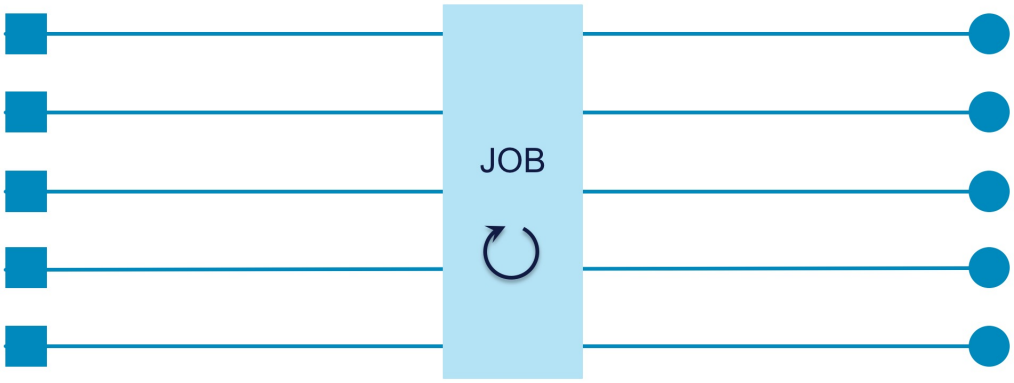
The SQL syntax is as follows

```
CREATE PIPELINE my_pipeline

AS LOAD DATA KAFKA 'host:port/topic'

INTO TABLE my_table
```

In a typical data movement scenario, the coordinating database node will start the job. It knows which files have been loaded, what Kafka partitions are being pulled, and the status off offsets. This node directs an efficient load operation with each database partition pulling from the data source in parallel. Data is moved in micro batches with each micro batch representing once database transaction. As the data is streamed into the database, a coordinated reshuffle operation ensures the data is properly stored.

Compared to the earlier diagram of initial data movement approaches, native pipelines operate as a coordinated job.

Native Pipelines



Native Pipelines Details

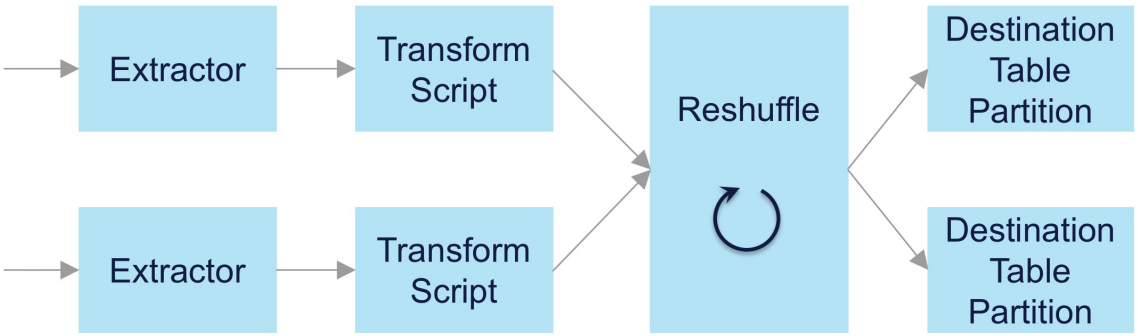
Each database partition has an extractor and is assigned to pull from a particular Kafka partition. These formats can be Avro, CSV, JSON or more, and typically require some type of transformation before storing in the database.

With the offsets stored locally in the database, it is easy to replay Kafka from a previous point, alter the pipeline, and set the offset.

Given the need for some type of transformation, that capability exists with the ability to run a script between the extractor and the database. The transformation can be any script, and simply reads from standard in, and prints to standard out. Then data is reshuffled and inserted into the database.

Of particular note is that this reshuffle operation is extremely efficient given the similar processes run by MemSQL. For example, the same logic that allows MemSQL to conduct distributed joins can be used to manage this reshuffle.

Native Pipelines Detail



Exactly-Once Semantics with Native Pipelines

A primary advantage of collapsing the pipeline is the ability to support exactly-once semantics.

Today most datastores try to accomplish this by letting the message queue keep track, sending an `ack` message back to the queue when a message is received.

This puts a restriction on the database schema designer that the transactions must be idempotent, or more specifically that doing something twice will have the same result as doing it once.

If you consider the example earlier of paying Sam $5 once and then $10 another, the datastore is incrementing counters without necessarily having unique IDs within the message.

Native pipelines do not require the schema designer to consider idempotence as these pipelines use a database transaction to store the latest offset, and this is also the same system storing the data. By knowing the latest offset, the database can insure it is correct, and if there is a question, it can query itself.

This approach provides a more intuitive and streamlined exactly-once semantics, a critical element of real-time data movement for enterprise applications.

Get The Message

According to Gartner in a report published in December 2016,

"Between 2016 and 2019, spending on real-time analytics will grow three times faster than spending on non-real-time analytics."

A critical element of real-time pipelines delivering analytics is the message queue, and the ability to move data quickly and reliably to a persistent database capable of supporting real-time queries.

To take a test drive today, download MemSQL at memsql.com/download.

To discuss your real-time analytics needs with a MemSQL expert, please contact us at 855-4-MEMSQL (463-6775) or info@memsql.com.

MemSQL Headquarters 534 4th Street, San Francisco, CA 94107