THE UNBUNDLED DATABASE

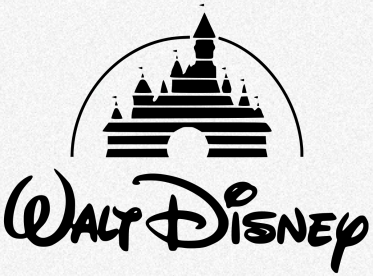
Leveraging the unbundled database via distributed logs and stream processing

O'REILLY®
Software Architecture Conference

Who Am I?

Data Infrastructure at **Pluralsight**

Software and data engineering at **Rackspace Hosting**



Software engineering at **WDPRO**

Pluralsight

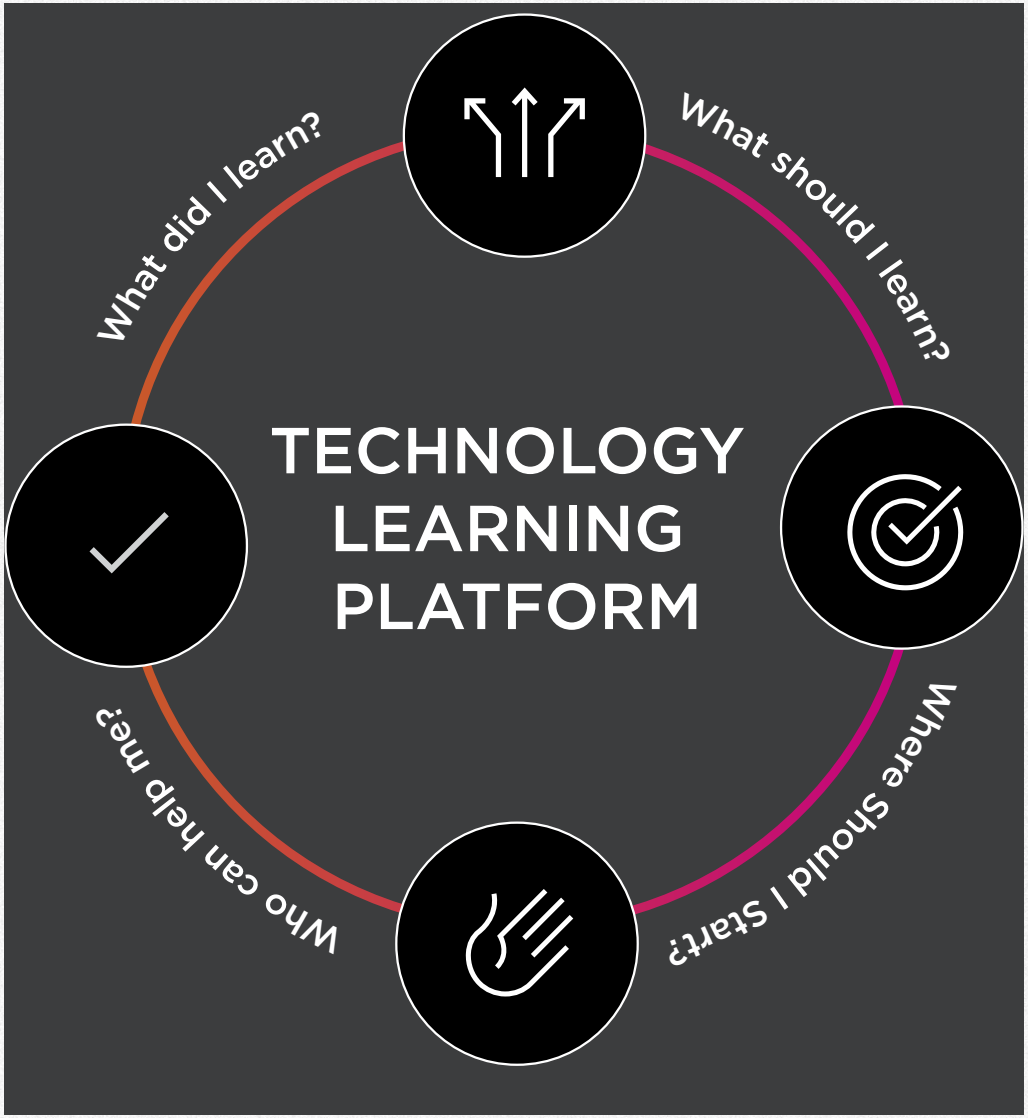


Table of Contents

Microservices

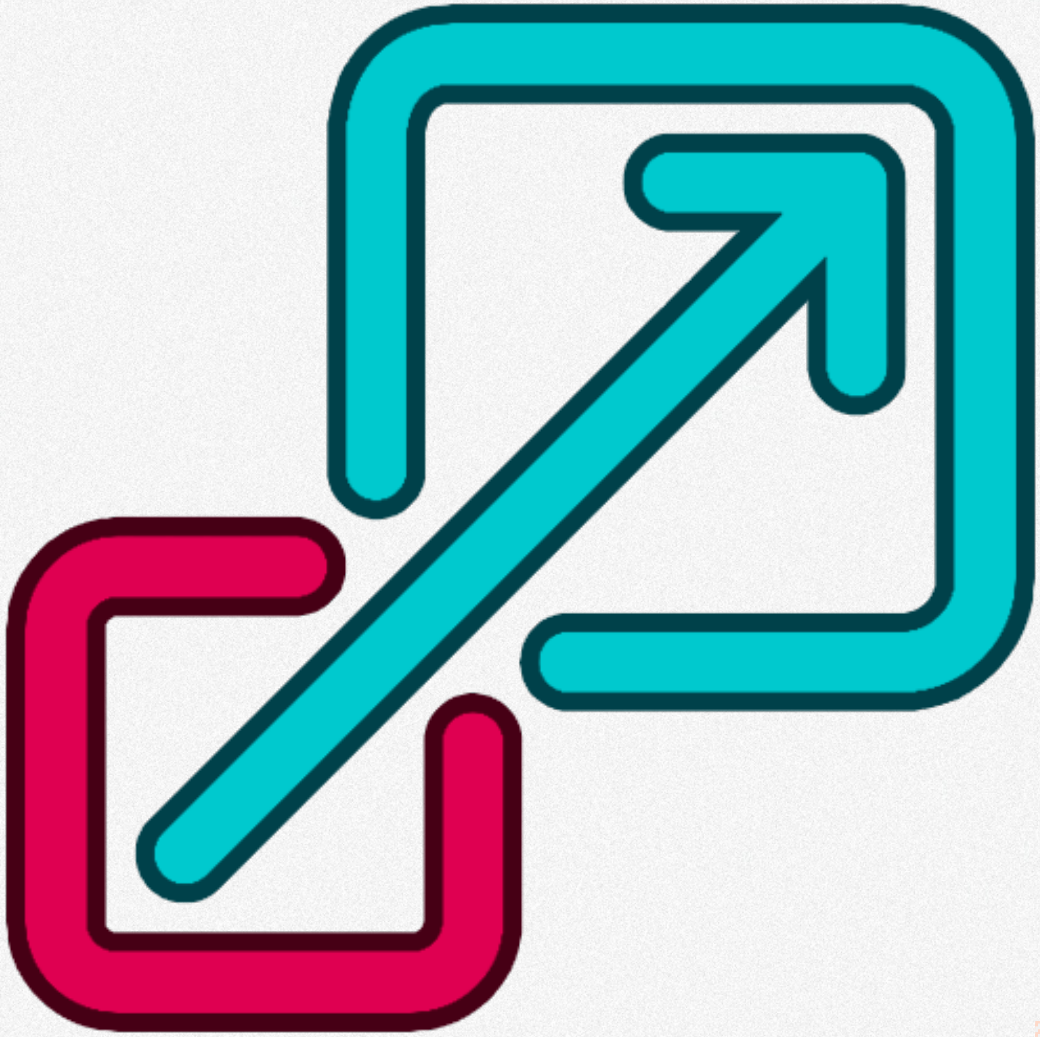
Background

Challenges

Data dichotomy

Streams

Why?



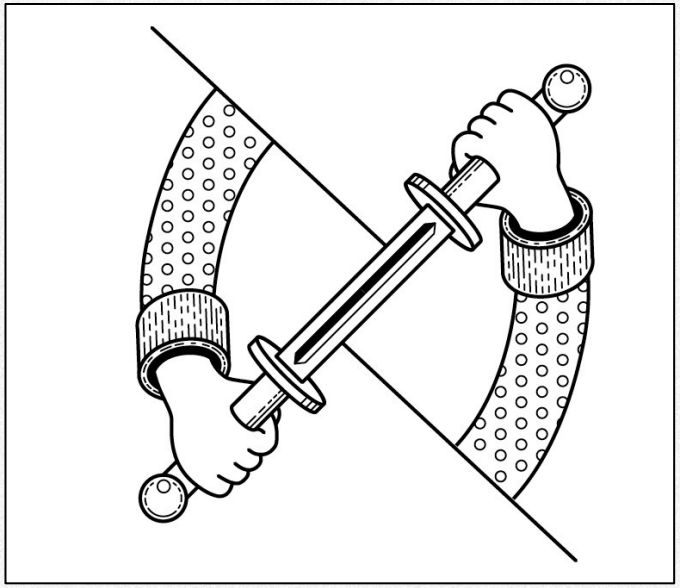
SCALABILITY

Independence comes at a cost

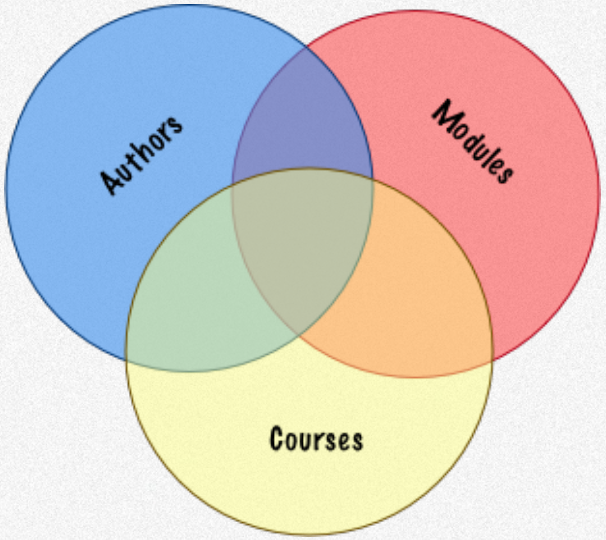
COMPLEXITY

BOUNDARIES

Independence is a double-edged sword.



Services depend on each other



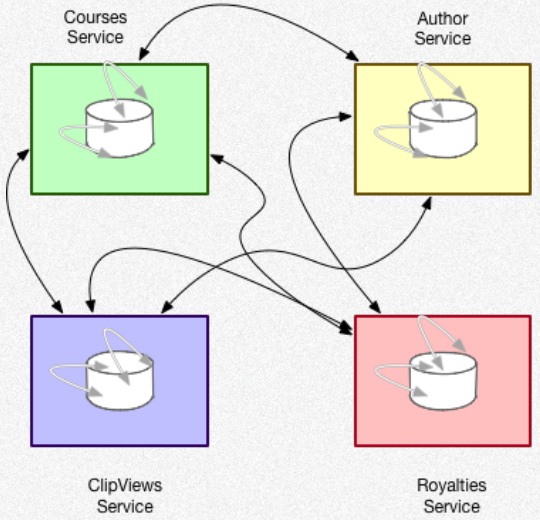
Services are inherently part of a bigger, interconnected ecosystem.

Most business services share the same notions of core facts.

This makes their futures inevitably connected.

Over time, services may become unable to retain the same clear separation of concerns.

Data on the “inside” vs Data on the “outside”



Data on the inside

Encapsulated private data contained within a service.

Data on the outside

Information that flows between independent services.

How do services share data?

Three well-known approaches:

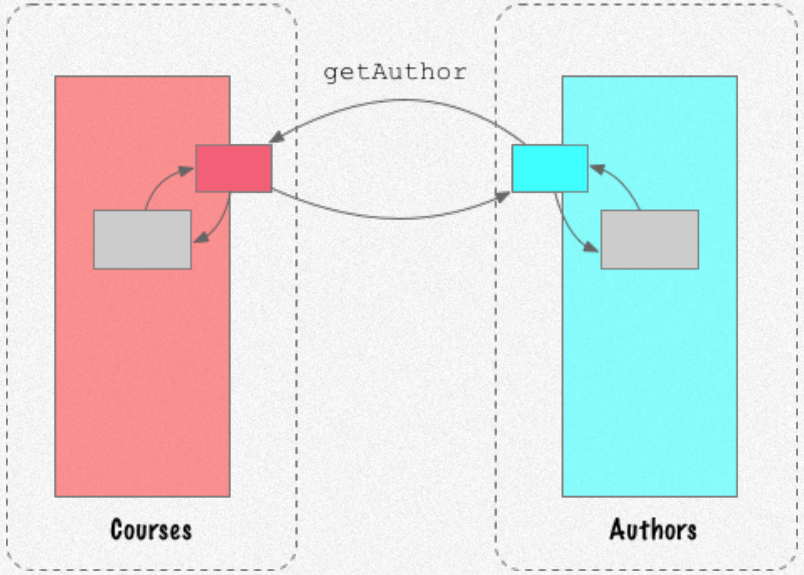
- *Service interfaces*
- *Messaging*
- *Shared databases*

Service Interfaces

Synchronized changes are hard!

Data and functionality are encapsulated in the service.

Goal is to clearly separate concerns between services and define different **bounded contexts.**

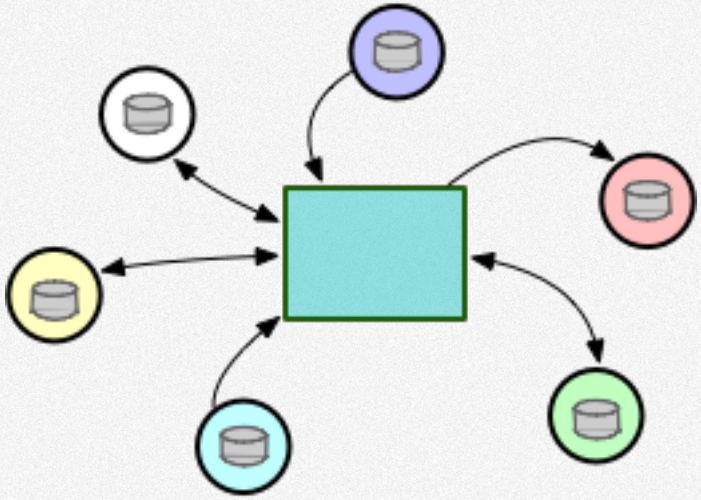


Messaging Middleware

Data and functionality are scattered across organization.

Messaging architectures can scale well.

Even though messaging architectures can move massive amounts of data, they don't provide any historical context, which can lead to data divergence over time.



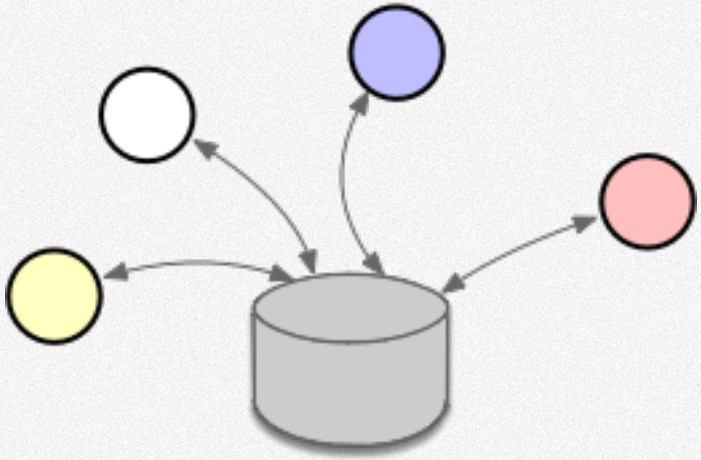
Shared Databases

Functionality is encapsulated within the service; data is not.

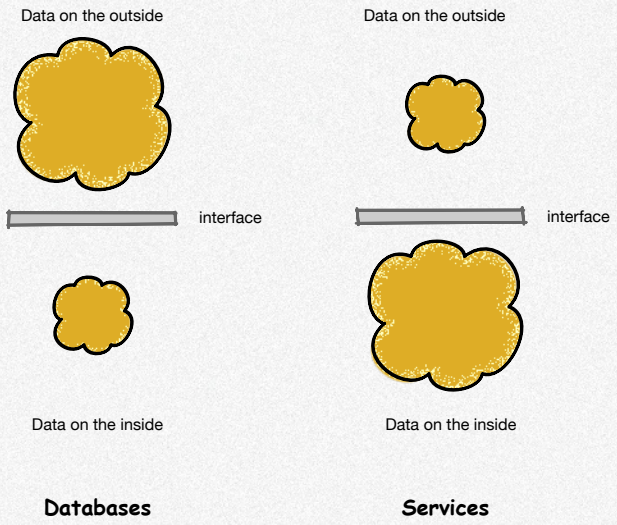
Shared databases concentrate too much data in a single place.

For microservices, databases create an usually strong rich coupling.

This is due to the broad interface that databases expose to the outside world.



Data Dichotomy



Data systems are about exposing data.
Services are about hiding it.

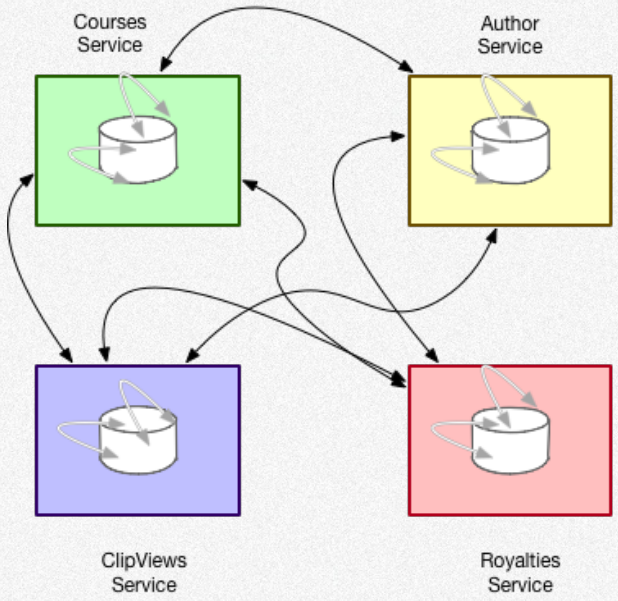
Service interfaces minimize the data they expose to the outside.

Database interfaces, on the other hand, tend to amplify the data they hold.

Data Diverges Overtime

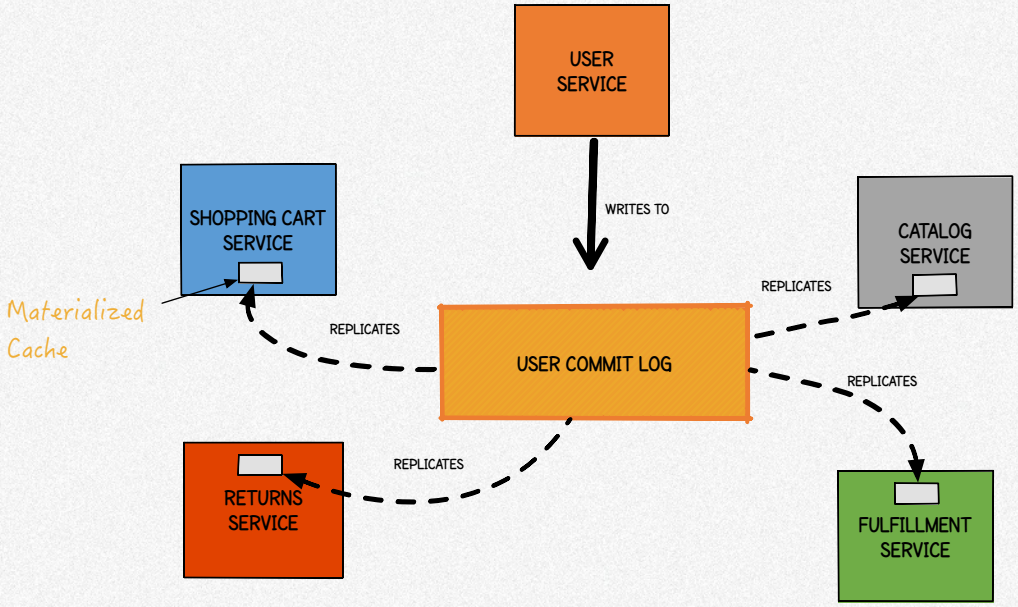
Different services make different interpretations of the data they consume, which leads to divergent information.

Services also keep that data around; data is altered and fixed locally and soon it doesn't represent the original dataset anymore.



Looking ahead: Sharing data with distributed logs

Events are broadcast to a log.



Event Driven Services

Ways services interact

Commands

Commands are actions in the form of side effect generating requests indicating some operation to be performed by another service.

Commands expect a response.

Queries

Requests to look up some data point.

Queries are side effect free and leave the state of the system unchanged.

Events

Events can be thought of are both a fact and a trigger.

They express something that has happened, usually in the form of a notification.

Event Driven Services

I broadcast what I did!

Broadcast events to a centralized, immutable stream of facts.

Downstream freedom to react, adapt and change to any consumer.

There are also some other interesting gains that event driven services provide, such as Exactly Once Processing.

This paradigm is in a departure from request-driven services, where flow resides in commands and queries.

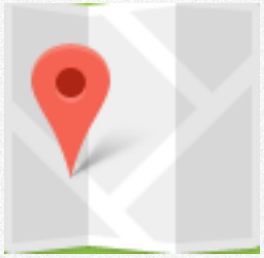
Advantages

State Transfer

Events are both triggers and facts, that can be used to notify and propagate entity state transfers.

Decoupling

Both data producers and consumers are completely decoupled. There is no API binding them together, no synchronized changes that need to be performed.



Locality

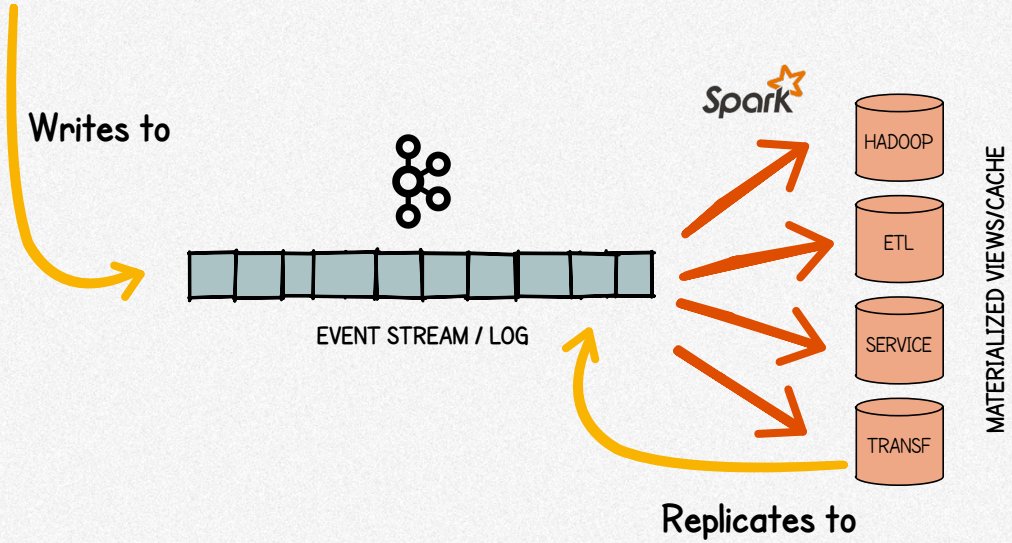
Queries and lookups are local to the bounded context and can be optimized in the best way that fits the current use case.

The Single Writer Principle

A single service owns all events for a single type.

Having a single code path helps with data quality, consistency, and other data sharing concerns.

This is important because these events represent durable shared facts.



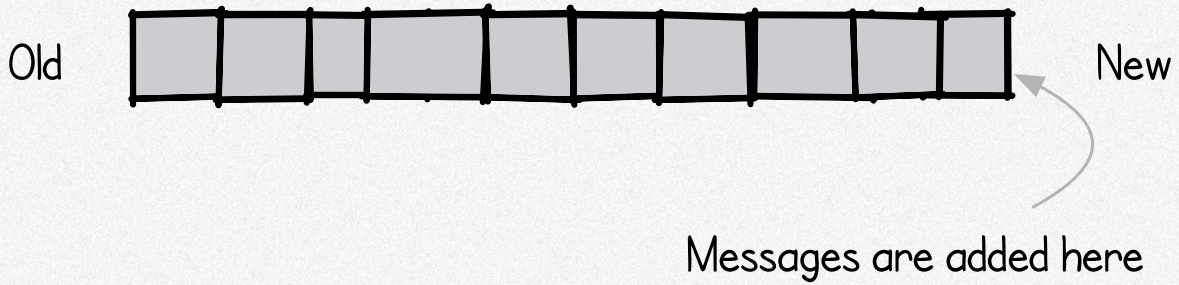
The Distributed Log

Log concepts

Kafka

Topics and partitions

What's a log?

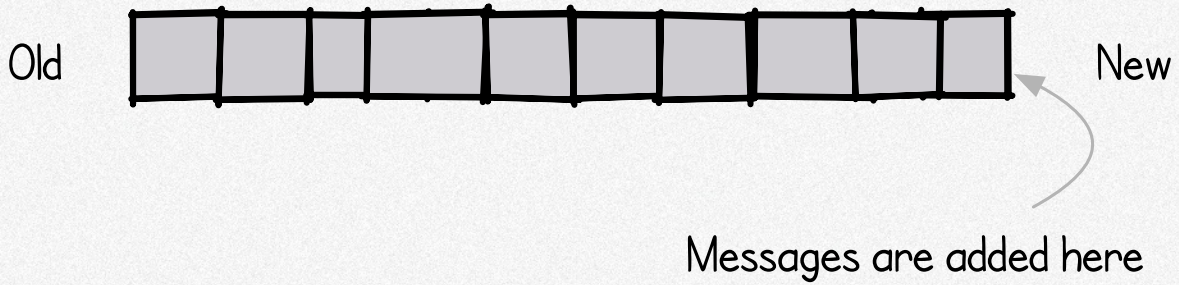


Ordered, immutable sequence of records that is continuously appended to.

Reads and writes are sequential operations.

They are, therefore, sympathetic to the underlying media, leveraging pre-fetch, the various layers of caching and naturally batching similar operations together.

Writing to the Log



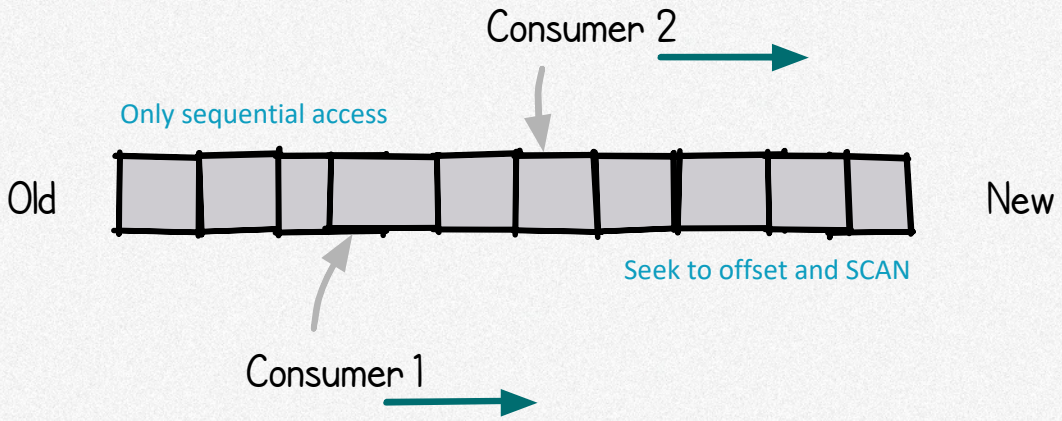
Writes are append only, always added to the head of the log.

Data is stored in the log as stream of bytes.

Due to their structure, logs can be optimized.

For instance, when writing data to Kafka, data is copied directly from the disk buffer to the network buffer, without any memory cache.

Reading from the Log



Reads are performed by seeking to a specific position and sequentially scanning.

Both reads and writes are sequential operations.

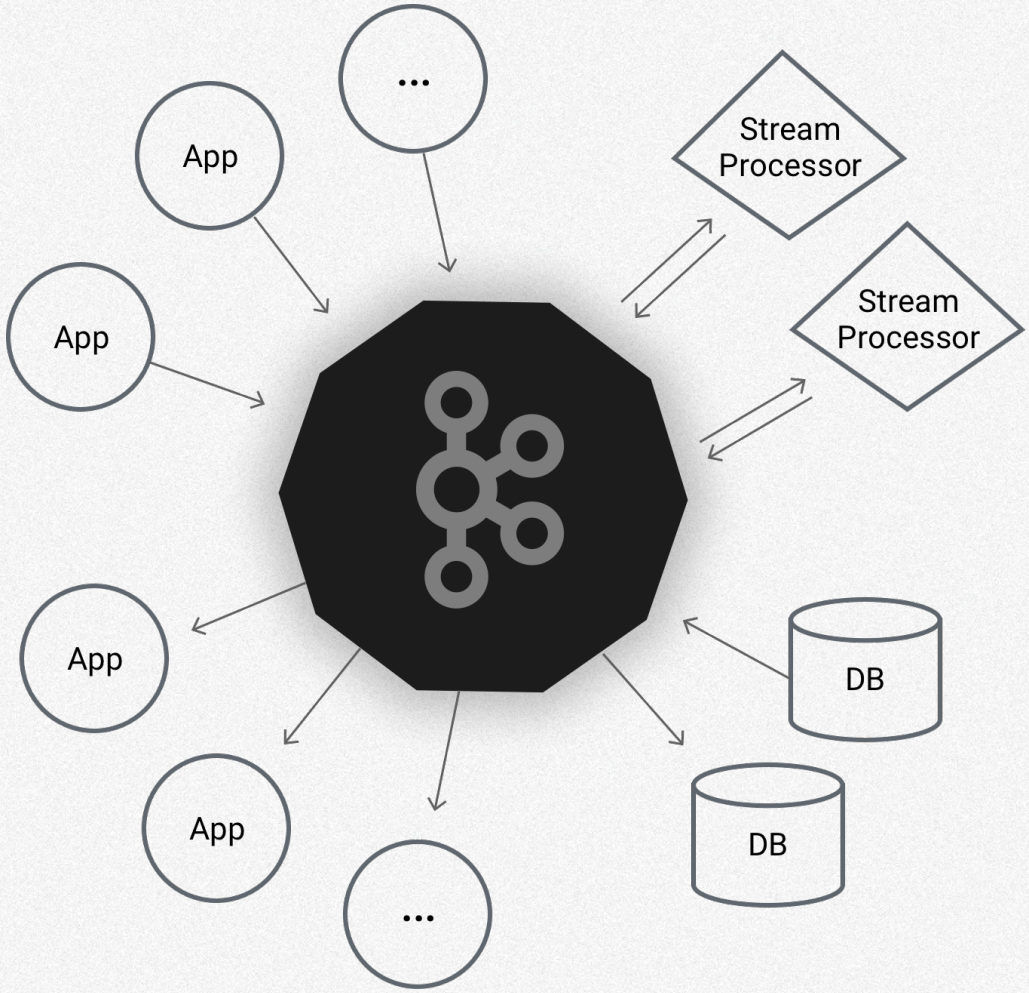
Messages are read in the order they were written.

Consumers are responsible for periodically recording their position in the log.

Since the log is durable, messages can be replayed for as long as they exist in the log.

Kafka

A distributed streaming platform



Key Capabilities

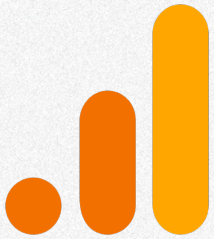
Publish / Subscribe

Kafka is like a message queue or enterprise messaging system, but with some very distinct design concerns and side effects.

Storage

Stores streams of records using replicated, fault-tolerant, durable mechanisms.

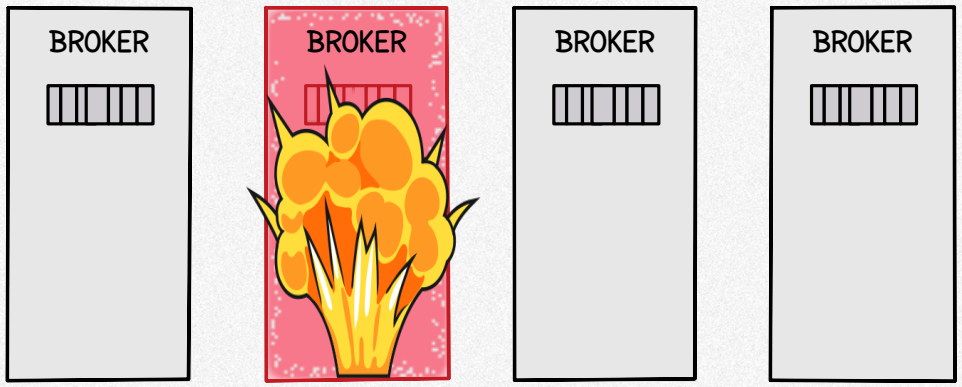
Persists all published records — whether or not they have been consumed, using a configurable retention period.



Processing

Kafka can process and apply logic to streams of records as they occur.

The Kafka Broker



Linearly Scalable

Scaling is a matter of adding more nodes to an existing cluster. Rebalancing, leader election, and replication are automatically adjusted.

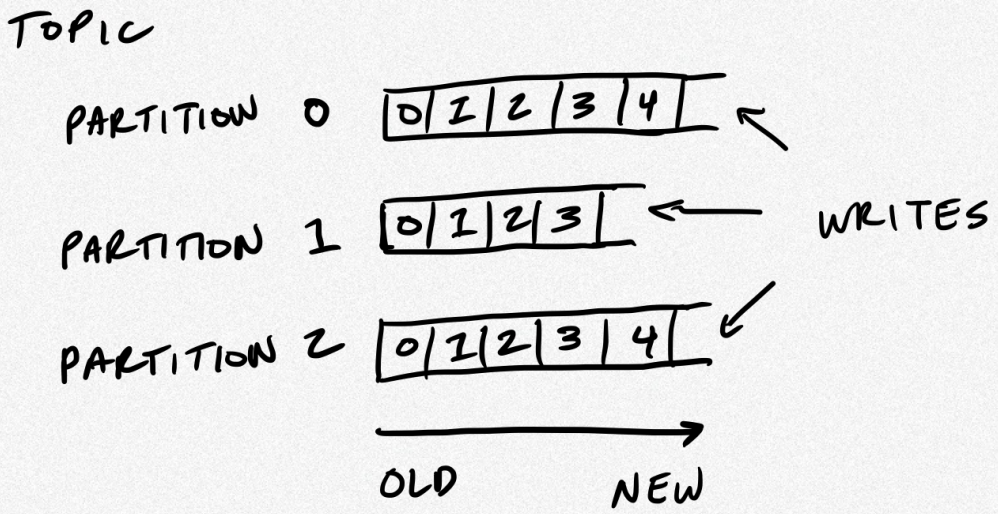
Resilient

Retries, message acknowledgement, ack strategies, are all baked into the platform.

Fault Tolerant

Messages are replicated across different nodes.

Topics and Partitions



Topics are categories or feed names to which records are published to.

Topics

Split into ordered commit logs called partitions.

Data in a topic is retained for a configurable period of time.

Partitions

Each message is assigned a sequential id called an offset.

Allow the logs to scale beyond a size that will fit a single broker.

Act as the unit of parallelism.

Kafka Log Semantics

Ordering guarantees

Message durability

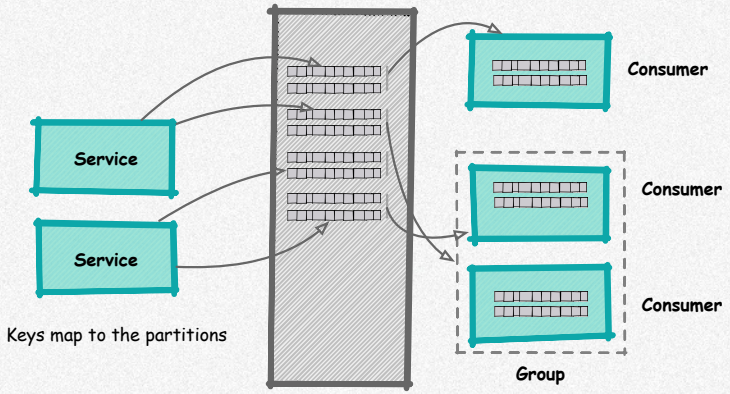
Load balancing

Compaction

Storage

Topic types

Ordering guarantees



Most business systems need strong ordering guarantees.

Consumers in a group are responsible for a single partition, so ordering is guaranteed.

Relative Ordering

Messages that require relative ordering must be sent to the same partition.

Message keys will map the same partition.

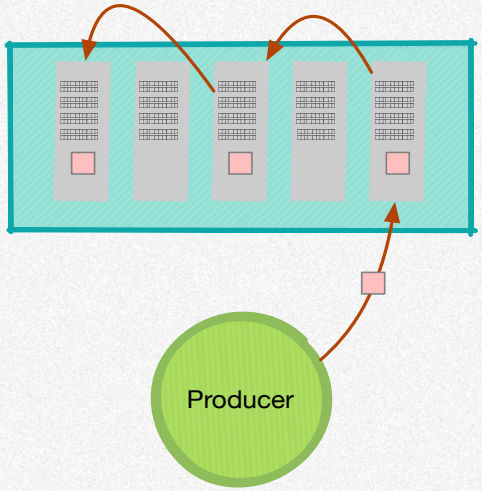
Global Ordering

Requires a single partition topic.

Tends to come up when migrating legacy systems where global ordering was an assumption.

Throughput limited to a single machine.

Message Durability

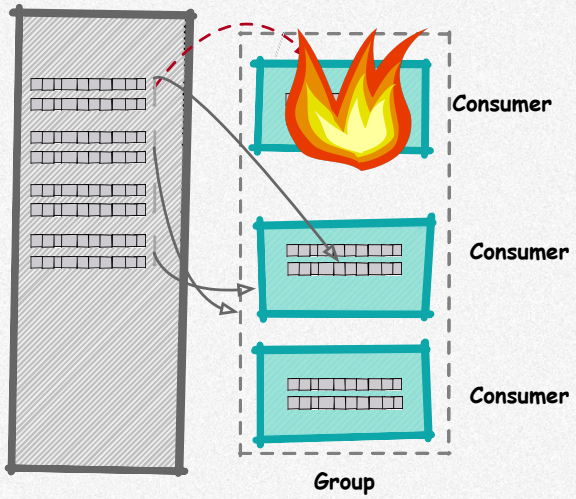


Kafka provides durability through replication.

Messages are written to a leader and then replicated to a user-defined number of brokers.

Records can be configured to be persisted for a period of time or based on keys.

Kafka can load balance services



Load balancing provides high availability

If a consumer leaves a group for any reason, Kafka will detect this change and re-balance how messages are distributed across the remaining consumers.

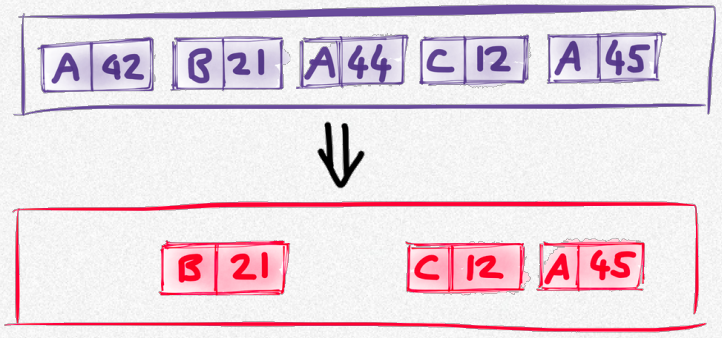
If the failed consumer comes back online, load is balanced again.

Kafka assigns whole partitions to different consumers.

In other words, a single partition can only ever be assigned to a single consumer.

Since this is always true, ordering is guaranteed, across failures and restarts.

Compaction



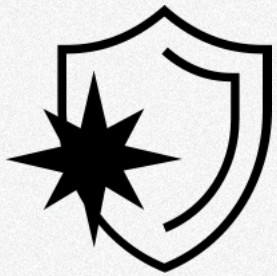
Key-based datasets can be compacted

'Compacted Topics' retain only the most recent events, with any old events, for a certain key removed.

They also support deletes.

Compacted tops reduce how quickly a dataset grows, reducing storage requirements while also increasing performance of replication jobs.

Topic Durability

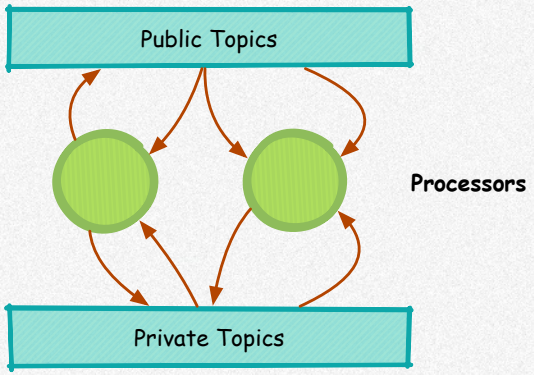


Kafka can be used as a long-term storage layer

If you set the retention to a *"forever"* or enable log compaction on a topic, data will be kept for all time.

Some use cases that support this are event sourcing, in-memory caches (with compacted topics), stream processing or change data capture.

Public and Private Topics



Public and private topics should be separated from each other.

Some teams prefer to do this by convention, but a stricter segregation can be applied using the authorization interface.

Assign read/write permissions for private topics only to the services that own them.

Use a Schema Registry

Always use schemas to promote a durable, shareable contract between producers and consumers.

A schema registry provides a centralized repository for stream metadata.

Having a schema registry helps with data management, data discovery and automatic data pipelines.

A schema registry can also help with proper guards around schema evolution, caching, storage and computation efficiency.

Serializing Messages with Avro

Open source data serialization protocol that helps with data exchange between systems, programming languages, and processing frameworks.

Why Avro?

Avro is a rich library and messaging protocol that supports direct mapping to/from JSON.

It is also space efficient and fast, with wide industry support across different languages.

Avro can also support automated pipelines for data replication.

Avro records are also evolvable.

The Unbundled Database

Databases

"We have been using the database as a kind of gigantic, global, shared, mutable state. It's like a global variable that's shared between all your application servers."

- Martin Kleppmann

40

Transactions

A sequence of one or more SQL operations are treated as a single operation.

Transactions appears to run in isolation, completely separated from each other.

If any part of the system fails, each transaction is either executed in its entirety or not all.

WHY ACID?

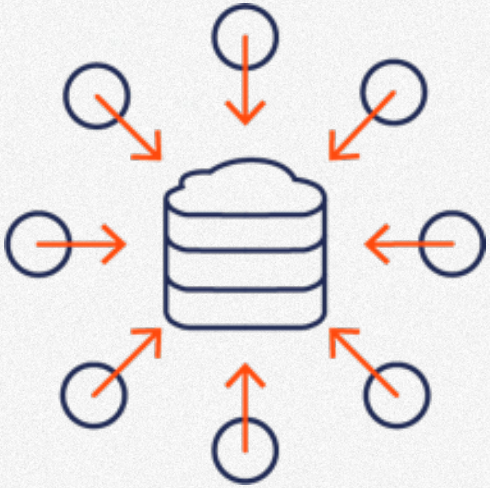
ACID is old school!

What consistency

do you really need

and ***when***?

ACID 2.0



Associative

a(bc) = abc

Set(a).add(b) = set(b).add(a)

Commutative

ab = ba

max(1, 2) = max(2, 1)

Idempotent

aa = a

map.put("key1", "value1").put("key1", "value1") always result in a single entry.

Distributed

(ab)c = (ac)b, for concurrent c and b

Mostly symbolical.

Indexes

Data structure that increases the speed of data lookup operations.

Indexes quickly locate data without having to scan every row in a database table, but incur the cost of additional writes and storage space to maintain the index data structure.

Views

The result set of a stored query on the data

Database users can query views just as they would any other persistent database object.

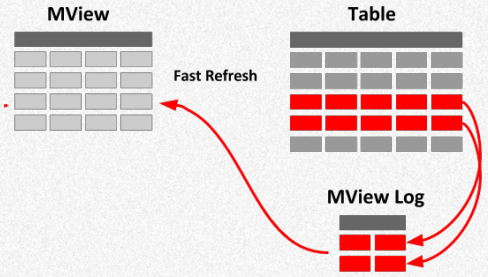
Views can be materialized or virtual.

Materialized Views

Query cached and run by the database

Whenever any of the underlying data changes, the materialized view is updated too.

A view *precomputes* the query to get the data in exactly the right form for your use case. When it comes to querying the view, all the hard stuff is already done.



Databases shouldn't be shared

Databases are pools of global, shared mutable state.

Databases introduce an unusually strong type of coupling.

This comes from the broad, magnifying interface that these systems expose to the outside world.

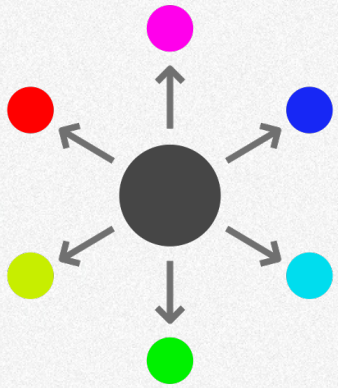
As services interact with the rich interface provided by databases, they get sucked in. Service and database becoming increasingly intertwined.

The Unbundled Database

Splitting database concerns into different layers.

A database is composed of several concepts rolled into a single logical unit: storage, indexing, caching, query, and transactions.

Unbundling is breaking out these components and recomposing in a way that is more sympathetic to the target system.



Rethinking the Materialized View

Can we think of materialized views as continuously updated caches?

We will need:

1. The ability to write data transactionally into a log that maintains immutable records of these writes.

2. A query engine that replicates the journal into a view that can be queried.

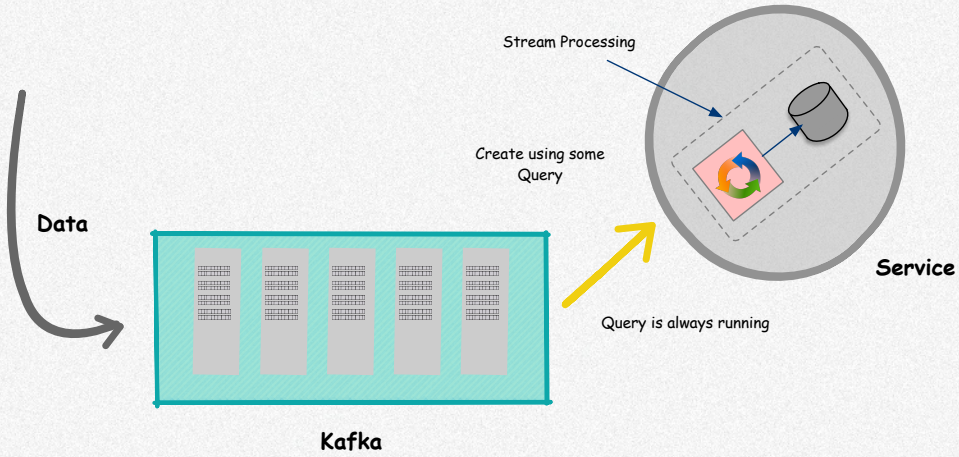
All these elements need to be decentralized and operate as independent entities.

Kafka can handle both the log structure and atomic writes.

Stream processing engines are a great fit for the role of the query engine.

.

Unbundled Databases are safe to share



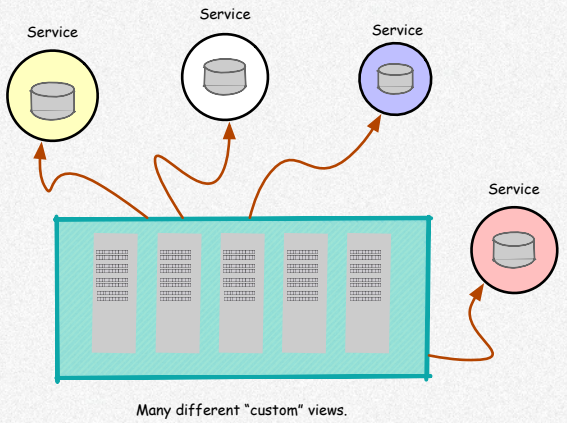
The log provides a safe, immutable, low-coupling mechanism to share datasets across services.

The loose coupling comes from the simple interface the log provides: seek and scan.

This makes the dominant source of coupling the data itself, freeing both sides of the equation from tight coupling.

Kafka becomes the **centralized** event stream log.

Creating Service specific views



Unbundled databases offer an approach where trade offs between reads and writes are no longer needed.

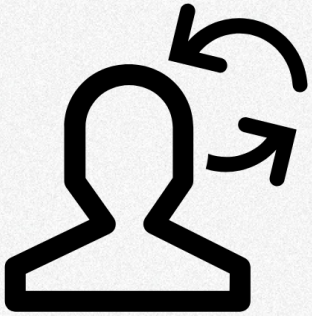
These materialized views don't need to be long-lived.

A distributed log 'remembers', which means the views don't have to.

Views can be ephemeral, implemented as simple caches that can be thrown away and rebuilt at anytime from the log.

Stream Processing

Remember…



Services broadcast events

Services are not modeled as a collection of remote requests and commands.

They become a cascade of notifications, decoupling each event source from its downstream destination.

Events are triggers and facts

Events make up a narrative that not only describes the evolution of the business domains over time, they also represent full datasets.

Kafka Streams

Library that any standard Java application can embed.

Low overhead.

Stream processing meant to run as part of the application.

Built for reading data from Kafka topics, processing it, and writing the results to back to Kafka.

Uses the Kafka cluster for coordination, load balancing, and fault-tolerance.

KSQL: SQL-like interface.

Ideal for ETL (*KStreams)* and aggregations (*KTables).*

Akka Streams

Scala / Java implementation of reactive streams

Based on Akka actors.

Designed for general purpose microservices.

Very low latency.

Mid volume, complex data pipelines.

Efficient per-event processing

Very rich ecosystem:

Alpakka - connect to almost everything! (dbs, files, etc.)

Akka cluster and Akka persistence

Apache Flink

Clustered stream processing engine.

Operator-based computational model.

Large scale.

Automatic data partitioning.

Exactly-once processing.

Very low latency.

Handles high data volumes: 1M/sec.

Can run batch jobs.

Apache Spark

Cluster computing framework with the largest global user base. Written in Scala, Java, R and Python.

Micro batches computing model.

Large scale and automatic partitioning.

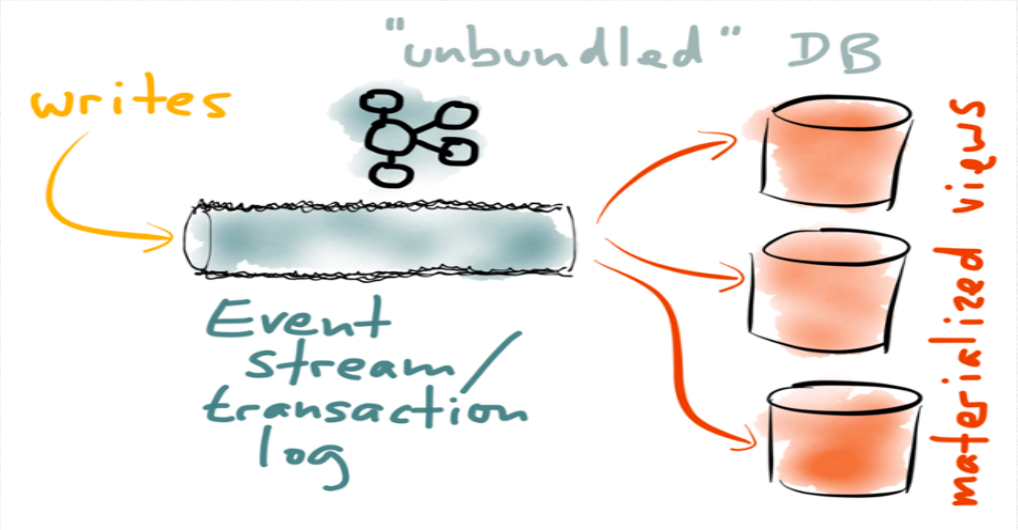
Medium latency, evolving to low.

Handles high data volumes: 1M/sec

Rich SQL, ML options.

Very mature; huge community support and drivers across a variety of programming languages.

Use Case: Replication Streams



Creates in-sync, read-only materialized views of the underlying log in a format that's most sympathetic to the target system.

Builds materialized views from the writes in the log.

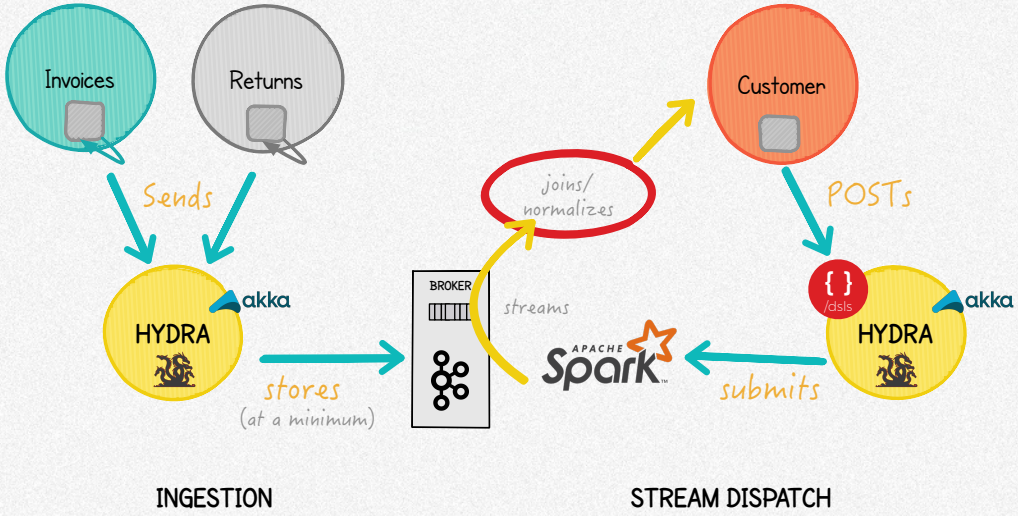
Replication stream behaves like a database transaction log.

Views are like database secondary indexes: optimized for querying and reading.

There could be many different shapes of the same data: a key-value store, a full-text search index, a graph index, an analytics system, and so on.

Even more generic…

Materialized Views and Stream Processing



Source: http://office.microsoft.com/en-us/excel-help/present-your-data-in-a-column-chart-HA010218663.aspx

thank you

contact information

alex-silva@pluralsight.com


http://linkedin.com/in/alexvsilva


@thealexsilva