CHAPTER 8

Cloud Native Data Architecture

In previous chapters, we discussed the application side of cloud native architecture and showed use cases. In this chapter, I will provide details of the data part of a cloud native architecture. As you already know, data is a vast subject; in fact, you can find hundreds of blogs, articles, and books about data. Here I am not covering the topic, but only what's relevant to cloud native architecture.

Enterprises are continuing to move to cloud native architectures, and data plays a pivotal role in that. Data is everywhere; however, the importance and usage of data has changed over time.

Bad data can have significant consequences in an enterprise. Poor-quality data is often pegged as the source of operational problems, inaccurate analytics, and ill-conceived business strategies. According to Gartner, recent research has shown that organizations believe that poor data quality is responsible for an average of $15 million per year in losses. This is a huge loss incurred because of data quality.

Almost every enterprise today is seeking to position itself as a data-driven organization. Businesses are aware of the myriad benefits that can be leveraged when making intelligently empowered decisions and providing customers with top-notch, hyper-personalized experiences, often using artificial intelligence and machine learning models.

This chapter covers the following details of data related to the cloud native phenomenon:

- How has data gained importance?
- How useful is data in your day-to-day business?
- Data storage types and polyglot data architecture
- Data replication strategies

© Shivakumar R Goniwada 2022
S. R. Goniwada, *Cloud Native Architecture and Design*, https://doi.org/10.1007/978-1-4842-7226-8_8

- Data lake and data mesh usage
- Data streaming and change data capture
- Data processing for an analytics platform

Rethinking Data in a Cloud Native World

When dealing with disruption in both business and technology, one area that cannot be forgotten is the data layer. Enterprises must rethink their data layer strategy as they move their landscape to cloud native technologies.

In today's digital world, if an IT application lags for even a few seconds, it can have an enormous downstream impact on the end customer experience and on the business's success. Data processing must be quick enough to keep up with the real-time business-critical applications and today's consumer demand. If travel aggregator apps, maps, food delivery apps, etc., don't provide data instantly, customers will stop using them.

Cloud computing has made a big impact on how we build and operate software today, including how we work with data. More and more companies are embracing the cloud on a daily basis, especially after the pandemic, and shifting their data centers to the cloud, decentralizing their organizations, and making their application architecture more cloud native distributed in nature to enable the pace of innovation necessary to service real-time user needs.

To deliver a consistently fast, satisfying customer experience, the data is very important and must be modernized, moving from batch to streaming and data lake to data mesh, etc. Your enterprise's application is generating more and more data. The traditional way of handling data is simply too slow and does not meet the customer's business goals. In cloud native architecture, the data store must follow the polyglot principle explained in Chapter 5. Just storing static data is not enough; the polyglot principle and analytics principles are required for future data analysis. Enterprises need to make real-time decisions and predictions.

Organizations continue to face a range of complexities in transforming to a data-driven approach and leveraging its full potential. While migrating legacy systems, shunning legacy cultures, and prioritizing data management are all valid goals, the architectural structure of data platform initiatives can prove to be a major roadblock.

The need for traditional data storage functions such as backups, replication, and security don't go away in cloud native data services; they are just initiated and managed in new and real-time ways. With data replication, you often pull data from multiple sources to carry out a task, and increasingly such aggregation is on demand. Earlier you were doing nightly batch jobs, but in the cloud native world, you use data streaming techniques in real time.

In the data storage part, there are no changes in the storage and the create, read, update, and delete (CRUD) operations, but there are various options available to store data by using polyglot principles. You can choose from various storage mechanisms such as traditional RDBMS, NoSQL, caching, etc.

There is no change in data visualization. Earlier we generated reports by using classic reporting tools; now you have more options to choose from with rich functionality.

In a nutshell, the changes are in the way you are adopting the data and using it for various analytical decisions.

Cloud Native Data Persistence Layer

For a lot of businesses and enterprises, cloud computing has made a big impact on how they store data. The cost of storing data has significantly decreased. The management of database systems requires less work with the advent of cloud vendor-managed and serverless data storage. This makes enterprises choose various data storage types based on the data classification.

A polyglot persistence principle encourages cloud native services to decentralize the data; it is also common that data is replicated and partitioned in order to scale the system. Figure 8-1 shows how a typical cloud native architecture applies the polyglot persistence principle with data spread across the architecture.

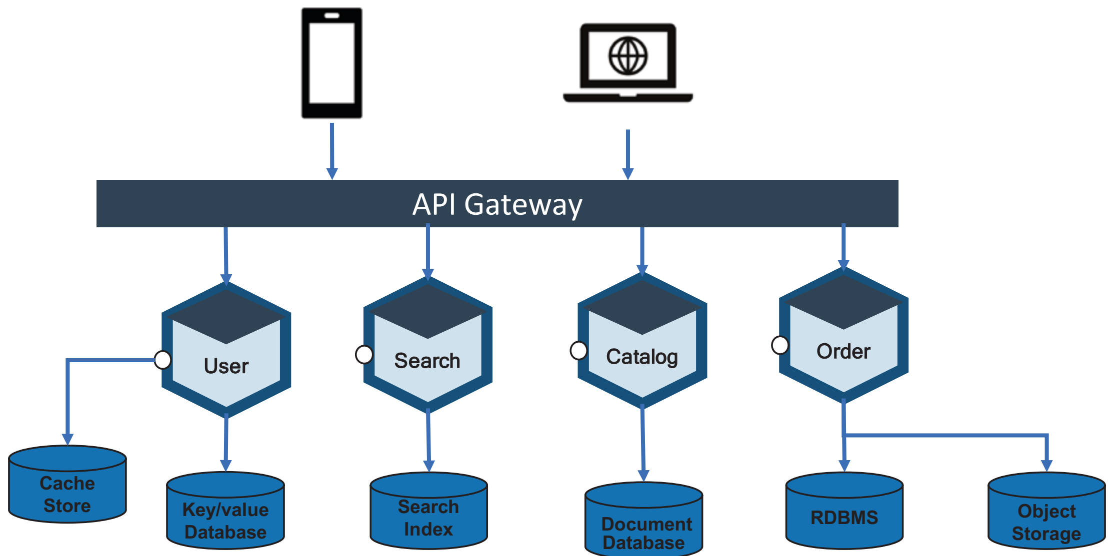

Figure 8-1. *Cloud native polyglot persistence*

Cloud native applications use managed and serverless data storage and processing services; all major cloud providers offer several different managed services to store, process, and analyze data. In addition to cloud providers, various database companies provide managed services on the cloud. For example, MongoDB provides managed services with Atlas, Redis provides managed cache storage, etc. By using managed cloud storage, you can focus on developing business logic that uses the data and database instead of spending time and resources managing the database.

Cloud Native Data Characteristics

For a cloud native application, you can use a blueprint like the 12-factor criteria to design it, as mentioned in Chapter 4, but for the data design, you need to consider the following key characteristics:

- Prefer a cloud native database that shards, tolerates faults, and is optimized for cloud storage.
- Prefer cloud native data that is independent of fixed schemas.
- Cloud native data can be duplicated for ease of access.
- Prefer managed data storage and analytics services.

- Use polyglot persistence, data partitioning, and caching.
- Embrace eventual consistency and use strong consistency when necessary.
- Cloud native data integrates through service and event streaming.
- Adopt a data mesh wherever possible instead of a data lake.
- Prefer real-time analysis to batching.
- Deal with data distribution across multiple data stores.

How to Select a Data Store

Selecting the right database is important for the successful completion of your project. There are about 347 databases available including RDBMS, NoSQL, event stores, etc. It can be difficult to determine which products to use, and sometimes you may choose the wrong database for your application that limits your whole application. I have witnessed projects change their database after pushing it into production. This might cause heavy loss to enterprises because you need to migrate, test, etc., so choosing the right database from the start is important. I will provide as many details as possible to help you to choose the right database.

I will first start with various types of data.

Objects, Files, and Blocks

Objects, files, and blocks are storage formats that hold, organize, and present data in different ways. Object storage manages data as an object and stores data with metadata and a key that is used as a reference for the object. File storage organizes and represents data as a hierarchy of files in folders. Block storage chunks data into arbitrarily organized, evenly sized volumes.

Note Objects are considered images, documents, and files.

The major cloud providers such as AWS, Azure, and Google provide inexpensive object storage, and the data can be accessed through APIs. Each object is stored in a key-value pair with metadata linked into it, and it is stored with versions and globally available. The object storage tools are AWS's S3, Azure's blob storage, and Google Cloud storage.

Every document in a file is arranged in some type of local hierarchy. Network-attached storage (NAS) is a file-level storage architecture. Use it when using a library or service that requires shared access to files. Various NAS providers are available in cloud environments including natively from cloud vendors. A few major NAS vendors are NetApp, Dell EMC, HPE, Hitachi Vantara, IBM, Cloudian, Qumulo, and WekaIO.

Block storage breaks data into smaller blocks and stores the blocks separately. Each block of data is given a unique identifier, which allows a storage system to place the smaller pieces of data wherever is most convenient. Use block storage for applications for persistent local storage. For this kind of data, use any database to store it.

Databases

A database is a collection of data stored in an orderly manner, as shown in Figure 8-2. It is a structured set of data hosted on the hardware. There have been some new players in the database world over the past few years, and the number of databases available for us to choose from continues to grow every year.

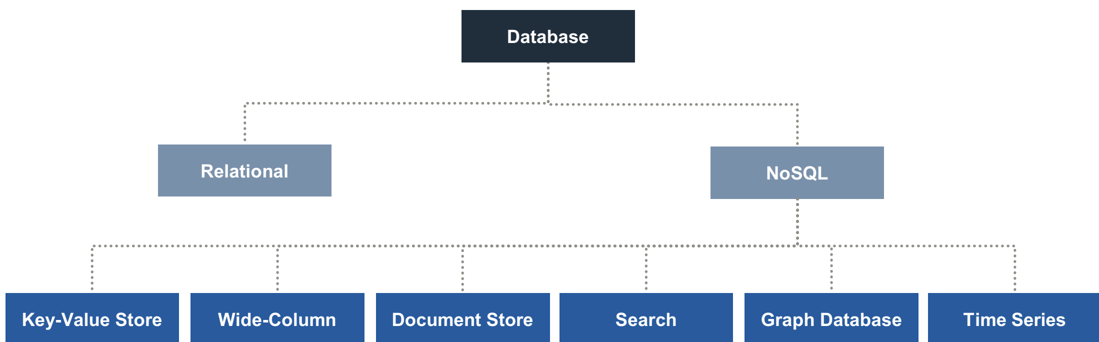

Figure 8-2. *Database types*

Many of these databases have been designed for specific use cases; some store graph-related data, some store financial models, etc.

Note I have not covered caching technology because it is part of the key-value store family. Relational and object databases are part of the relational family.

Relational Database

A relational database is a collection of data items with a predetermined relationship between them. This data is organized into a set of tables, columns, and rows. A relational database provides access to data points that are related to one another. Each column in a table holds a certain kind of data and fields to store the actual values of an attribute. Each row in a table can be marked with a unique identifier called the *primary key*, and the rows in multiple tables can be made related using foreign keys. This data can be managed with CRUD operations.

Relational databases have been around for a long time. The most popular commonly used database, as of today, is still a relational database. The relational model is the best for maintaining data consistency across application and database instances. The relational databases support atomicity, consistency, isolation, and durability (ACID) properties with strong consistency.

Several factors can guide your decisions when choosing among relational database types. You need to ask the following questions before choosing a vendor:

- What is our data accuracy requirement?
- Do we need scalability? What is the anticipated growth?
- How important is concurrency?
- Where are we hosting the database?
- What kind of application are we developing?

Use Figure 8-3 to decide whether you need an RDBMS for your application.

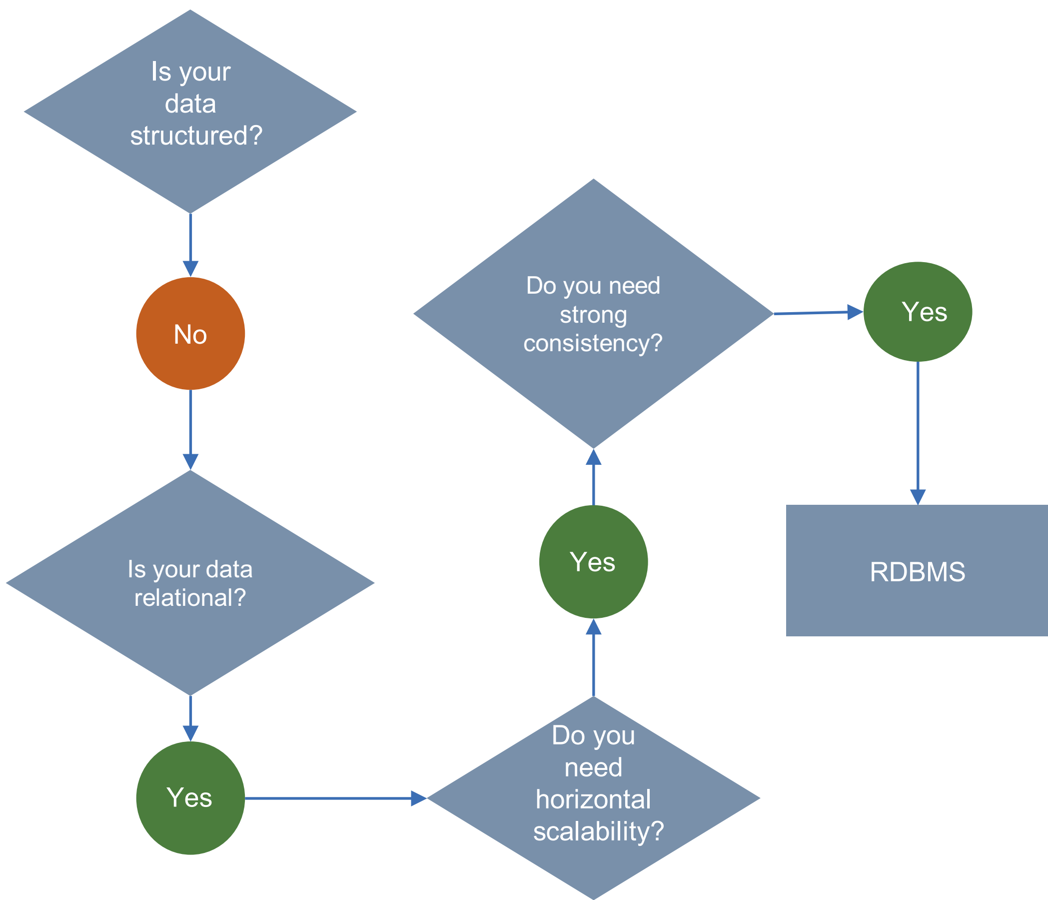

Figure 8-3. *RDBMS decision flow*

Key-Value

A key-value data store is a type of nonrelational database that uses a simple key value to store data. In key-value pairs, a key serves as a unique identifier. Both keys and values can be anything, ranging from simple objects to complex compound objects, and they can store dictionary/map/array objects.

Key-value databases use compact, efficient index structures to be able to locate a value quickly and reliably by its key, making them ideal for systems that need to find and retrieve data in real time. Key-value databases allow programs to retrieve data via keys, which are essential names, or identifiers, that point to some stored values.

Key-value databases are scaled out by implementing partitioning, replication, and autorecovery. They can scale by maintaining the database in RAM and can minimize the effects of ACID guarantees by avoiding locks, latches, and low-overhead server calls.

Several factors can guide your decision when choosing among the key-value database types. You need to ask the following questions before choosing any type of database:

- What kind of data do we want to store?
- Do we need scalability?
- Do we want our data to share across microservices?
- What kind of application we are developing?

Use Figure 8-4 to decide whether you need the key-value type for your application storage.

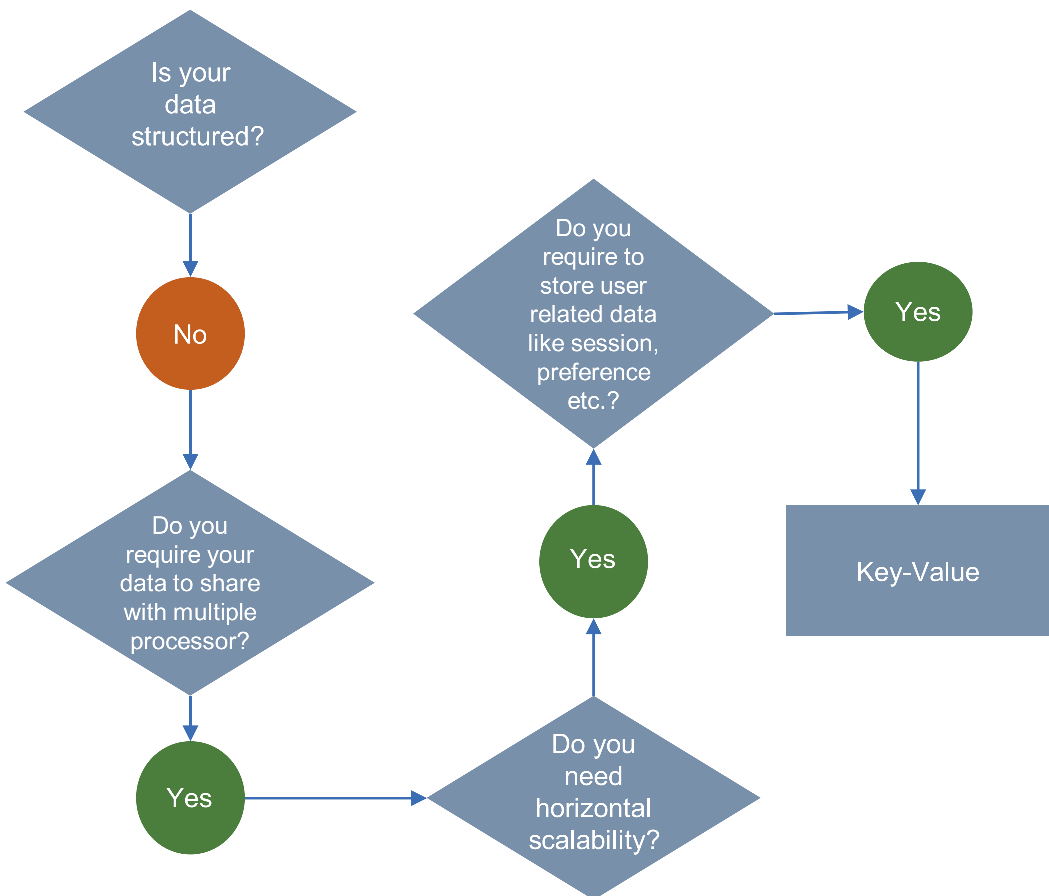

Figure 8-4. *Key-value store decision flow*

The following are key-value stores: AWS Dynamo DB, Redis, Riak, Couchbase, Berkeley DB, Cassandra, etc.

Document Database

A *document-oriented database* is a way to store data in JSON format rather than simple rows and columns. A document store does assume a certain document structure that can be specified with a schema. A document store is the most natural way of storing data among NoSQL-type databases, which are designed to store the document as is.

Each document in a store contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, Booleans, arrays, or objects, and their structure is aligned with the application developer working with the code. Because of their variety of field value types and powerful query languages, document databases are great for a wide variety of use cases. You can use these databases for much of what was traditionally stored in a relational database like PostgreSQL or MySQL.

Documents in a database map to the objects in your services. There is no need to decompose data across tables, run JOINs, or integrate a separate ORM layer. The schema in the document database is dynamic, and you don't need to define it at design time. The document database features are expressivity of the query language and richness of indexing. With ACID transactions, you maintain the same guarantees you're used to in SQL databases.

Document databases are distributed systems at their core, and documents are independent units, which makes it easier to distribute them across multiple servers while preserving data locality. Replication with self-healing recovery keeps the database highly available, and native sharding provides elastic and application-transparent horizontal scale-out.

Several factors can guide your decision when choosing from the document database types. The following are a few questions you need to ask before choosing any type of database:

- What kind of data do we want to store?
- Do we need scalability?
- Does our application need to be available globally?
- Do we want SQL queries?

- Do we want a flexible schema?
- Do we want to store all kinds of data like modeling, semistructured, and unstructured data in one database?
- What kind of application are we developing?
- Do we want to store content or catalogs?

Use Figure 8-5 to decide whether you require a document database for your application storage.

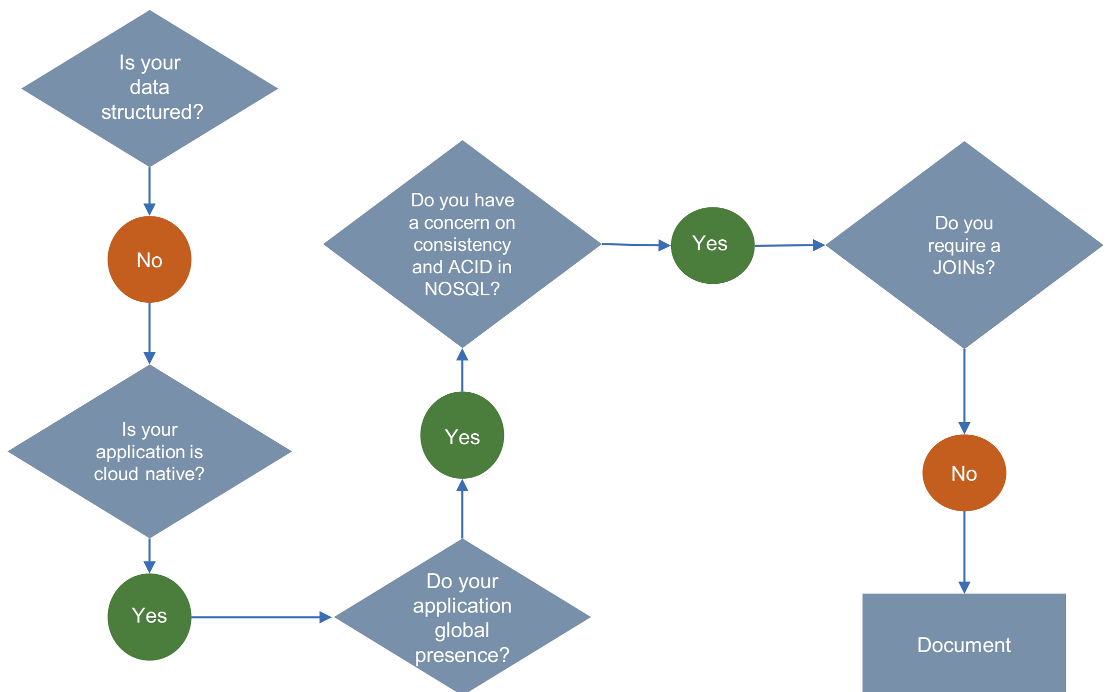

***Figure 8-5.** Document store decision flow*

The following are a few major players in the area of document databases: MongoDB, CouchDB, Couchbase Server, Cosmos DB, Document DB, MarkLogic, Oracle NoSQL, etc.

Wide-Column Database

A *column database* organizes data into rows and columns and can initially appear very similar to a relational database. It stores data in tables, rows, and dynamic columns. A columnar database stores each column in a separate file. One file stores only the key column, and the other stores the remaining fields. Wide-column stores provide a lot of flexibility over relational databases because each row is not required to have the same columns.

Each column holds a set of columns that are logically related and typically retrieved or manipulated as a unit. Other data that is accessed separately can be stored in separate column families. The wide columns store data like a two-dimensional key-value database. They are good to store a large amount of data when you can predict what your query pattern will be. They are commonly used for storing Internet of Things (IoT) and user-profile data.

Each column in a row is governed by auto-indexing on each function. It gives improved automation with regard to vertical and horizontal partitioning with better compression and auto-indexing columns.

Several factors can guide your decision when choosing among the wide-column database types. The following are a few questions you need to ask before choosing a type of database:

- What kind of data do we want to store?
- Do we want to store data with horizontal scaling?
- Is our application required to be available globally?
- Do we want SQL kinds of queries?
- Do we want data to be compressed?
- What kind of application are we developing?
- Do we want to store IoT or geographical map data?
- Do we want a database for analytics?

Use Figure 8-6 to decide whether you require a wide-column database for your application storage. The major databases are Cassandra and HBase.

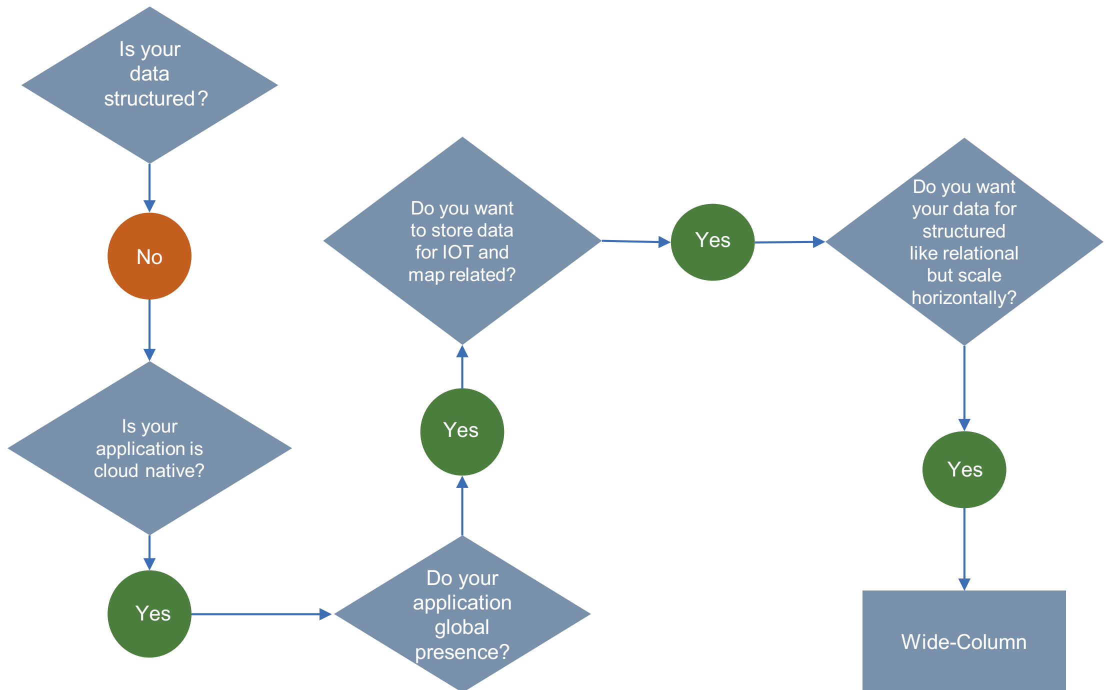

Figure 8-6. *Wide-column decision flow*

Time-Series Database

Time-series data is a sequence of data points collected over time intervals, giving you the ability to track changes over time. Time-series data can track changes over milliseconds, days, or even years. This could be server metrics, application performance monitoring, network data, sensor data, trades in the market, etc.

The time-series database is optimized for a time. It is built specifically for handling metrics and events or measurements that are time-stamped. These databases generally need to support a very high number of writes. Time-series databases are commonly used to collect large amounts of data in real time from many sources. Updates to the data are rare; more common are inserts and bulk deletes.

The size of the data structure is small for time and other coordinates. Time-series data is good for storing telemetry data; popular uses include Internet of Things (IoT) sensor devices such as autonomous cars, digital twin use cases, etc.

Several factors can guide your decision when choosing a time-series databases. The following are a few questions you need to ask before choosing any type of database:

- What kind of data do we want to store?
- Do we want stored data with horizontal scaling?
- Is our application required to be available globally?
- Do we want to store data for IoT sensors or telemetry?
- Do we want to use this data for metrics or analytics?
- What kind of application are we are developing?

Use Figure 8-7 to decide whether you require a time-series database for your application storage. The major databases are Apache Druid, Riak-TS, and AWS Dynamo DB.

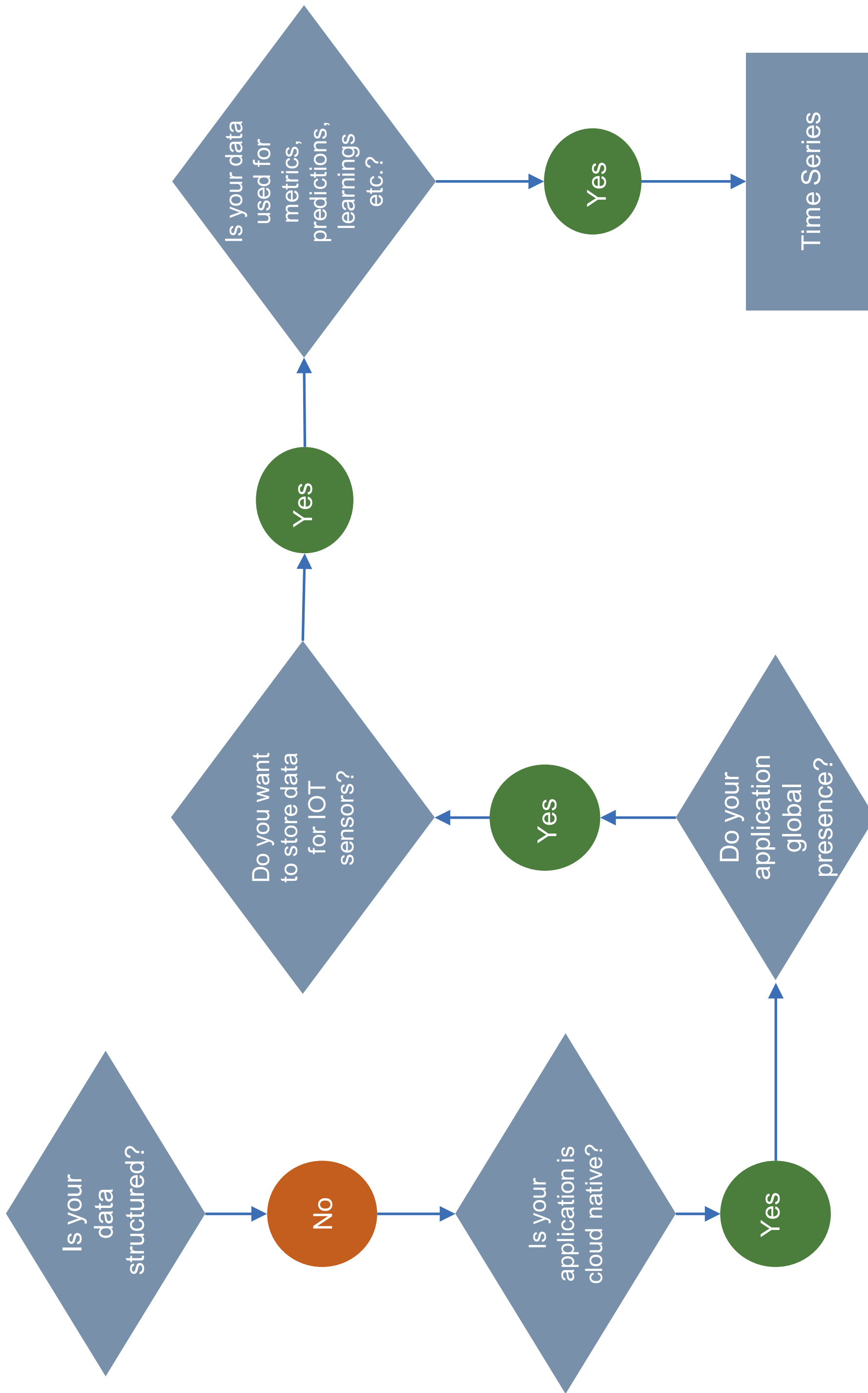

Figure 8-7. *Time-series data store decision flow*

Graph Database

A *graph database* is a special kind of database storing complex data structures and most notably used for social networks, interconnected data, fraud detection, knowledge graphs, etc.

It stores data in nodes and edges. Nodes typically store information about people, places, and things, while edges store information about the relationship between the nodes. You can think of a node as an entity, and edges define the relationship between the nodes. An edge will often define the direction of the nature of a relationship.

The graph database shards data across many servers or clusters and locations. It distributes and parallelizes queries and aggregations over multiple databases.

Several factors can guide your decision when choosing among graph database types. The following are a few questions you need to ask before choosing any type of database:

- What kind of data do we want to store?
- Do we want stored data with multiple shardings or clusters?
- Is our application required to be available globally?
- Are our use cases related to a social network, fraud detection, or knowledge graph?

Use Figure 8-8 to decide if you require a graph database for your application storage. The major graph databases are Neo4J, Orient DB, Arango DB, AWS Neptune, DataStax, IBM Graph, and Apache Graph.

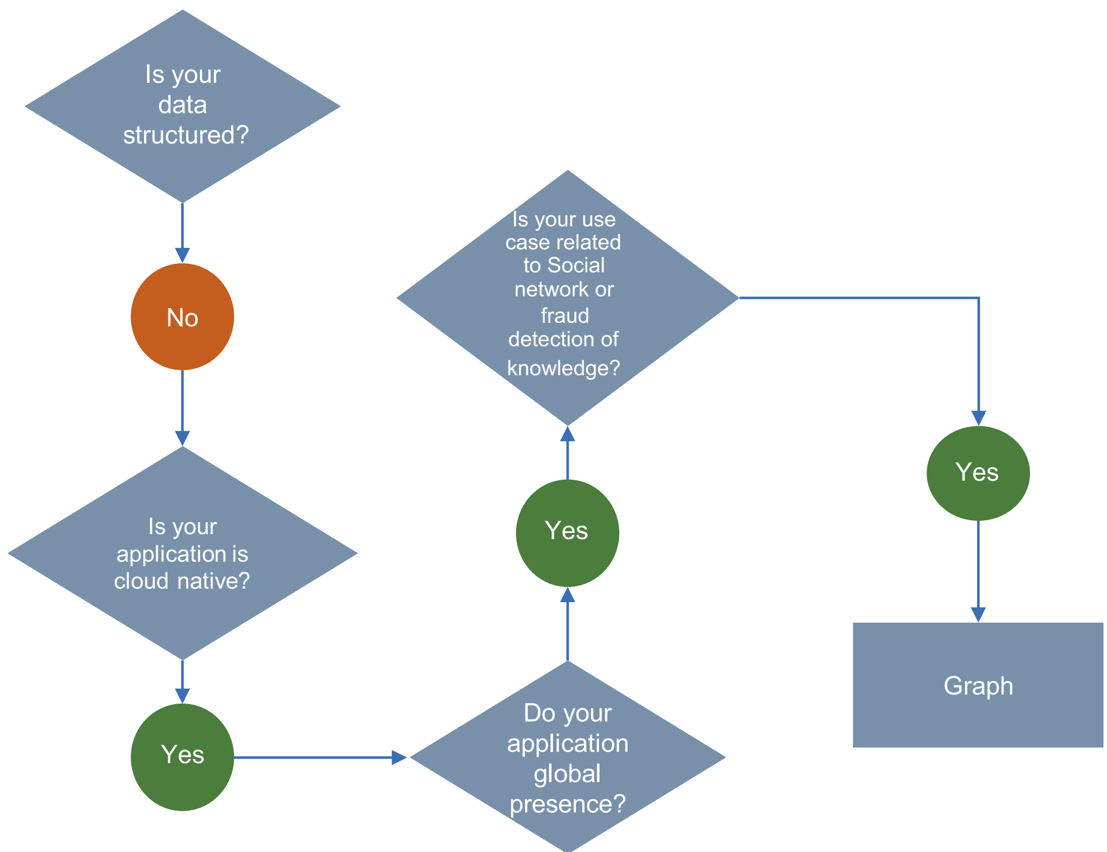

Figure 8-8. *Graph data store decision flow*

Event Store Database

In event-driven architecture, streams and queues are required to store events and messages (more details are explained in Chapter 6). In an event stream, the data is stored as an immutable stream of events. All the events in the event store are new records and do not allow updates; also, you cannot remove or delete an event.

The data in an event store is used to validate an aggregate sequence numbers of events, event snapshots, event sourcing details, etc.

There is various event store data store available such as IBM DB2 Event Store, Event Store DB, and NEventStore.

Search Engine Database

The search engine database is a type of nonrelational database that is used to search for information held in other databases and services. Search engine databases use indexes to categorize similar characteristics among data and facilitate search capability. A search engine index database can index large volumes of data with near-real-time access to the index.

The search engine databases are optimized for dealing with data that may be long, semistructured, or unstructured, and they provide specialized methods for search such as full-text search, complex search expression, and ranking of search results.

The search engine databases can handle full-text search faster than relational databases with indexes.

Several factors can guide your decision when choosing among search engine database types. The following are a few questions you need to ask before choosing any type of database:

- What kind of data do we want to store?
- Is our data used for search or log analysis or integrated monitoring and dashboard?
- Do we require indexing in the data store?

Use Figure 8-9 to decide whether you require a search database for your application storage. The major databases are Elasticsearch, Splunk, ArangoDB, Solr, AWS Cloud Search, Alibaba Cloud Log Service, and MarkLogic.

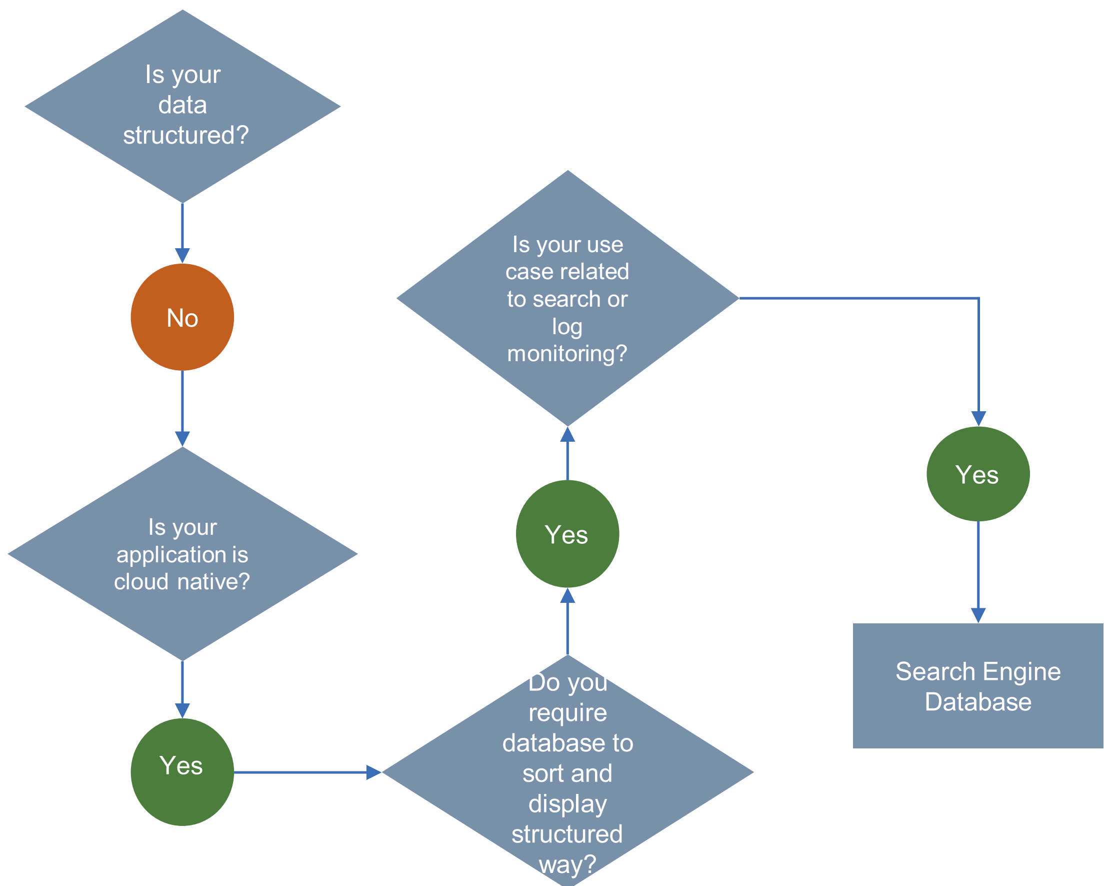

Figure 8-9. *Search engine data store decision flow*

Selecting a database is confusing when you have a vast number of options available today and new ones are constantly improving and adapting to cloud native. A website that tracks database popularity, DB-engines (`https://db-engines.com`), lists 347 different databases as of this writing. As you are moving toward cloud native architecture, you have the flexibility to choose a specific database based on your use cases. When choosing the specific use cases, you need to consider the following aspects:

- Consider the skillset of the team.
- Go for the managed serverless database from cloud vendors or individual database vendors (for example, MongoDB offering Atlas).

- Go for lightweight databases instead of big monolithic databases.
- Analyze your use cases and ask questions as provided for each database type; nowadays most NoSQL databases offer similar features like relational databases.

Data Replication

Data replication is the process of updating copies of your data in multiple places at the same time to improve reliability, fault tolerance, accessibility, and decision-making. The goal of replication is to keep your data available for various purposes like to make decisions or to make transactions available to your customers.

Data replication works by keeping the source and target synchronized. That means any change in source data is reflected accurately and quickly in the target data based on your replication model.

The use case of data replication includes high availability, migration between systems, operational data stores or data hubs, data consolidation in the reporting system, data warehouses, and data lakes, etc.

Traditionally, in an enterprise, the data replication occurs either from database to database or file are uploaded to the database by using ETL tools.

There are two database replication methods.

- Physical database replication
- Logical database replication

Physical Database Replication

Physical database replication is a block-based replication that uses a binary format to keep an exact database copy in sync with the primary database. Using the binary format for database replication provides completeness: the replicated database is an exact copy of the primary database including tables, relationships, indexes, triggers, stored procedures, etc. This kind of replication is common in disaster recovery use cases.

Logical Database Replication

This is a method of replicating data objects and their changes based on their replication key. Logical database replication is the most common method of replication in a cloud native architecture. It uses the publish/subscribe paradigm to replicate data from source to target databases. The logical replication of a table starts by taking a snapshot of the data on the publisher database and copying that to the subscriber.

In the logical database replication, you can do full database refreshes or logical refreshes or change data capture (CDC).

Full Data Refresh

In the full load refresh replication, all the data in the publisher loads data to the subscriber at an interval and overwrites all the data in the subscribed database. This method is very resource-intensive; usually enterprises adopt this approach only for the initial load.

Partial Data Refresh

In the partial refresh replication, use a column in the table that is modified for every change to the row with the timestamp. Use a filter when retrieving the data from the publisher instead of selecting full data. This approach is reliable only when data is not truncated.

Change Data Capture

CDC is a replication solution that captures database changes as they happen and delivers them to the target database. CDC typically starts by taking a snapshot of the data on the publisher database and copying it to the subscriber database, as shown in Figure 8-10. Once that is done, the changes on the publisher are sent to the subscriber as they occur in real time.

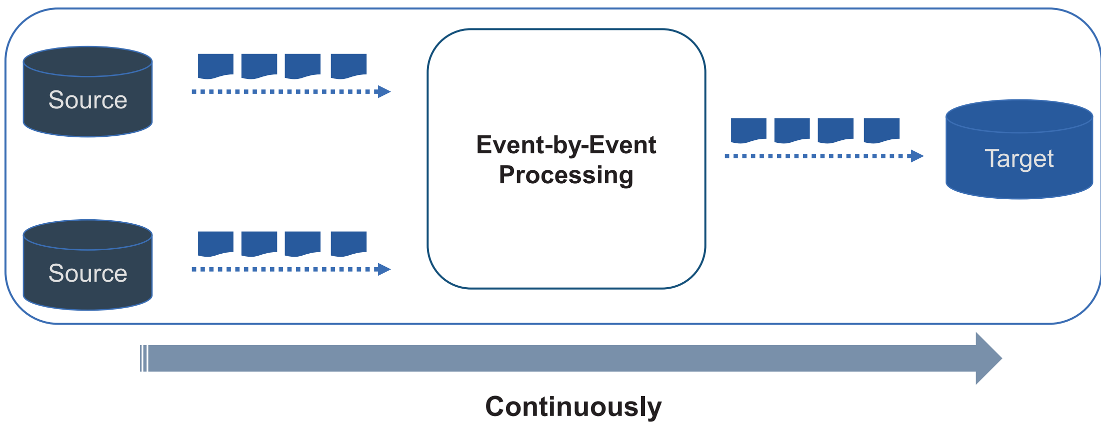

Figure 8-10. *CDC process*

The subscriber applies the data in the same order as the publisher so that transactional consistency is guaranteed for publication with the same subscription.

There are many techniques available to implement CDC depending on the nature of your implementation.

- *Timestamp*: The Timestamp column in a table represents the time of the last change; any data changes in a row can be identified with the timestamp.
- *Version number*: The Version Number column in a table represents the version of the last change; all data with the latest version number is considered to have changed.
- *Triggers*: Write a trigger for each table; the triggers in a table log events that happen to the table.
- *Log-based*: Databases store all changes in a transactional log to recover the committed state of the database. The CDC reads the changes in the log and identifies the modification and publishes an event.

The most preferred approach is the log-based technique. In today's world, many databases offer a stream of data change logs and expose them through an event.

Log-Based CDC

The log-based approach provides real-time asynchronous data integration and provides continuous integration through database logs. This approach allows the solution to transfer and integrate changes to the data incrementally as they occur, rather than making larger updates all at once.

Database transaction logs that store all database events allow for the database to be recovered in the case of a crash. The changes in the source database are captured without making application-level changes and without the overhead on the database and without having scans on operational tables, all of which add workload and reduce source system performance.

In the Figure 8-11 example, service A writes data to a database, and the database writes a change to the logs. The change is then managed by CDC tools and written to a stream of events and subscribed to by multiple consumers; the consumer could be a target database, data lake, or real-time analytics.

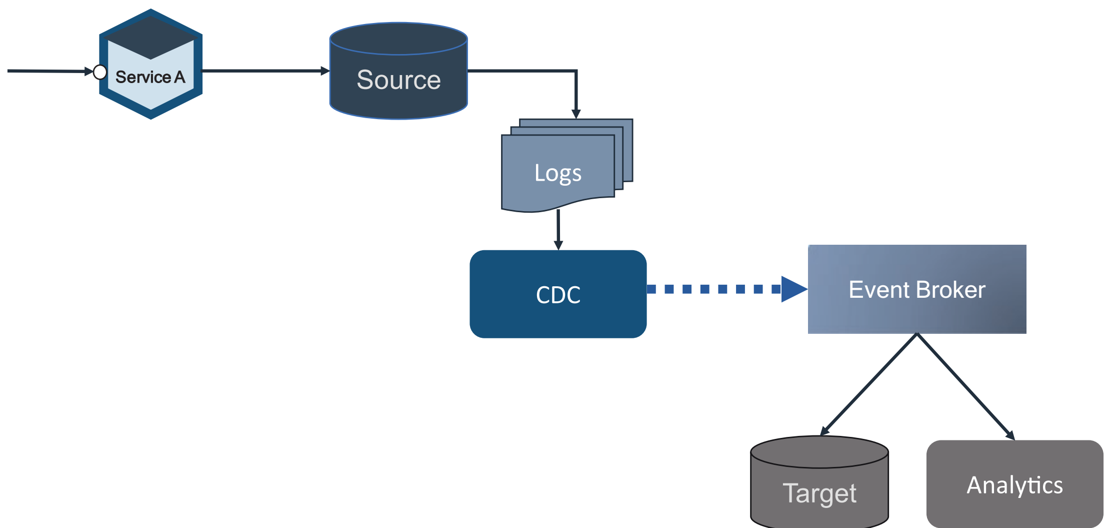

Figure 8-11. *Log-based CDC*

There are a few areas where you need to aware of, such as the following:

- *Concurrency*: Most CDC tools manage the order.
- *Data consistency issues*: In a microservices polyglot architecture, transactions span multiple databases. You need to write a set of changes to the changelog and then apply those changes. All the changes can be written to a stream maintaining order.
- *The compensating transaction*: Apply multiple techniques like saga and CQRS to manage the transaction (refer to Chapter 6 for more details).

The following are the advantages of log-based CDC approaches:

- This approach has a minimal impact on the transactional database.
- This works in near-real-time asynchronous event streaming; it helps you to manage analytics on the fly.
- This approach maintains the order in which the transaction was committed. This is important when the target application depends on the order of transaction, for example, if two services modify the same record instantly.

In cloud native architecture with polyglot persistence and decentralized datastores, these event streams are incredibly helpful in maintaining consistency across these databases. The following are a few common CDC uses cases:

- *Materialized views*: The changes in events can be used to update these views in real time.
- *Auditing and fraud management*: Many transactions are required to conduct auditing. You can use these log changes to track what was changed and when and to help scan all the transactions in real time for anti-money laundering and fraud management.
- *Analytics*: You may require data analytics both on the fly and off the fly. This approach will apply a machine learning model on the event streams and will use the fly analysis from a data lake or data mesh.

- *Decoupling*: When you consider moving from a legacy monolithic application to microservices, your approach should be iterative by applying strangulation. In this case, you need to use this approach to decouple legacy applications and their databases.

Extract, Transfer, and Load

ETL is a process that extracts data from different source systems, transforms the data, and finally loads the data into the target database. This process is not new; you have been using this approach for very long time. As the name indicates, ETL has three steps, as shown in Figure 8-12.

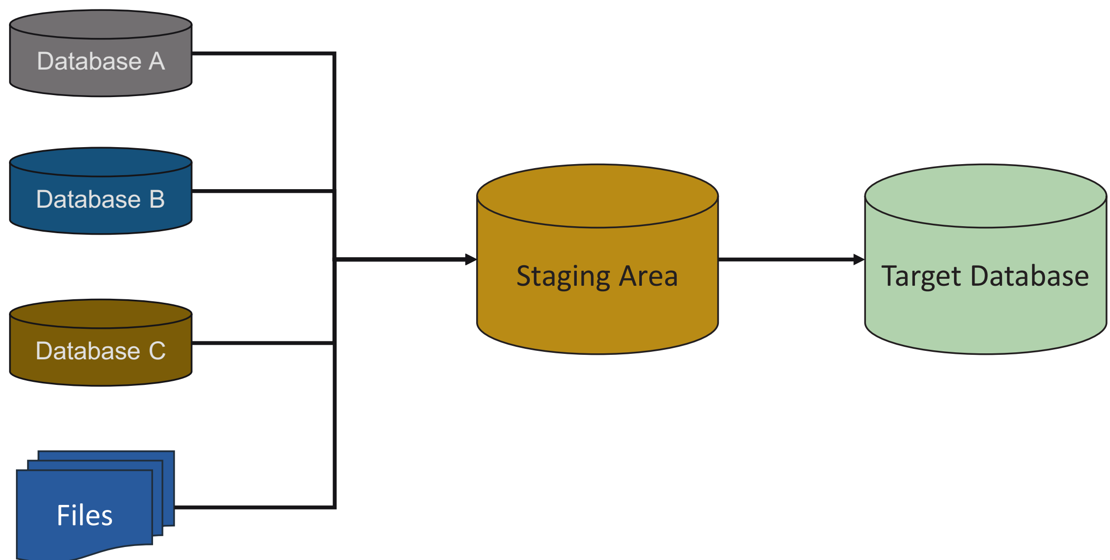

Figure 8-12. *ETL process*

Extraction

In this step, data is extracted from heterogeneous systems and files into the staging area. The source data is transformed in to the staging area without impacting the source system. The staging area is where you can check and apply rules before loading the data into the target database. During the data extraction, the ETL tool will do a sanity check of the data such as type check, duplicates, keys, etc.

Transform

Data extracted from the source databases is in the source database format and needs to be cleansed, mapped to the target, and transformed. This is the key step in the ETL process. In this transformation, you will apply a few rules such as aggregation, etc.

Load

Loading data into the target database is the last step of the ETL process. In batch mode, you need to load a huge volume of data in a short period; hence, the load process should be optimized for performance.

The ETL process is increasingly important to help your organization to analyze data, reducing the load on the transaction database by keeping read data from the operational data store.

In a modern-day cloud native architecture, moving and processing data from one source to another is increasingly important and common. Still, a lot of use cases are required to use the ETL process such as nightly batches in the financial domain, inventory reconciliation in the retail domain, etc. All the major cloud providers offer managed ETL services, such as AWS Glue, Azure Data Factory, and Google Cloud Data Flow.

Decoupling Big Data Management from Distributed Data Meshes

Currently, the big data platforms available in the industry are data lakes and data warehouses. These two hold big, replicated data from various siloed domain databases either through ETL batch jobs or event streaming jobs. The data lake implementation of your organization or client's organization has unclear responsibilities and ownership of the domains in a lake.

In modern-day business, disruption is happening like never before; therefore, we need to make sure that our technology supports the business. Data lakes and data warehouses are good but have their limitations such as centralization of domains and domain ownership. To overcome these challenges, the concept of a data mesh provides a new way to address common problems. Zhamak Dehghani from ThoughtWorks coined the data mesh and wrote a detailed paper on it.

The data mesh essentially refers to the concept of breaking down data lakes and siloes into smaller, more decentralized parts. It is like shifting from a monolithic legacy application toward a microservice architecture. In a nutshell, the data mesh is like a microservice architecture in application development.

You are already familiar with the microservices architecture and the decoupling approach from legacy monolithic services to microservices based on domains by using the domain-driven approach The domain-driven design approach addresses the problems in an application domain and in the transactional data related to that domain, but usually we are not addressing the domains or ownership of the data. The data mesh addresses data domain-driven design.

In a data lake and data warehouse, you might have observed the ownership issues. There might be an owner who can manage and operationalize the big data platforms but not from the domains. The ownership is important. For example, in your organization, you might have seen each vertical tower for finance, healthcare, retail, etc. There is someone in charge of that tower who owns the entire team and is responsible for delivering it and related clients. Similarly, you need an owner for the domain.

The data mesh implementation is based on the four principles shown in Figure 8-13. These are as follows:

- *Domain-oriented decentralized data ownership and architecture*: This principle is about implementing the data domain-driven concept to decouple and decentralize the data and ownership.
- *Data as a product*: This principle is about addressing a concern around accessibility, usability, and harmonization of distributed datasets.
- *Self-service data infrastructure as a platform*: This principle is about services and skills required to operate the data pipeline technologies and infrastructure in each domain.
- *Federated computational governance*: This principle is about data governance and standardization for interoperability, enabled by a shared and harmonized self-service data infrastructure.

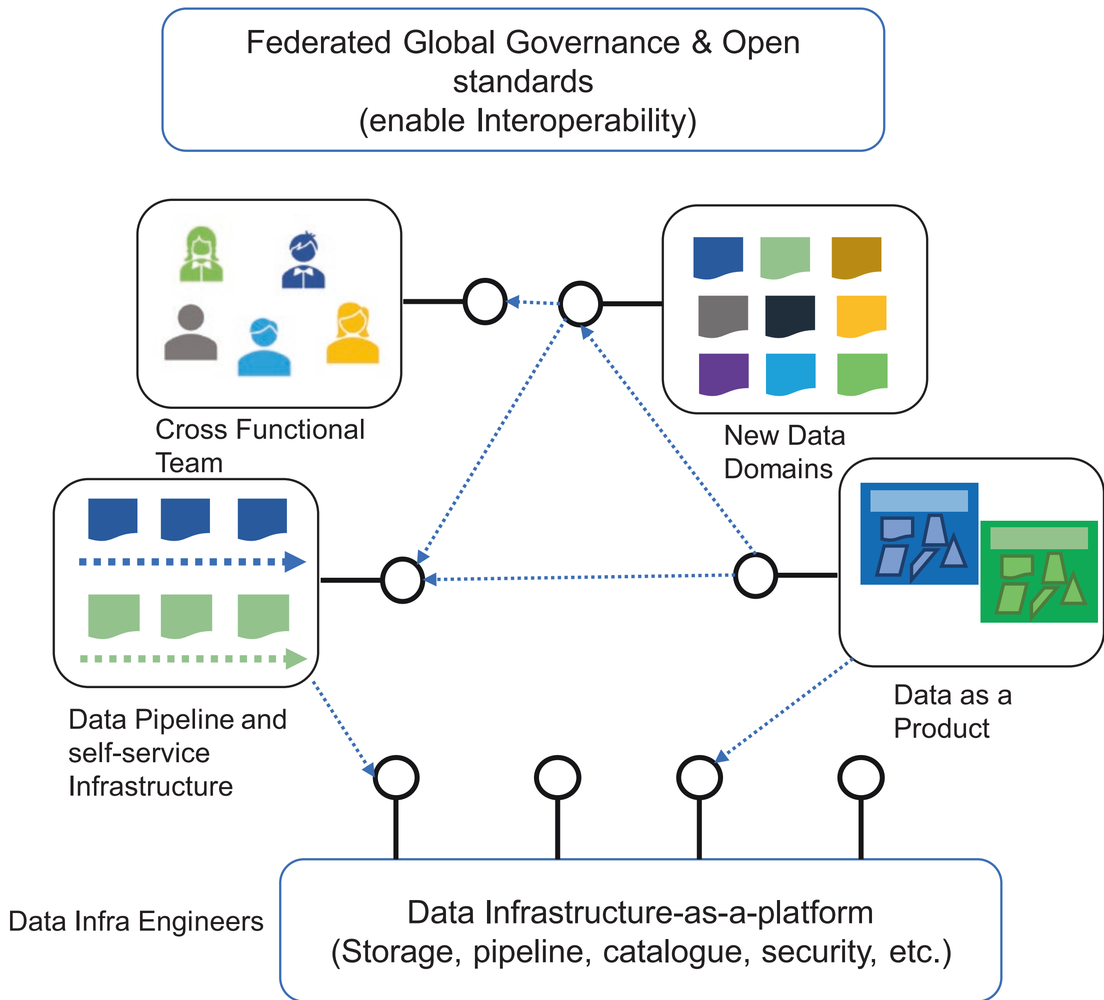

Figure 8-13. *Data mesh architecture*

A *data mesh* refers to the concept of decoupling data lakes and siloes into the smaller, decentralized domain-based model. The analytical scale can scale in the way the microservices and polyglot persistence have allowed transactional data to scale. Zhamak Dehghani explained all four principles in a detailed way at `https://martinfowler.com/articles/data-monolith-to-mesh.html`. I will cover them in a more structured way with an example of how you can implement data meshes in your project. I am using an example of an ecommerce application to explain data meshes.

Figure 8-14 shows the example data architecture. It's a centralized data lake architecture whose goals are to ingest data from all corners of the enterprises; cleanse, enrich, and transform data to the data lake; and serve the dataset in a data lake to diverse requests.

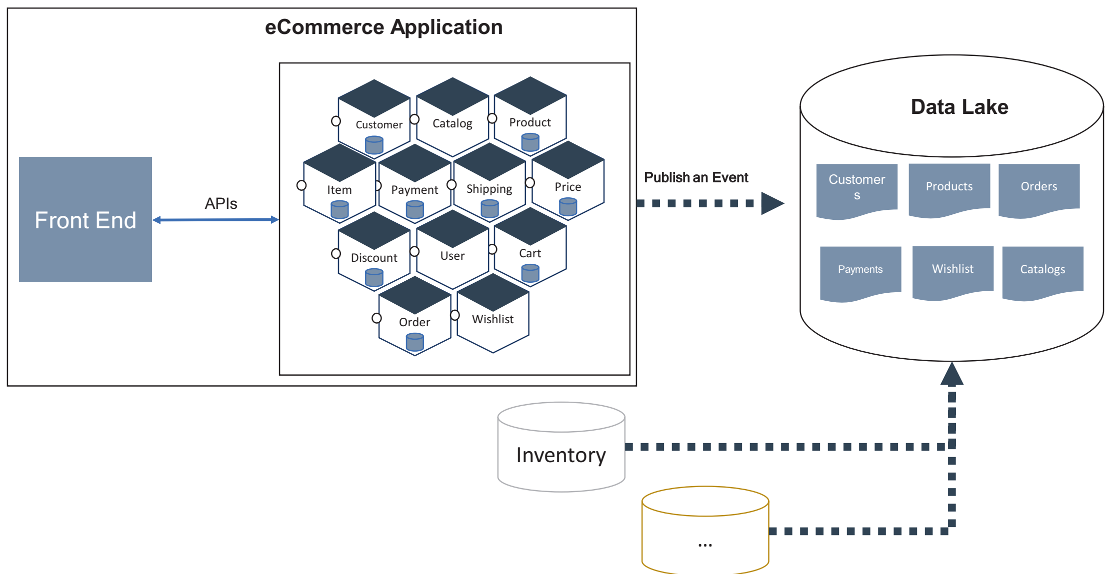

Figure 8-14. *Current data lake architecture*

The monolithic data lake platform contains and owns the data that belongs to different domains, e.g., customers, sales KPIs, inventory, payments, orders, etc., with the business changes. This kind of implementation is no longer helpful to support the required business growth, because of the diverse customers, more adoption of the cloud native approach in an application landscape, and the minimum viable product (MVP) approach.

On the replication side, you are streaming from diverse sources to the data lake, usually any in the organization. You are not building all the replication at once. You might follow an iteration model to build as the business grows. For this replication, you may use an ETL approach or streaming based on the events approach. Both approaches include ingestion, cleansing, transformation, and loading or subscribing to events. In this approach, if you want to add a new domain replication, then you need to change the whole set of replications, which leads to maintainability and testability problems.

The data ownership of today's monolithic data lake platform is based on who builds the data lake. In a nutshell, the ownership is based on technology and skills, not on the domain. The data mesh approach provides a solution to most of the problems you are facing with today's monolithic big data approach.

The following paragraphs explain the next-generation data lake implementation steps.

Step 1: Self-Service Data Infrastructure as a Platform

The principle of shifting the dataset ownership from the tool is specific to the domain. To support this approach, the data pipeline needs to move from the ingesting, cleansing, transforming, and subscribing approach to the domain-based approach.

For the domain-based approach, you need to split the replication pipeline based on domain, such as the customer pipeline, order pipeline, etc. In this split, the source database is required to own and take the responsibility for domain-based cleansing, deduplicating, and enriching of their domain events. Each domain dataset must establish service-level objectives for the quality of the data it provides.

For example, as shown in Figure 8-15, your customer domain provides customer demographic details. The "add product to wish list" domain can include cleansing and standardizing the data pipeline in the customer domain pipeline, which provides a stream of de-duped, near-real-time add product events. The aggregation of domains is responsible for the new data domains.

- Customer demographics + add the product to wish list = customer domain pipeline

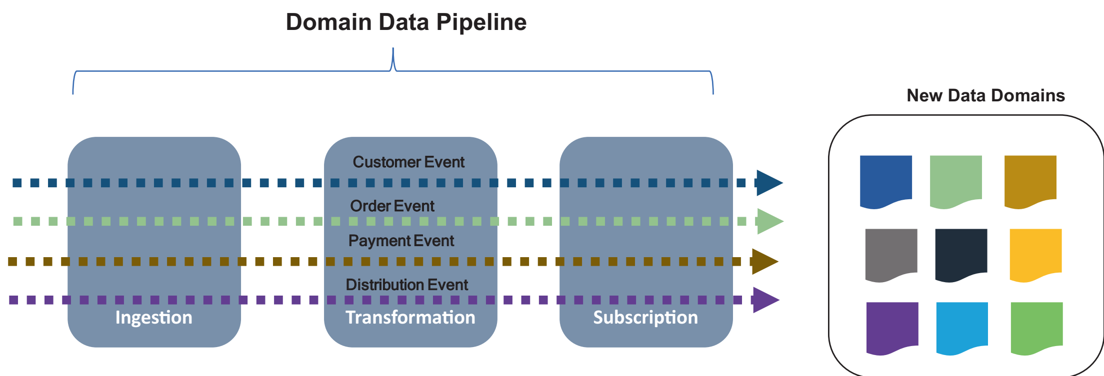

Figure 8-15. *Domain-based pipeline*

To summarize, the source side of the domain data pipeline has the responsibility to provide domain-related events, such as cleansing. The target side's responsibility is to subscribe to data, shown as New Data Domains.

Step 2: Data as a Product

Based on the previous step, the data ownership and data pipeline implementation are the responsibility of the business domain, as shown in Figure 8-16. This raises an important concern around the accessibility, usability, and harmonization of these new domain datasets.

Figure 8-16. *Data as a product with a "as-a-service" model*

This is where you can implement data domains as a service by creating domain capabilities as APIs and make them available to the rest of the consumers in an organization. As part of the as-a-service approach, you need to create a set of well-designed APIs and events with discoverable, well-documented, and well-tested sandboxes.

Step 3: Data Infrastructure as a Platform

The main concern of distributing the ownership of data to the domain is the duplicated effort and skills required to operate the data pipeline's technology stack and infrastructure in each domain. Harvesting and extracting domain-agnostic infrastructure capabilities into a data infrastructure platform duplicates the effort of creating a domain-related pipeline, storages, and domain-specific streaming engines. The data infrastructure as a platform should be domain agnostic and configure the platform to be domain specific.

To build the data infrastructure for data meshes, you can use the existing available infrastructure; for example, you can use AWS S3, Google Cloud Storage, or Azure Blob Storage to store domain models, and for the "as a service," you can use standard API stacks and event stacks. For the data pipeline, use event brokers and ETL tools to create a separate pipeline and codebase for each domain-related replication.

Step 4: Domain-Oriented Decentralized Data Ownership and Architecture

To decouple and decentralize the monolithic data platform, we need to start thinking from a data domain angle, instead of just replicating data from heterogeneous sources to target data. In my ecommerce example, the customer domain owns and serves the dataset for access to any team for any purpose. The physical location of the customer domain can be anywhere like Google Cloud storage or AWS S3 or Azure Blob storage on the respective cloud implementations, but the domain owner should be the same team that owns the overall customer domain in your enterprise.

The team that owns the customer domain is responsible not only for providing the business domains but also for the truths of customer demographics and their likes and dislikes of the products. The customer usage pattern is required for other transaction details that are related to other domains; in this case, you need to create a domain-specific data set that requires consumption.

Step 5: Data Governance

The data mesh platform should be designed with a distributed data architecture, under the centralized governance and standardization for interoperability, and enabled by a shared and self-service data infrastructure. Once the data infrastructure is matured, then you can apply a centralized with decentralized governance concept to improve the innovation, independence, etc.

Data Processing with Real-Time Streaming for Analytics

Big data architecture is designed to handle the processing and analysis of data. Over the years, the data processing landscape has changed, and the business dependency on data processing has grown dramatically. Every business in any industry is relying on data processing for key decisions and also to provide a better experience to their customers. Therefore, you can say that managing big data processing is becoming the main interest of the CIO office because there are business deadlines to meet.

In data processing, some data arrives in real time, and some arrives in a batch with large chunks. Figure 8-17 shows the classic data processing of any data. You can choose whichever option you want, either batch or stream processing, based on the requirements. Real-time processing requires qualities such as scalability, fault tolerance, predictability, and resiliency.

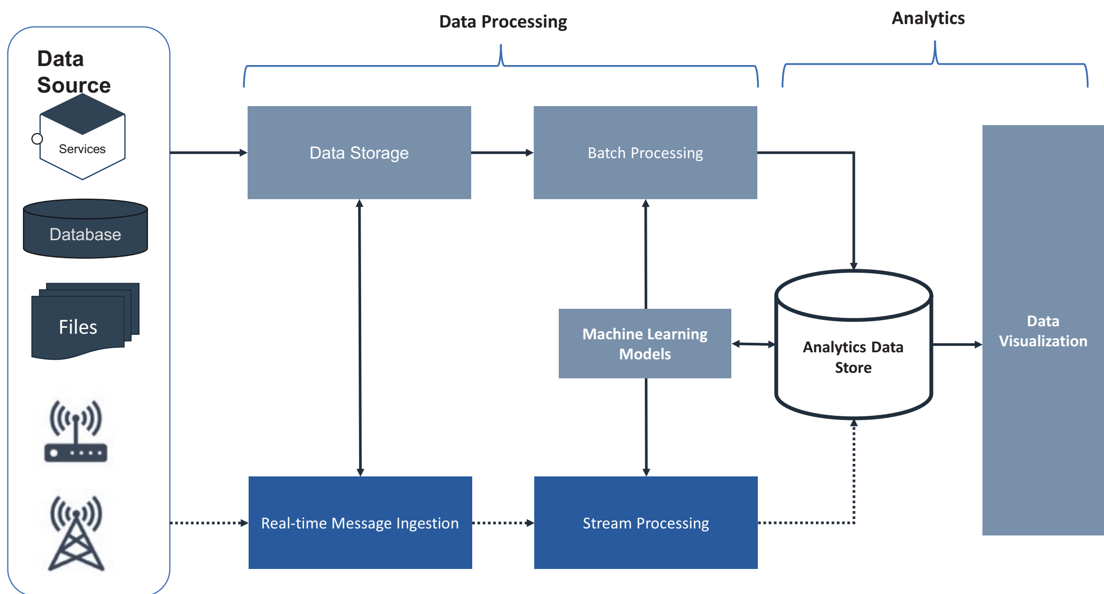

Figure 8-17. *Classic data processing*

The following are the main components of data processing for analytics platforms:

- *Batch processing of data source*: Processing of data files using long-running batch jobs

- *Real-time processing of data*: Processing of data in real-time stream processing
- *Machine learning models*: Applying various ML models on data analytics for predictive analysis
- *Proceed data storage*: Processed data storage for data visualization
- *Data visualization*: Generating various reports and dashboards for business and leadership

To support your organization's need for data analytics, you can choose from the following available industry architectures.

Lambda Architecture

The Lambda architecture is a reference architecture for scalable, fault-tolerant data processing and is designed to handle a big chunk of data by using both batch and stream processing methods. This reference architecture was first introduced by Nathan Marz. This architecture helps you to combine both traditional batch processing and stream processing pipelines. The Lambda architecture tries to solve the concerns around latency, data consistency, scalability, and fault tolerance.

In the Figure 8-18 reference architecture, the main components are data source, batch layer, serving layer, speed layer, and query.

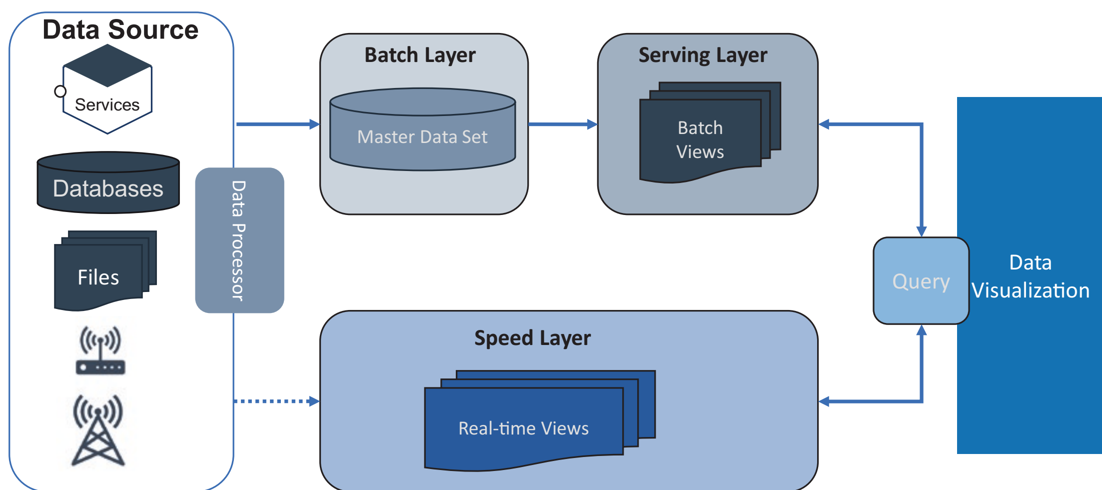

Figure 8-18. *Lambda architecture*

Data sources can be combined with various sources in an enterprise. This source can be designed by adopting ETL methods using streaming technologies. This data will be delivered simultaneously to both the batch and speed layers.

> *Batch layer*: The batch layer saves all the data coming into the system as batch views in preparation for indexing. The data is treated as immutable and append-only to ensure a trusted historical record of all incoming data. The objective is to maintain accuracy by being able to process all the available data when generating views. This layer can fix any errors if they occur by recomputing based on the data set; the output of this layer is stored in the read-only database. A technology like Apache Hadoop is often used as a system for ingesting the data as well as cost-effectively storing the data.
>
> *Serving layer*: The serving layer incrementally indexes the batch views to make a query by the data visualization. This layer can customize the indexes depending on the use cases. The objective of this layer is to make queries fast and serve them parallelly. While an indexing job in the service layer is for indexing data and service layer creates a new job for every new data processing.
>
> *Speed layer*: The speed layer processes data streams in real time and handles the data that has not already been delivered to the batch layer due to the latency of the batch layer. It also processes the latest data to provide a complete view of the data. Technology like Apache Stream, Flink, Spark streaming, etc., can be used to design a speed layer.

How Does the Lambda Architecture Work?

The batch and serving layers continue to index incoming batch data in batches. There will be latency in the indexing of all batches. The speed layer complements the batch and serving layer by indexing in real time all the new and also delayed batch indexes. Both the batch layer and speed layer collaborate to provide a large consistent view of data in the batch/serving layers that can be re-created at any time.

Once a batch indexing job completes the newly indexed data available for visualization, the speed layer's copy of the same data is no longer needed and is deleted from the speed layer. The serving layer processes the data that is already indexed by the speed layer.

Kappa Architecture

The Kappa architecture is a reference architecture for data processing for analytics and is used for processing streaming data. The reference architecture was introduced by Jay Kreps. The objective of this reference architecture is to process both real-time and batch processing for analytics, with a single technology stack. It is based on streaming immutable architecture in which data is stored in a database. The stream engine reads the data, transforms it in an analytical format, and finally stores it in analytical database for query and data visualization.

The Kappa architecture provides real-time analytics based on data availability. This helps the business team to reduce the decision time. It also supports historical analytics by reading the data stored in the data lake in the batch process. Kafka, AWS Kinesis, Azure Stream Analytics, Azure Event Hub, Google Pub/Sub, and Confluent are stream processing engines. For more information on the streaming, please refer Chapter 6.

The Kappa reference architecture shown in Figure 8-19 is considered simple compared to the Lambda architecture as it uses the same layers and technology stack for both streaming and batch processes. In a nutshell, the Kappa architecture is a simpler reference architecture for data processing.

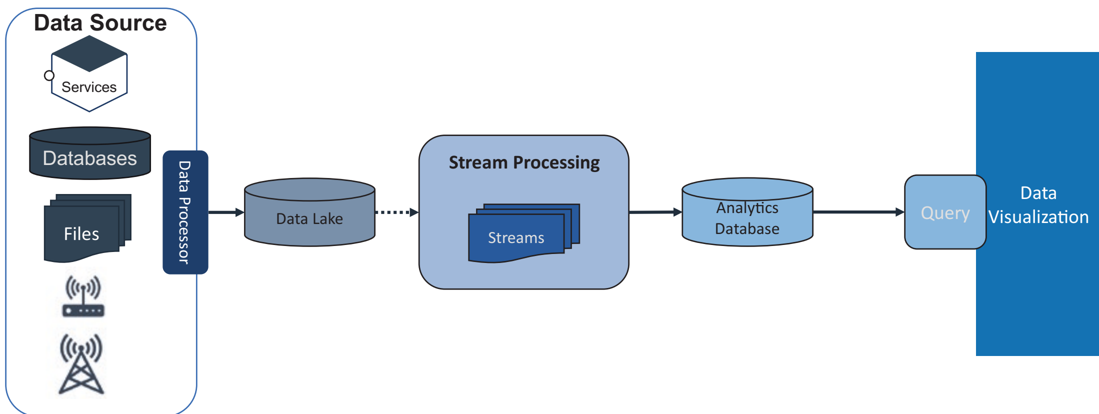

Figure 8-19. *Kappa architecture*

Microservices in Data Processing with Real-Time Streaming for Analytics

In the previous sections, I explained the real-time data processing reference architecture for the data analytics platform. You are already familiar with the microservices decentralized polyglot persistence principle. One challenge of dealing

with decentralized data in a microservices architecture is the need to collate data for analytics. A common way to approach this is through data movement, meaning aggregating the data into a centralized data lake by using the Kappa architecture to provide data visualization, as shown in Figure 8-20.

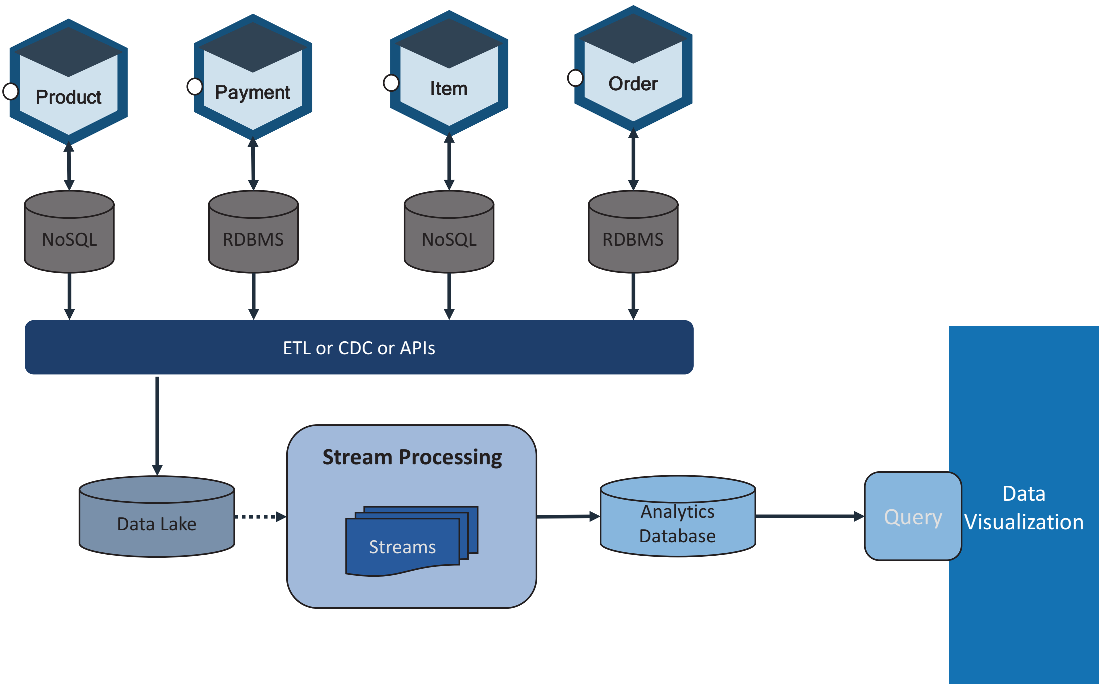

***Figure 8-20.** Polyglot persistence with Kappa architecture*

Both the service and data analytics team can collaborate with each other to replicate data from each service to the data lake through ETL, CDC, or APIs.

Mobile Platform Database

Mobile computing applications need to store information locally to make your applications more responsive and less dependent on network connectivity. The trend of offline usage, or less dependency on the network, is gaining popularity. The use cases are a list of contacts, price information, distance traveled, etc.

A mobile application keeps the database locally or makes a copy of the database over the cloud onto a local device and syncs with it as required, as shown in Figure 8-21. This will help create faster and more responsive applications that are functional even when there is no or limited back-end connectivity.

There are various mobile database providers such as Realm MongoDB, Couchbase Lite, SQLite, and Core Data. They support a lighter version of the database being installed as part of the mobile applications and to work on both iOS and Android.

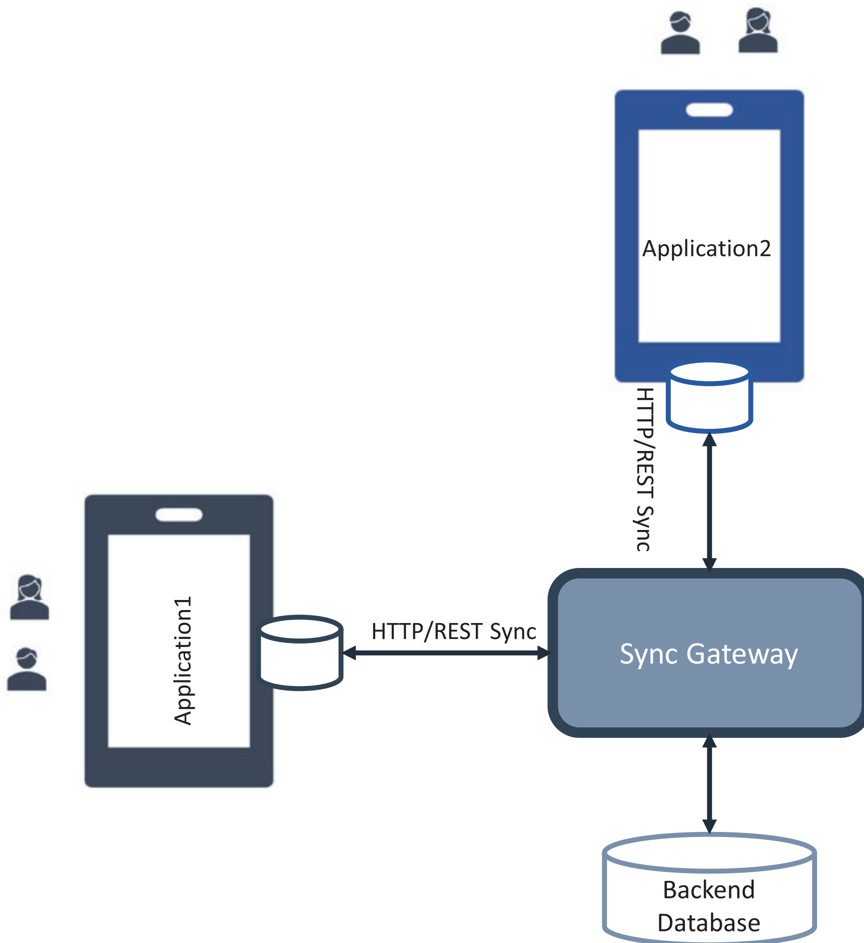

Figure 8-21. *Mobile database architecture*

The mobile databases must be installed along with your app, and they store all the data that is required to provide a customer experience on a slow network or offline. These databases will sync often to your back-end databases through sync gateways. The data synchronization is done via asynchronous data syncs and synchronous data syncs.

The asynchronous sync manages data events asynchronously without blocking any app functionality through reactive REST APIs.

In synchronous, the sync services are responsible for syncing data from a remote server to a mobile device and then storing the data locally in the mobile databases.

The data synchronization in the mobile application is achieved by using a sync service, sync adapter, and sync gateway. A sync adapter is a plugin that handles background syncs along with sync gateways.

A mobile database needs to have the following characteristics:

- Fast and secure
- Very lightweight
- Can work with low memory and power
- No server requirement
- Must work efficiently with mobile app code

There are various mobile databases available to choose from, such as Realm from MongoDB, Couchbase Lite from Couchbase, SQLite, and Core Data.

Intelligent Data Governance and Compliance in the Cloud Native World

Digital transformation in cloud native architectures is disrupting business. Along this journey, quality data is becoming an organization's most strategic asset for business decisions and better customer experiences that support business growth.

Why Data Governance?

With the exponential growth of data, a strict regulatory environment, and cyberthreats on the rise, protecting and extracting the value from your most strategic asset are imperative. These tasks are also a formidable challenge. The cost of failing to comply

with stringent regulatory requirements may be a legal battle. Regulations such as General Data Protection Regulation (GDPR), Securities and Exchange Commission (SEC) regulations, and the legislation and regulations of each country are outpacing the capabilities of existing IT infrastructure investment. Data complication further increases as the IDC predicts global data will grow to 163 zettabytes by 2025.

Data governance helps organizations better manage the availability, usability, integrity, and security of their enterprise data. The objective of data governance is not just to bring data at rest under control but also to know where data is located; how it originated; and who, where, and access to data. Effective data governance must be self-governed irrespective of which country is compliant.

In the modern digital economy, anyone can access data anywhere at any time, on any device. The CxO demands easy access to data with tight regulations with the best-in-class compliance process. To satisfy these regulations, you need more than just strong governance; you need governance based on the data analytics with intelligence embedded.

What Is Data Governance?

Data governance helps you to better manage the availability, usability, integrity, and security of your enterprise data. Data governance moves beyond information management to support business processes and encompasses a broad set of data strategies and functions including the following:

- *Data delivery and access*: Any actions related to storing, retrieving, and acting on data.
- *Data integrity*: Ensuring the veracity, accuracy, and quality of data.
- *Data lineage*: Managing the movement of data.
- *Data loss prevention (DLP)*: Ensuring sensitive data isn't sent outside your organization's network and controlling what data can be transferred.
- *Data security*: Protecting unauthorized access or data corruption.
- *Data synchronization*: Ensuring data consistency.

- *Master data management (MDM)*: The complete collection of process, policies, standards, and tools for defining governance and managing data.
- *Data profile*: Reviewing the source data and understanding the structure, content, and interrelationships.
- *Data quality*: Measuring the condition of data based on factors such as consistency, completeness, accuracy, and reliability.
- *Data standardization*: Bringing data into a common format that allows for further analysis.
- *Data General Data Protection Regulations (GDPR)*: This is a privacy and security law that states that personal data is any information that is related to an identified or identifiable natural person.

Governance Framework

Figure 8-22 illustrates the overall framework for intelligence data governance. This framework is based on these five pillars:

- Change management
- Intelligent tooling
- Secure
- Decentralize
- Operating model

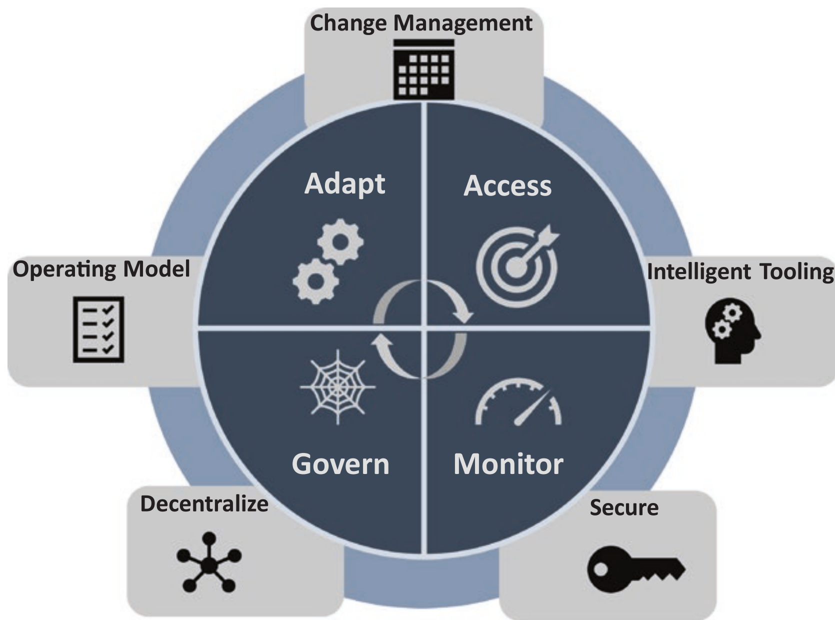

Figure 8-22. *Governance framework*

Change Management

Change management is the approach to planning, designing, and implementing data governance without any unintended disruption of the business. As part of the change management plan, the following key practices need to be adopted:

- *Leadership engagement*: Enabling leaders and sponsors to champion the transition.
- *Communication and stakeholder management*: Information, announcements, and updates through various channels; the updates include where and how the changes are impacting the organization.
- *Training and performance support*: Data governance process, policy, roles, and competency training.
- *Organization alignment*: Recommendations for new roles, performance measures, responsibilities, and workgroup structures.
- *Measurement and readiness*: Preparing the business and measuring its readiness to adopt the changes.

Intelligent Tooling

In intelligent tooling, you need to adopt best-in-class technology and accelerate business value from data assets. The following are the different tooling strategies that need to be adopted for data intelligence governance:

- *Rapid discovery and recognition*: Rapid discovery and recognition of personal and sensitive data across the ecosystem
- *Smart tagging of metadata and lineage*: Smart recommendations for business tagging of technical metadata and lineage using multi-metadata stores
- *Intelligent data quality rule recommendations*: It is based on the corpus and usage of ML models
- *Auto-remediation of data*: Learning from data curation actions and auto-remediation suggestions
- *Intelligent workflow triggers*: Automated workflow triggers based on user behavior

Operating Model

In the operating model, you need to manage the roles, responsibilities, processes, policies, and standards required to manage and govern the data ecosystem. In the operating model, you need the following teams:

- *Executive governance council*: This council is the ultimate authority in defining program-level scope, arbitrating escalated resolutions, and approving data governance strategy with a centralized and decentralization approach.
- *Business data owners*: The owners play a leadership role in championing data management and data governance efforts.
- *Data governance council*: The cross-functional and cross-entity leadership team provides direction and oversight to the overall data governance structure.
- *Data governance organization*: This organization provides overall non-IT support to the council.

Decentralization

In the decentralization approach, this framework embraces each portfolio in an organization to set its subgovernance under the guidance of the central governance framework. This helps an organization to decentralize the responsibility and accountability and helps to fasten the decisions. Each portfolio follows the same tooling and structure as central governance and tweaks it based on the nature of data.

Secure

The security of data is of utmost importance. As mentioned, you need to have a set of country-specific compliances in place and always conduct an audit across the organization.

You need to consider the following points when you execute this framework:

- Data governance should be viewed as an ongoing program, not as just a project.
- Data governance must have executive sponsorship, and they must take significant ownership of the initiative.
- Data governance councils must have real authority to resolve overall organization issues so the portfolio governance council can resolve the portfolio issues.
- There should be a clearly defined set of data governance and quality metrics published regularly and reviewed regularly.
- There must be a clear and timely communication method for data governance initiatives.

You must train your team regularly.

Summary

The cloud has made a big impact on how we work today, including with data. The cost of storing data has been significantly reduced; it is now cheaper and more feasible for companies to keep vast amounts of data. The operationalization of data has reduced significantly due to managed services and serverless data storage; this has made it easier to spread data across different storage types.

In this chapter, I explained five main requirements of your data layer. The first is how to choose the database based on the use cases, the second is how to replicate the data, the third is to decouple the data lake to a data mesh, the fourth is data for analytics, and finally the fifth is the governance model.

In the cloud native world, one thing you cannot forget is the data layer. To deliver a consistently fast, satisfying customer experience, the data layer must also be modernized along with your application. You must embrace all five requirements of data modernization.

Although there are many reasons to adopt governance approach, it enables data accessibility, data confidence and understanding, and data activation. Some of the benefits are as follows:

- Data consistency ensures completeness and accuracy.
- Proactive data quality checks ensure data alignment.
- It removes confusion over the data meaning.
- You can make fact-based decisions in real time with accurate data.