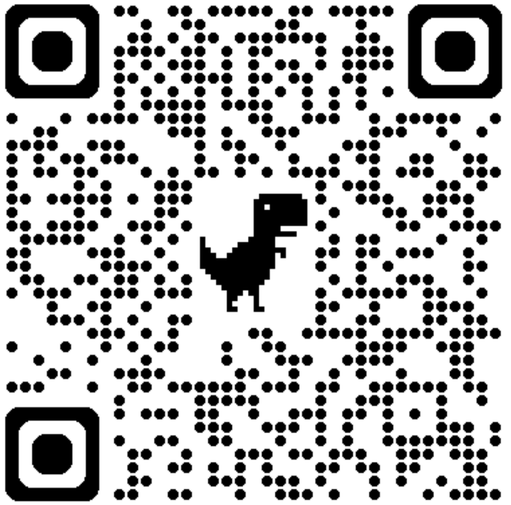
Architectural Patterns Focus: Data

Daniel Hinojosa

Design Pattern Data Standards

In this Presentation

- CQRS
- Materialized View
- Warehousing, Lakes, Streams, and other bodies of water
- OLAP vs. OLTP
- Pinot, Kafka, Spark, etc.
- Business Intelligence
- Making Data Available for ML/AI

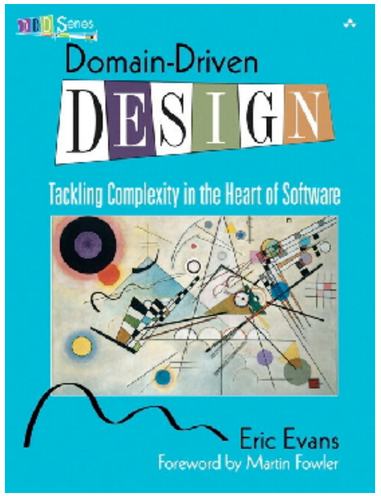


Slides and Material: https://github.com/dhinojosa/nfjs-architectural-patterns-data

Domain Driven Design

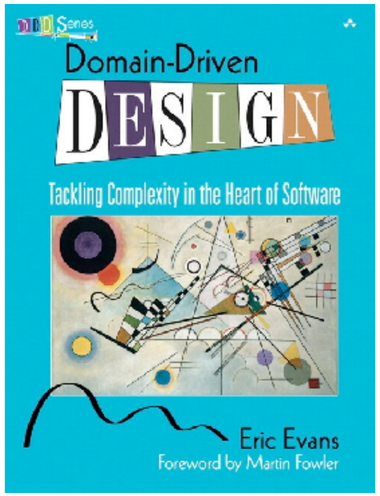
Domain Driven Design

- Methodology focused about the important of the domain
- Build an *ubiquitous language* for domain
- Classifies Objects into Entities, Aggregates, Value Objects, Domain Events, and Service Objects
- One of the other important aspects to DDD is the notion of a bounded context and subdomains



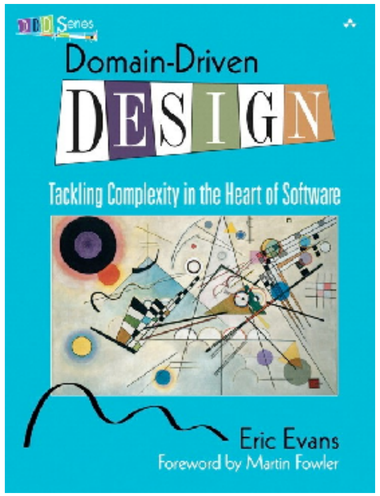
Ubiquitous Language

- Cornerstone of Domain Driven Design
- A dictionary of the same language
- A language for describing your business domain
- Represents both the business domain and the domain experts' mental models.



Bounded Contexts

- Divide the Ubiquitous Language into multiple smaller languages
- Assign each team to the explicit context in which it can be applied: its bounded context



What goes in the bounded context?

- Developers with then develop the following software artifacts:
  - Value Objects
  - Entities
  - Aggregates
  - Domain Services
  - Application Services

Value Objects

- Object that can be identified by the composition of its values
- No explicit identification is required
- Changing the attributes of any of the fields yields a different object
- Value Objects can be used to counter the “Primitive Obsession” code smell
- Typically immutable

```
class Color {
   int red;
   int green;
   int blue;
}
```

Entities

- Opposite of a value object and requires explicit identification
- Explicit identification is required
- For example, an Employee, just identified by an employee's first name and last name
- Entities are subject to change

```
class Employee {
   private Name name;
   //Constructors, equals, hashCode, etc.
}
```

```
class Employee {
   private EmployeeId employeeId;
   private Name name;
   //Constructors, equals, hashCode, etc.
}
```

Aggregates

- An entity that manages a cluster of domain objects as a single unit
- Consider an `Order` to `OrderLineItem` relationship
- The root entity ensures the integrity of its parts
- Transactions should not cross aggregate boundaries
- They are domain objects(order, playlist), they are not collections, like `List`, `Set`, `Map`

```
class Album {
   private Name name;
   private List<Track> tracks;

   public void addTrack(Track track) {
      this.tracks.add(track);
   }

   public List<Track> getTracks() {
      return Collections.copy(tracks)
   }
}
```

Domain Services

- Stateless object that implements the business logic
- It naturally doesn’t belong to any of the domain model’s aggregates or value objects

```
class OrderTax {
   //no state
   private void calculateTaxWithRate(Order
         order, TaxRate taxRate) {
       //...
   }
}
```

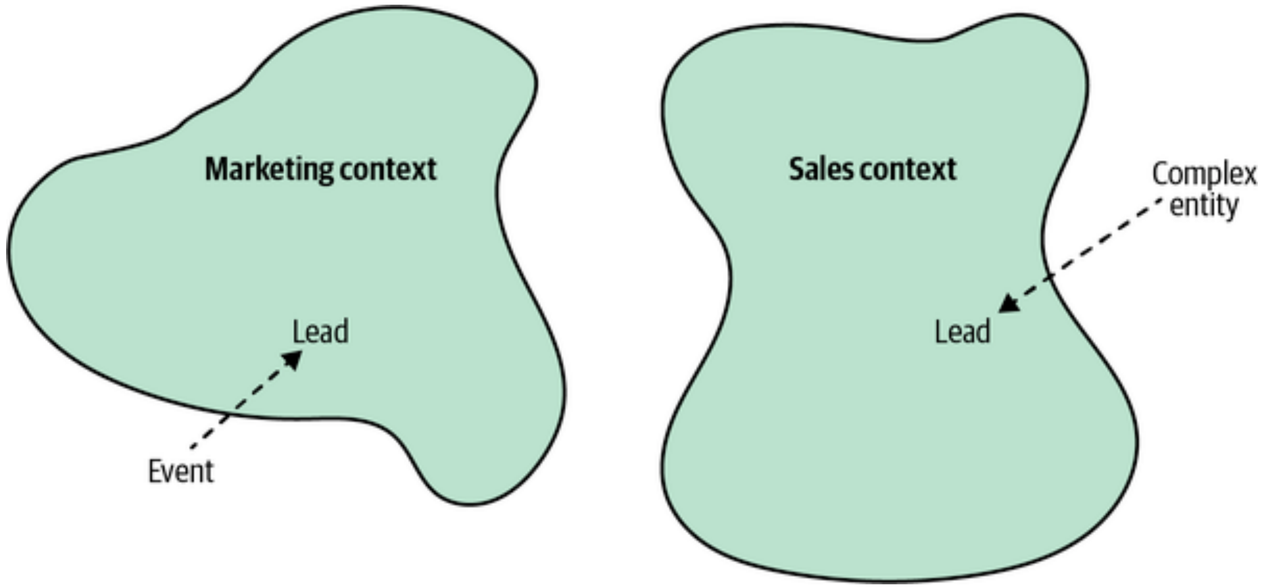
Application Services

- Makes use of Repositories (Storage Abstractions)
- Aggregate instances and then sent for transforming to a transformed object
- The transformed object will typically be routed to the UI

```
class OrderService {
   private OrderRepository
               orderRepository;
   private void persistOrder(Order) {
       //...
   }
}
```

Bounded Context

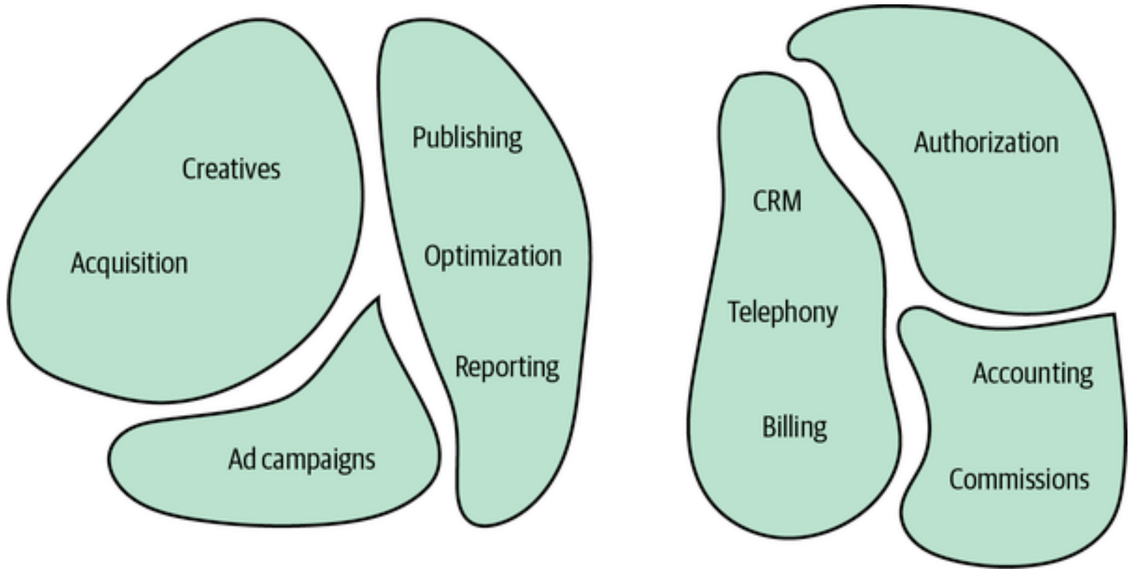
The Diagram



Learning Domain Driven Design - Vlad Khononov

The Diagram

Contexts are neither big nor small, but useful



Learning Domain Driven Design - Vlad Khononov

Multitenancy

The Problem

- **Inefficiency in Resource Use:** Dedicated infrastructure per customer increases costs and complexity.
- **Scalability Limitations:** Managing isolated deployments for every user doesn’t scale.
- **Complex Maintenance:** Updates, fixes, and monitoring become cumbersome with isolated systems.
- **Delayed Time-to-Market:** Building and provisioning environments for each tenant takes time.

What’s a Tenant?

- **A tenant** can be an organization, team, user group, or individual user.
- Each tenant is logically separate from others in terms of functionality and data.
- While tenants share the same underlying hardware or application instance, they experience the service as if they had their own dedicated resources.

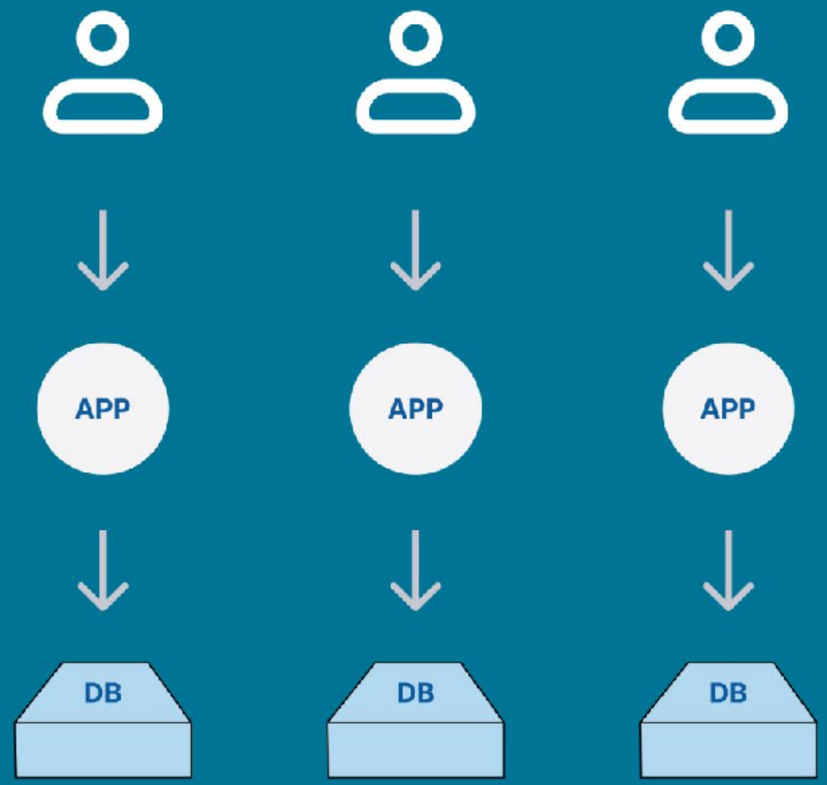
What are examples of a tenant?

- A business customer using a SaaS platform like Salesforce.
- A development team running isolated namespaces in Kubernetes.
- A merchant managing their storefront in Shopify.
- A consumer group in Apache Kafka accessing specific topics.

The Solution

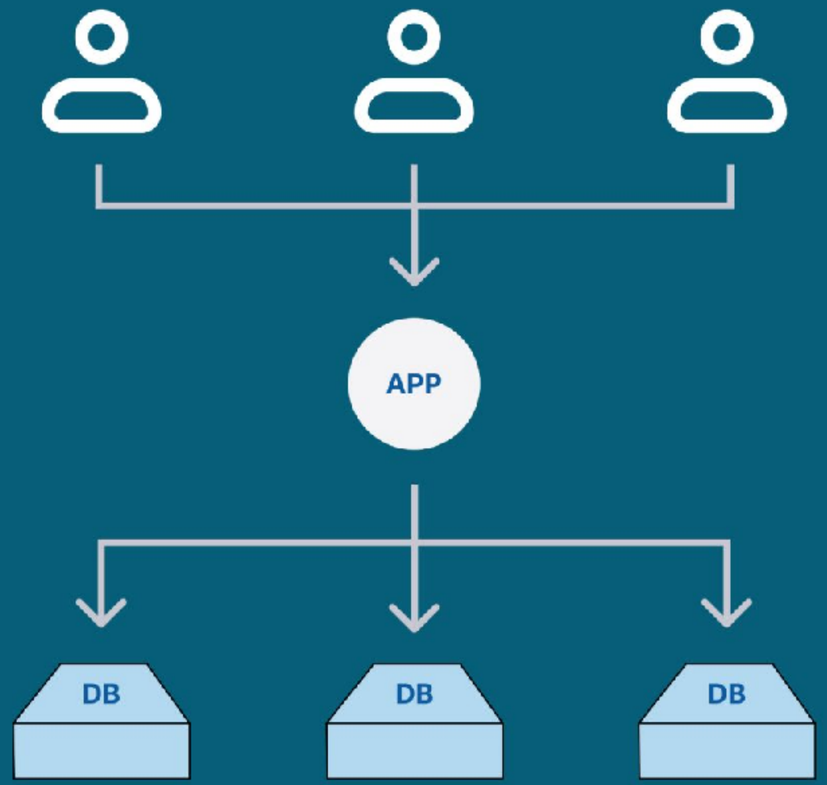
- **Multi-tenancy** is an architectural pattern where a single instance of software serves multiple customers (tenants), each with its own isolated data.
  - Shared infrastructure and resources.
  - Tenant-specific data segregation.
  - Lower cost because infrastructure is shared

Single -Tenant



VS

Multi-Tenant



https://ralabs.org/blog/multi-tenancy-explained-benefits-challenges-real-world-examples/

The Technology

- **Cloud Platforms:** Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure
- **SaaS Platforms:** Salesforce, Slack, Zoom, Microsoft 365
- **Databases:** PostgreSQL (tenant-specific schemas), MongoDB Atlas
- **DevOps Tools:** Argo CD (namespace-based isolation), Jenkins (folders and credential isolation)
- **Messaging Systems:** Apache Kafka (topic isolation, ACLs, quotas)
- **E-commerce Platforms:** Shopify (multi-tenant storefronts for merchants)
- **Analytics System:** Apache Pinot

OLTP vs OLAP

OLTP vs OLAP

- Online analytical processing (OLAP) and online transaction processing (OLTP) are data processing systems that help you store and analyze business data.
- OLAP combines and groups the data so you can analyze it from different points of view
- OLTP stores and updates transactional data reliably and efficiently in high volumes
- OLTP databases can be one among several data sources for an OLAP system.

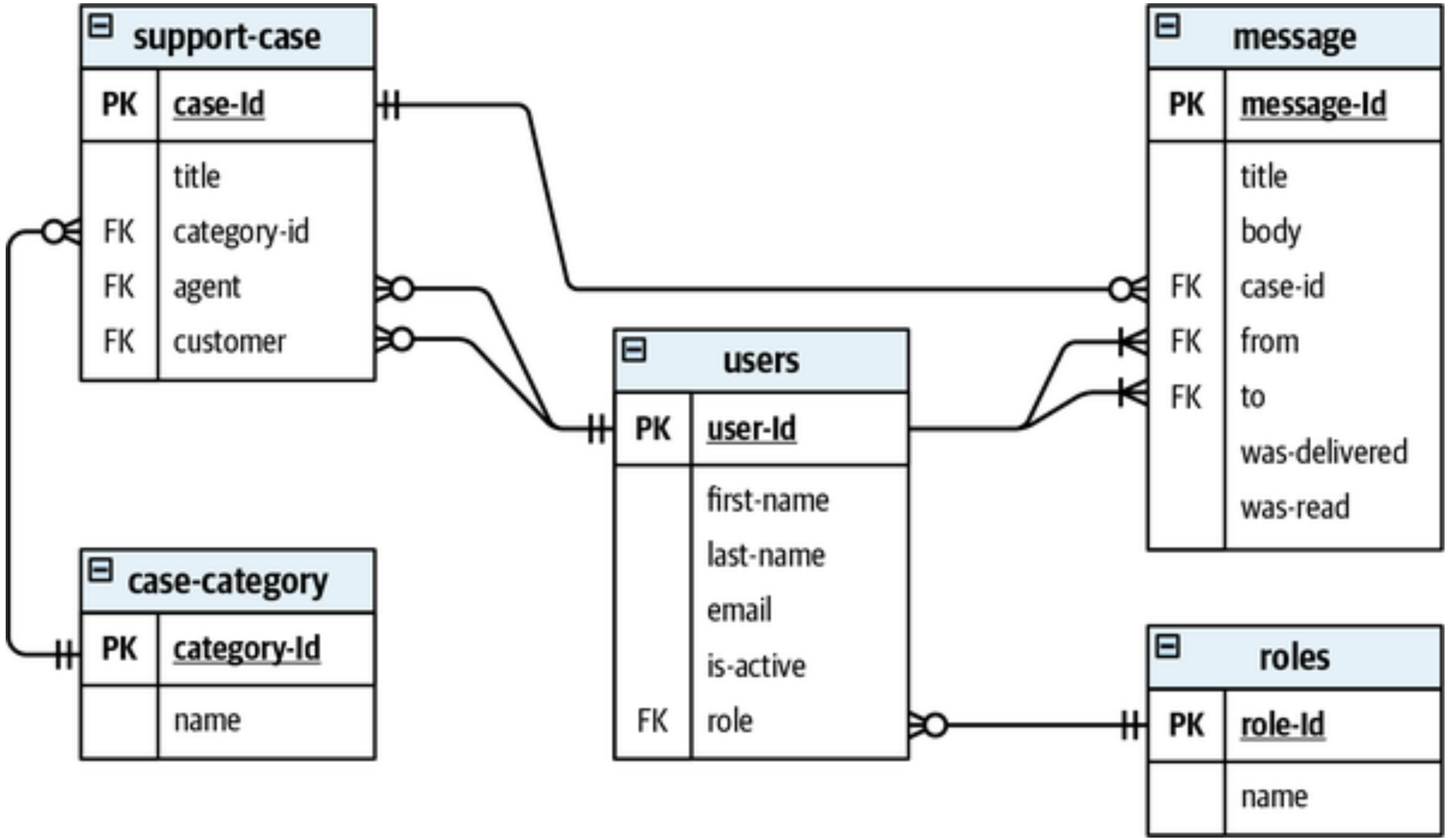
What is the difference between OLTP and OLAP

OLTP vs OLAP

OLAP vs OLTP

Aspect	OLAP (Analytical)	OLTP (Transactional)
Purpose	Analytics, reporting, decision-making	Real-time transaction processing
Query Complexity	Complex, multi-dimensional queries	Simple, fast queries
Data Type	Historical, aggregated, and derived data	Current, detailed, and transactional data
Operations	Read-intensive	Read and write-heavy
Schema Design	Star or snowflake schema	Normalized schema (3NF or higher)
Performance Goal	Optimized for query speed	Optimized for transaction speed
Concurrency	Few concurrent users	Many concurrent users
Example Use Cases	BI dashboards, trend analysis, forecasting	Order processing, inventory tracking
Examples	Snowflake, Google BigQuery, Tableau	MySQL, PostgreSQL, Oracle DB

OLTP



Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy - Vlad Khononov

Analytical OLAP Models

- Analytical models are designed to provide different insights into the operational systems.

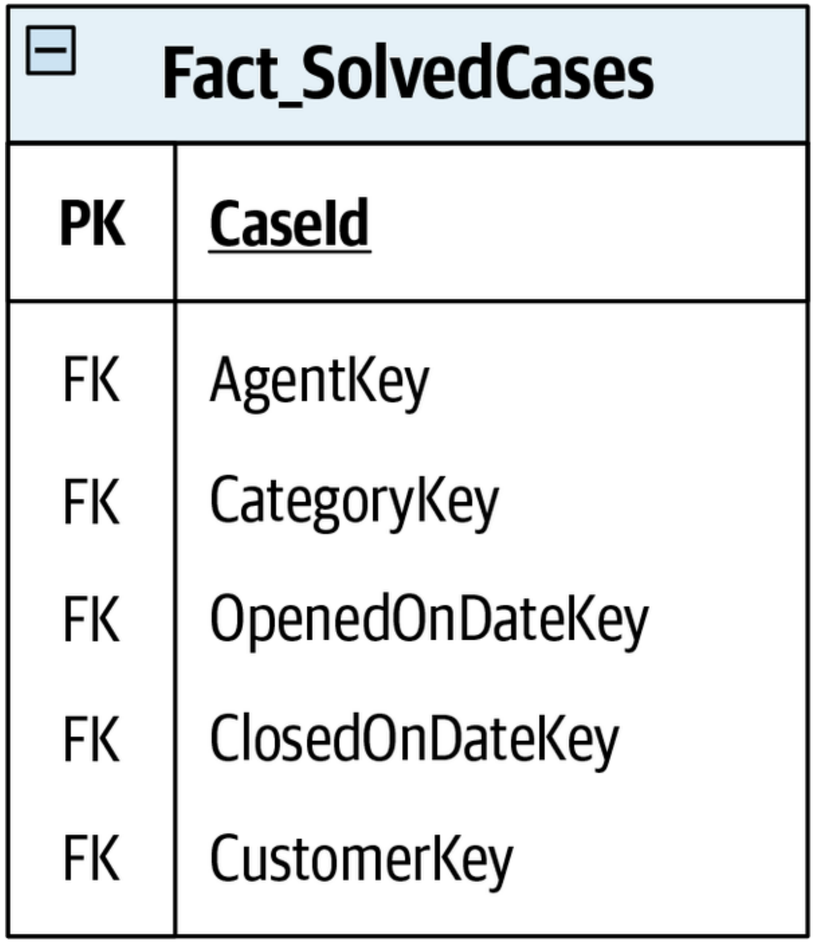
- Instead of implementing real-time transactions, an analytical model aims to provide insights into the **performance of business activities** and, more importantly, how the business can **optimize its operations to achieve greater value**.

- From a data structure perspective, **OLAP models ignore the individual business entities and instead focus on business activities by modeling fact tables and dimension tables**. We'll take a closer look at each of these tables next.

Star Schema

Fact Tables

- Facts represent business activities that have already happened.
- Facts are similar to the notion of domain events in the sense that both describe things that happened in the past.
- However, contrary to domain events, there is no stylistic requirement to name facts as verbs in the past tense. Still, facts represent activities of business processes.



Fact Tables

- Fact records are never deleted or modified: Analytical data is append-only data
- The only way to express that current data is outdated is to append a new record with the current state.

Fact_CaseStatus					
CaseId	Timestamp	AgentKey	CategoryKey	CustomerKey	StatusKey
case-141408202228	2021-06-15 10:30:00		12	10060512	1
case-141408202228	15/06/2021 11:00:00	285889	12	10060512	2
case-141408202228	15/06/2021 11:30:00	285889	12	10060512	2
case-141408202228	15/06/2021 12:00:00	285889	12	10060512	3
case-141408202228	15/06/2021 12:30:00	285889	12	10060512	2
case-141408202228	15/06/2021 13:00:00	285889	12	10060512	4

Dimension Table

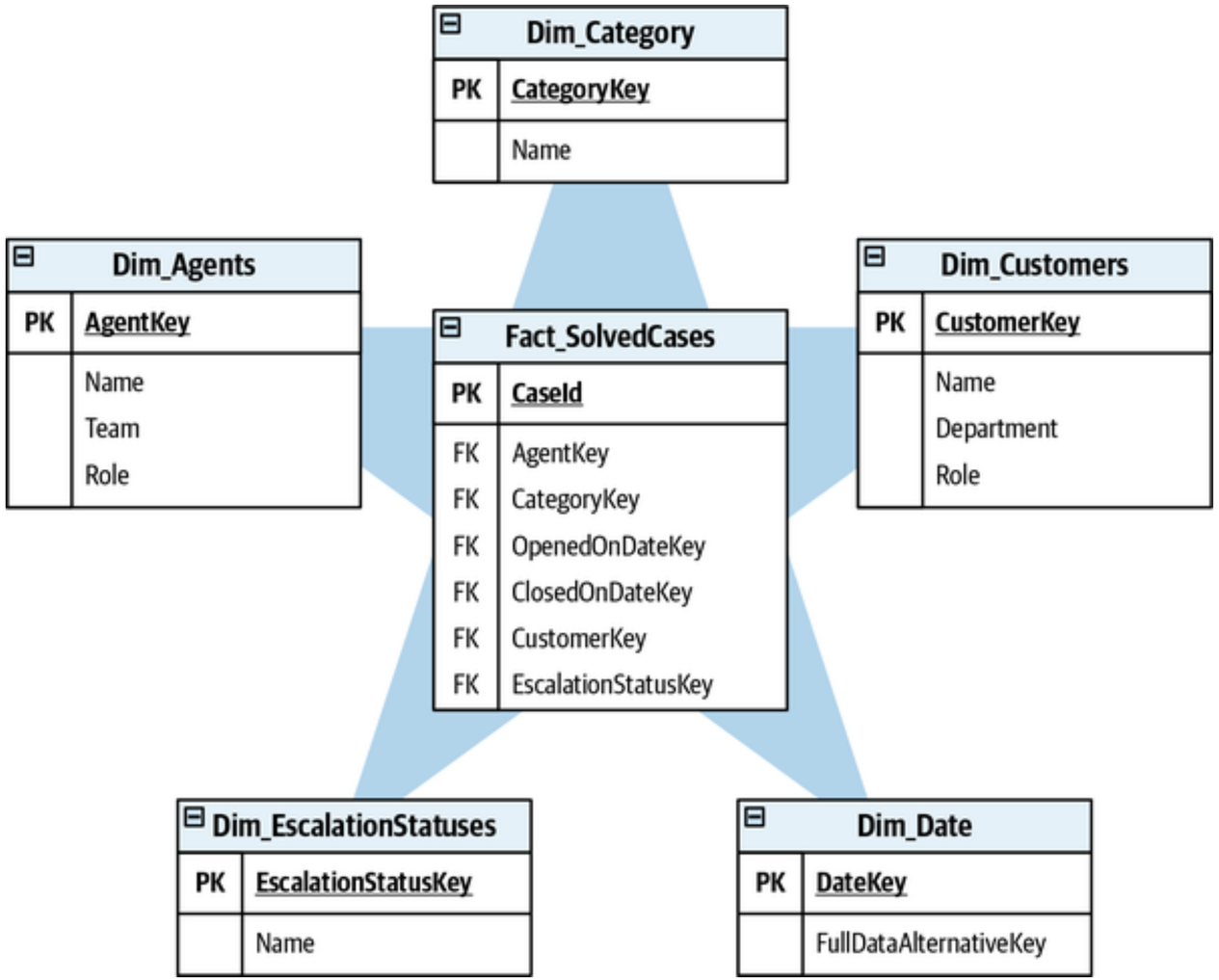
- Another essential building block of an analytical model is a *dimension*.
- If a fact represents a business process or action (a verb), a dimension describes the fact (an adjective).
- The dimensions are designed to describe the facts' attributes and are referenced as a foreign key from a fact table to a dimension table.
- The attributes modeled as dimensions are any measurements or data that is repeated across different fact records and cannot fit in a single column.
- Contain **descriptive attributes** (e.g., product name, date, region) that provide context for the facts.

Star Schema Example

- For a retail business:
  - **Fact Table**:
    - Sales (columns: `SalesID`, `DateID`, `ProductID`, `StoreID`, `SalesAmount`)
  - **Dimension Tables**:
    - Date (columns: `DateID`, `Year`, `Month`, `Day`, `Weekday`)
    - Product (columns: `ProductID`, `ProductName`, `Category`, `Price`)
    - Store (columns: `StoreID`, `StoreName`, `Location`, `Manager`)

Star Schema

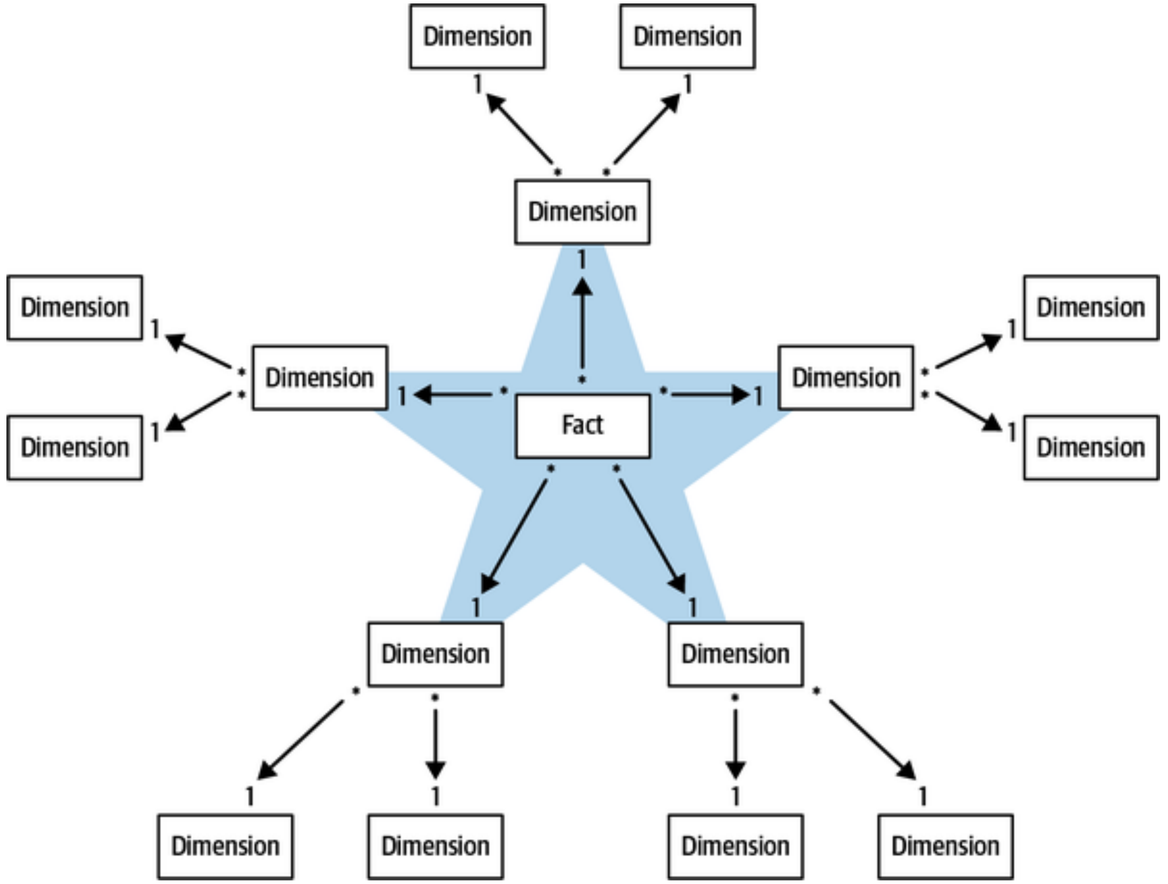
- Each Dimension Table is meant to expound up the fact table
- The querying patterns of the analytical models are not predictable. The data analysts need flexible ways of looking at the data, and it's hard to predict what queries will be executed in the future.



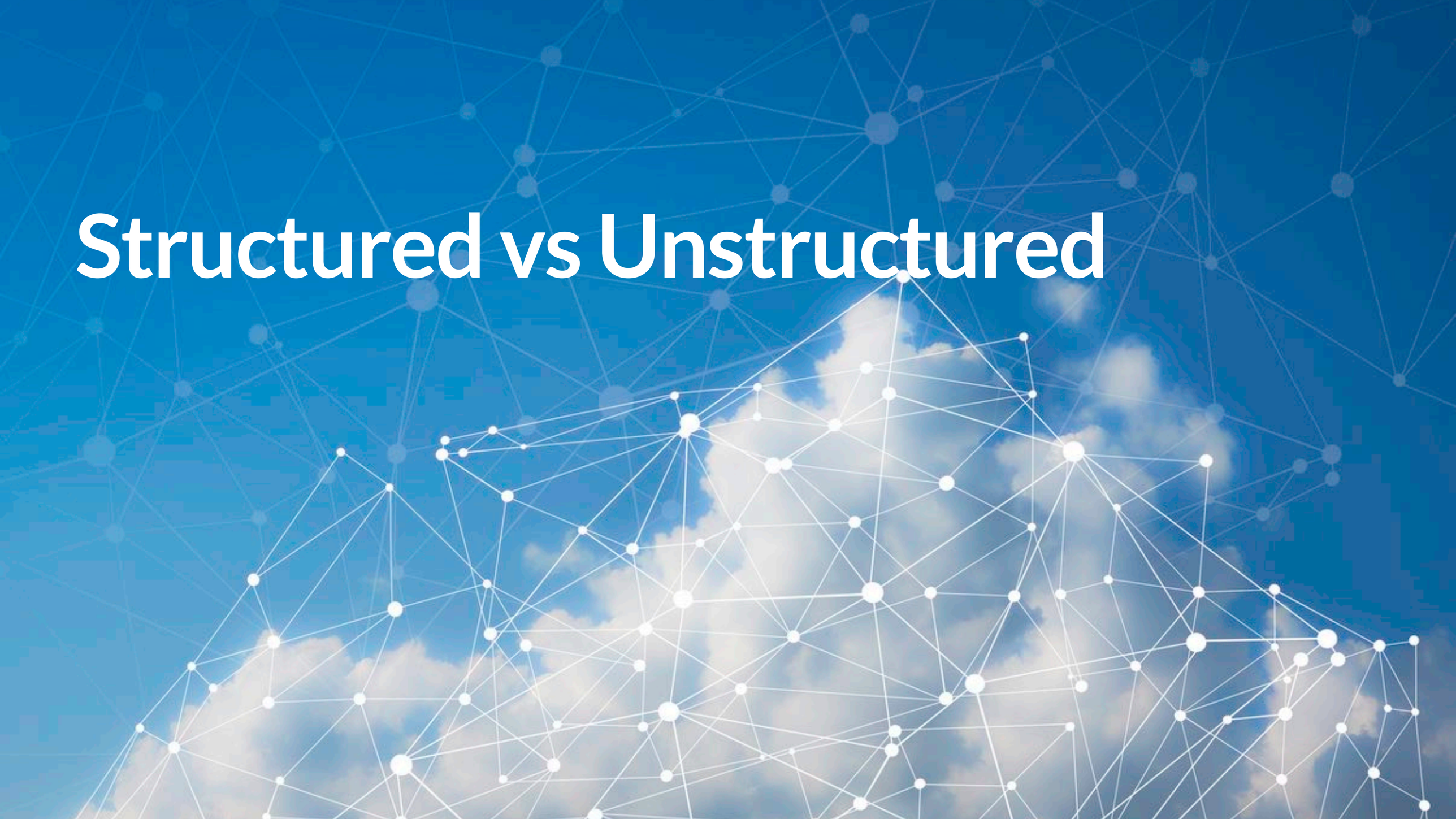
Snowflake Schema

Snowflake Schema

- The snowflake schema is based on the same building blocks: facts and dimensions.
- However, in the snowflake schema, the dimensions *are multilevel*: each dimension is further normalized into more fine-grained dimensions
- Both the star and snowflake schemas allow data analysts to analyze business performance, gaining insights into what can be optimized and built into business intelligence (BI) reports.



Structured vs Unstructured



Structured vs Unstructured Data

- **Structured:**
  - Data that conforms to a predefined schema or structure, making it easily searchable and queryable.
  - The schema defines the types, relationships, and constraints of the data.
  - Can exist in non-tabular formats like JSON, XML, or Avro if a schema is defined.
- **Unstructured Data:**
  - Data that lacks a predefined schema or consistent organization.
  - Examples: Text files, images, videos, audio files, and logs without consistent structure.

Structured vs Unstructured Data

Unstructured Logging

```
6:01:00 accepted connection on port 80 from 10.0.0.3:63349
6:01:03 basic authentication accepted for user foo
6:01:15 processing request for /super/slow/server
6:01:18 request succeeded, sent response code 200
6:01:19 closed connection to 10.0.0.3:63349
```

Structured Logging

```
time="6:01:00" msg="accepted connection" port="80" authority="10.0.0.3:63349"
time="6:01:03" msg="basic authentication accepted" user="foo"
time="6:01:15" msg="processing request" path="/super/slow/server"
time="6:01:18" msg="sent response code" status="200"
time="6:01:19" msg="closed connection" authority="10.0.0.3:63349"
```

Structured vs Unstructured Data

Structured vs. Unstructured

Aspect	Structured Data	Unstructured Data
Storage	SQL/NoSQL databases (e.g., MySQL, JSON, Parquet)	Object stores or file systems (e.g., S3, HDFS)
Querying	SQL or tools like Presto/Spark with schema	Advanced NLP, computer vision, or ML
Format	Tabular, JSON, XML with schema	Free-form (e.g., raw text, videos, images)
Schema Dependency	Schema-defined before or during use	No schema or schema inferred dynamically
Cost to Store	Higher per unit due to indexing, schema enforcement, and redundancy requirements	Lower per unit because raw data requires minimal processing or indexing

Backward vs Forward Compatibility

Backward vs Forward Compatibility

- **Backward Compatibility:**
  - Ensures that new versions of a system or API can handle data or requests from older versions.
  - **Key Goal:** Allow older clients to work seamlessly with newer versions.
  - **Example:** A new database schema still supports queries written for the old schema.
- **Forward Compatibility:**
  - Ensures that older versions of a system or API can handle data or requests from newer versions, typically by ignoring unknown fields or extensions.
  - **Key Goal:** Allow older systems to understand or tolerate future changes.
  - **Example:** A legacy system processes JSON with extra fields added by a new client.
- **Both Compatibility:**
  - A system that maintains both backward and forward compatibility, enabling robust interoperability between versions in both directions.

Backward vs Forward Compatibility

1980s	1990s	Today
• company name • contact person • physical address • phone • fax number	• company name • contact person • physical address • phone • fax number • **email address**	• company name • contact person • physical address • phone • ~~fax number~~ • email address • **social media**

Transitive Compatibility

- **Transitive compatibility** ensures that multiple versions of a system can interoperate through a chain of compatibility.
- If version **N** is compatible with **N-1**, and **N+1** is compatible with **N**, then **N+1** is transitively compatible with **N-1**.
- **Forward Transitive:** Older systems or APIs (e.g., **N-1**) can interact with multiple newer versions (e.g., **N** and **N+1**) by ignoring unknown fields or gracefully handling new functionality.
- **Backward Transitive:** A newer system or API (e.g., **N+1**) can interact with multiple older versions (e.g., **N** and **N-1**) by preserving legacy functionality.

Data Warehousing

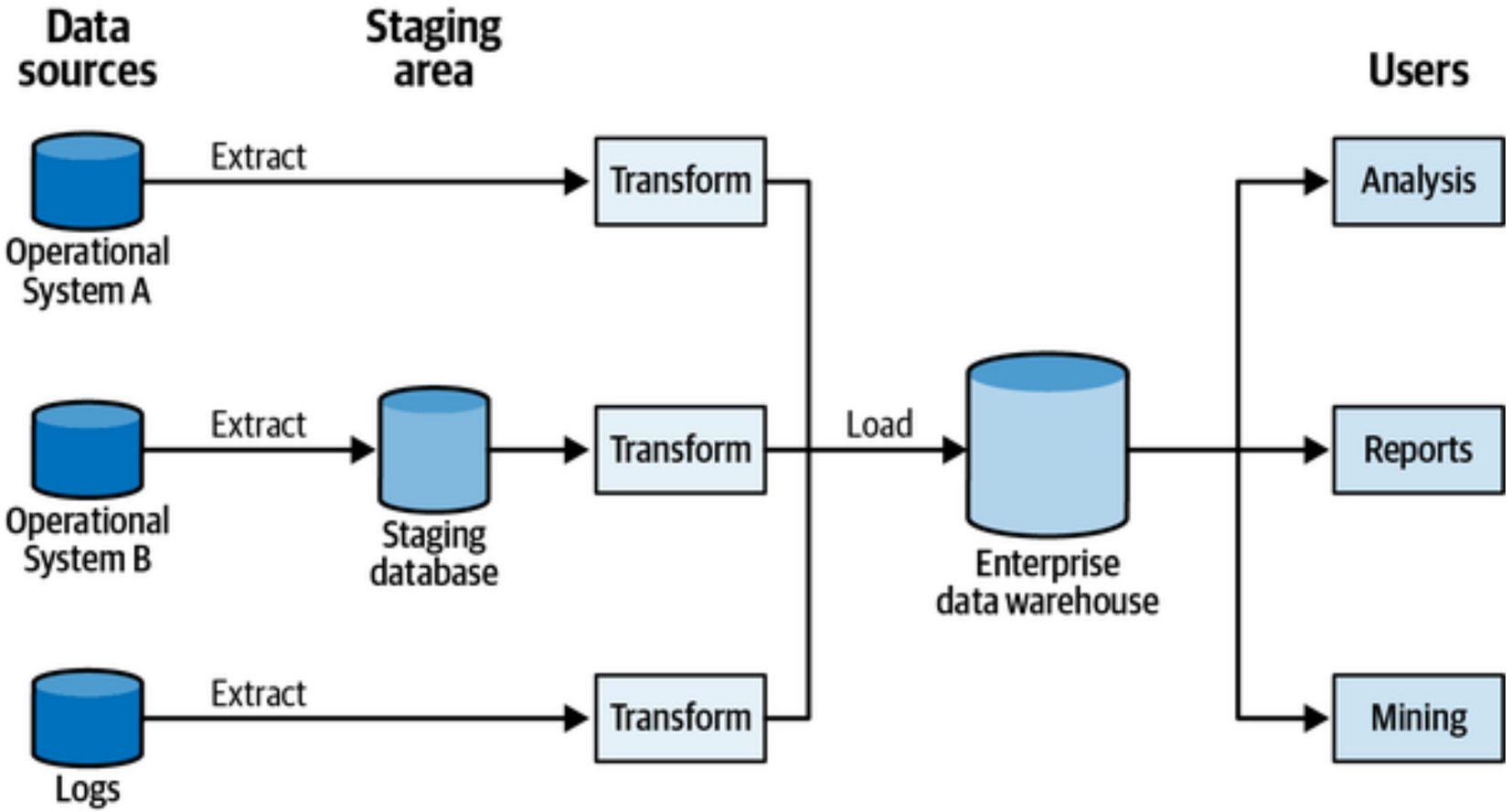
The Problem

- **Data Silos:** Business units (e.g. Microservices) maintain separate databases, leading to fragmented insights.
- **Complex Queries:** Operational databases are optimized for transactions, not analytics, making complex queries slow and resource-intensive.
- **Data Inconsistency:** Different systems use varying formats and standards, complicating analysis.
- **Scaling Issues:** Traditional OLTP systems struggle with high-volume analytical workloads.
- **Decision Delays:** Lack of a unified view slows down decision-making processes.

The Solution

- A Data Warehouse, a centralized repository for *structured data*, designed for querying and analysis,
- Consolidates data from multiple sources.
- **Schema-on-Write: Data schema is applied when stored.**
- Optimized for read-heavy analytical workloads.
- Supports historical analysis through snapshot storage.
- Single source of truth for analytics.
- Enables complex reporting and BI tools.
- Provides pre-aggregated, summarized data for fast querying.

The Diagram



Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy - Vlad Khononov

The Tradeoffs

- **Advantages:**
  - High-performance analytics on large datasets.
  - Consolidates disparate data for unified insights.
  - Enables advanced BI and dashboarding.
  - Historical analysis for trend detection.
- **Disadvantages:**
  - Expensive to build and maintain.
  - ETL (Extract, Transform, Load)/ELT (Extract, Load, Transform) processes introduce latency for near-real-time needs.
  - Requires significant upfront design and ongoing governance.
  - Scaling can be challenging with traditional architectures.

The Technologies

- **Technologies Implementing the Pattern**:
  - **Amazon Redshift:** Managed cloud warehouse, scales with workloads.
  - **Google BigQuery:** Serverless, highly scalable analytics warehouse.
  - **Snowflake:** Cloud-native with separation of compute and storage.
  - **Azure Synapse Analytics:** Unified analytics service.
- **Build your own:**
  - **Data Sources** (e.g., operational databases, logs, CRM, ERP).
  - **ETL (Extract, Transform, Load)/ELT (Extract, Load, Transform) Processes.**
  - **Centralized Data Warehouse.**
  - **BI and Analytics Tools consuming data for insights.**

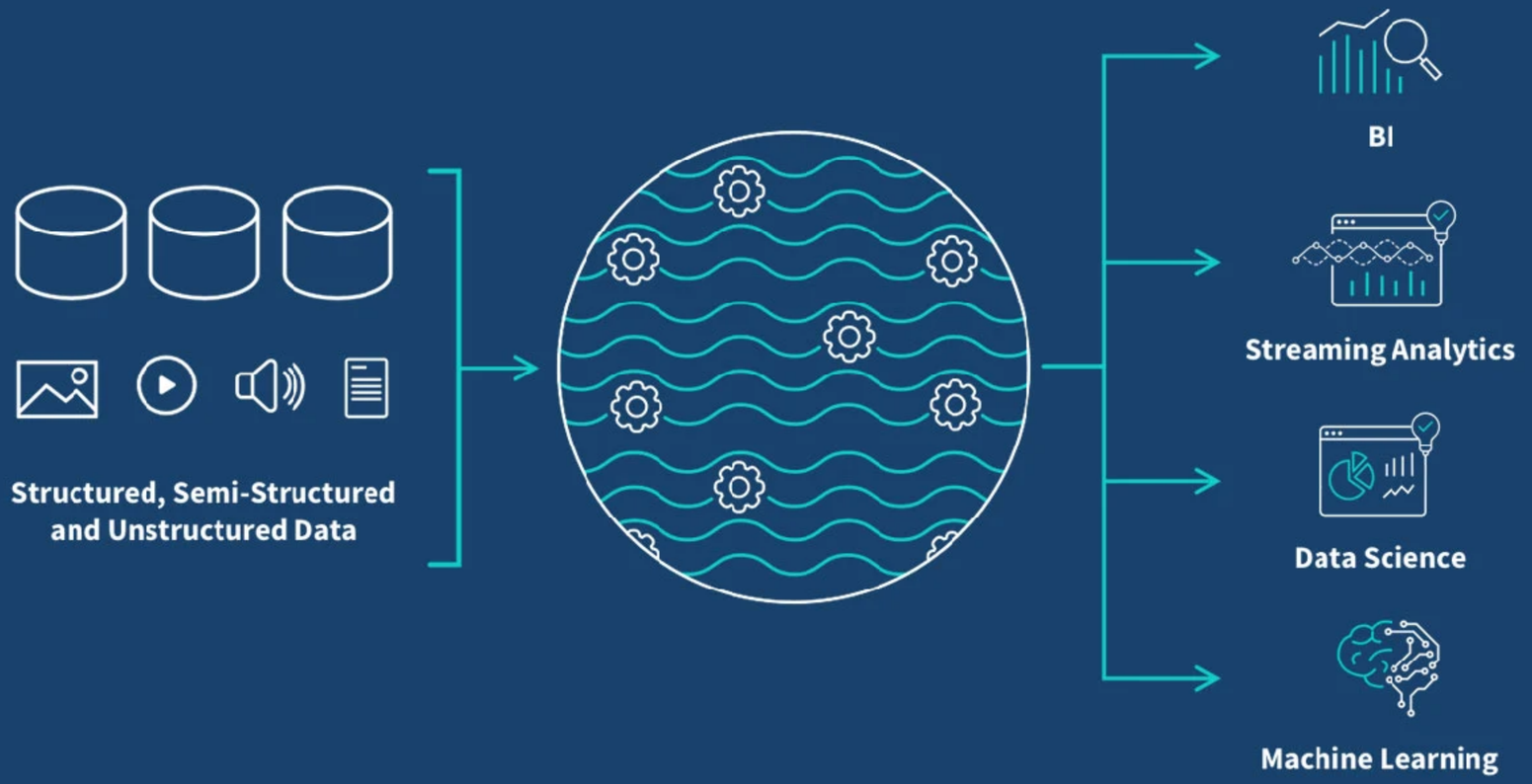
Data Lakes

The Problem

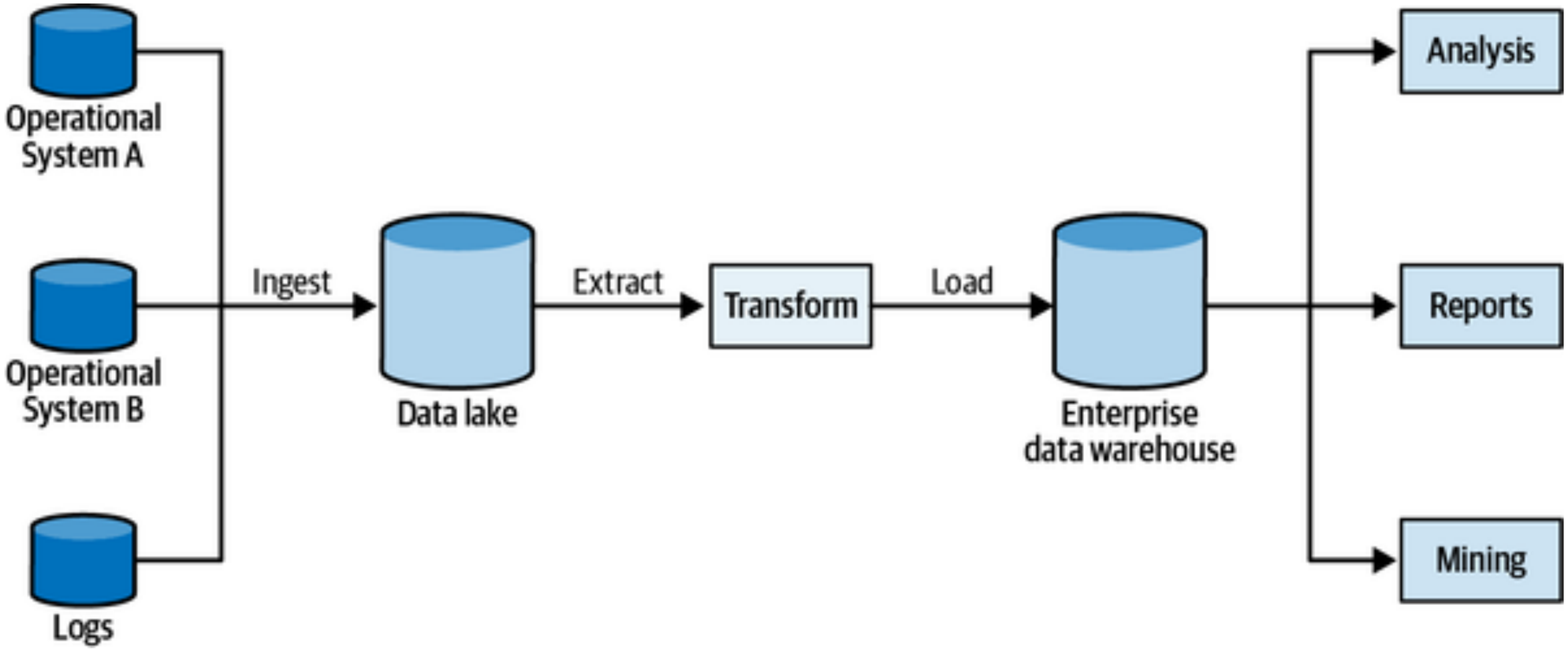
- **Data Silos:** Business units (e.g. Microservices) maintain separate databases, leading to fragmented insights.
- **Complex Queries:** Operational databases are optimized for transactions, not analytics, making complex queries slow and resource-intensive.
- **Data Inconsistency:** Different systems use varying formats and standards, complicating analysis.
- **Scaling Issues:** Traditional OLTP systems struggle with high-volume analytical workloads.
- **Decision Delays:** Lack of a unified view slows down decision-making processes.

The Solution

- **A Data lake is a unified repository for all data**
- A data lake is a centralized repository that **stores raw, unprocessed data in its native format.**
- **Store everything:** Handles structured, semi-structured, and unstructured data.
- **Schema-on-Read:** Data schema is applied when queried, not when stored.
- **Scalability:** Designed to scale with increasing data volumes.
- **Cost-Effective:** Often built on cheap, scalable storage like object stores (e.g., Amazon S3).


Structured, Semi-Structured
and Unstructured Data
BI
Streaming Analytics
Data Science
Machine Learning

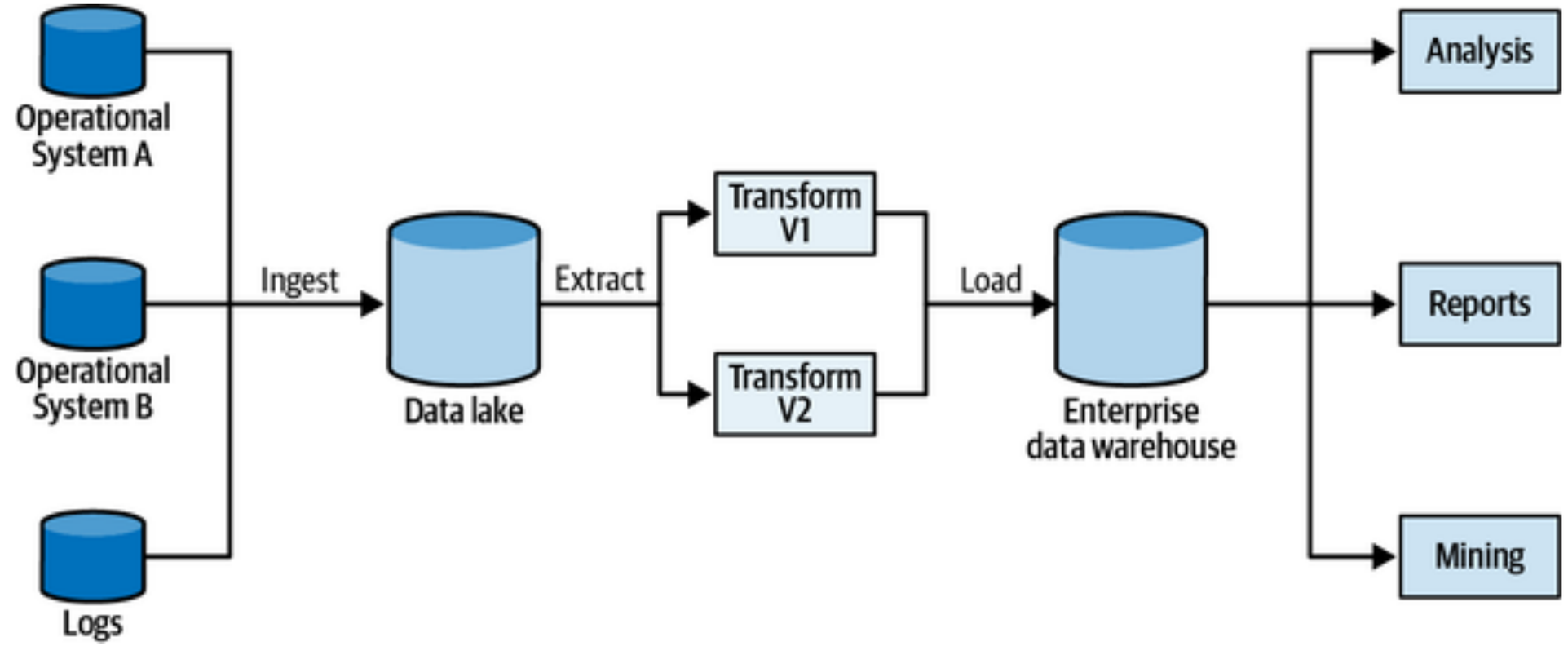
Data Lake



Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy - Vlad Khononov

Data Lake

It’s not uncommon for data engineers to implement and support multiple versions of the same ETL script to accommodate different versions of the operational model



Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy - Vlad Khononov

Small Details

- Supports large volumes of data in diverse formats: CSV, JSON, Parquet, Avro, video, images, logs, etc.
- Typically implemented using object storage (e.g., Amazon S3, Azure Data Lake Storage).
- Integrates with tools like Kafka, Flume, or AWS Glue for ingesting data.
- Supports AI/ML workflows, ad-hoc analysis, and business intelligence tools.
- Implements data governance and security policies (e.g., AWS Lake Formation).

Tradeoffs

- **Advantages:**
  - Cost-effective for storing raw, unstructured data.
  - Flexible schema-on-read approach.
  - Scalable for petabyte-scale storage and analytics.
  - Facilitates AI/ML and real-time processing.
- **Disadvantages:**
  - Risk of becoming a “Data Swamp”: Poorly managed data lakes can lead to unorganized, low-quality data.
  - Higher complexity for users to derive value (requires robust querying and data governance).
  - Latency in querying raw data compared to structured warehouses.
  - Security challenges with managing diverse data formats and access policies.

The Technologies

- **Storage**
  - Amazon S3 + AWS Lake Formation
  - Azure Data Lake Storage
  - Google Cloud Storage
- **Querying**
  - **Databricks:** Unified data analytics platform built on Apache Spark.
  - **Presto/Trino:** Distributed SQL query engine for querying data directly in lakes.
  - **Apache Hive:** Query and manage data stored in Hadoop-based lakes.
  - **AWS Athena:** Serverless SQL engine for querying data in Amazon S3.

The Technologies

- **Data Integration Tools:**
  - **Apache Kafka:** Streaming data pipelines feeding the data lake.
  - **Apache NiFi:** Data flow management for ingesting data into lakes.
  - **Fivetran / Stitch:** Cloud-native tools for ELT into lakes.
- **Data Governance and Management:**
  - **Apache Atlas:** Metadata management for data lakes.
  - **AWS Lake Formation:** Security, governance, and cataloging for AWS-based lakes.

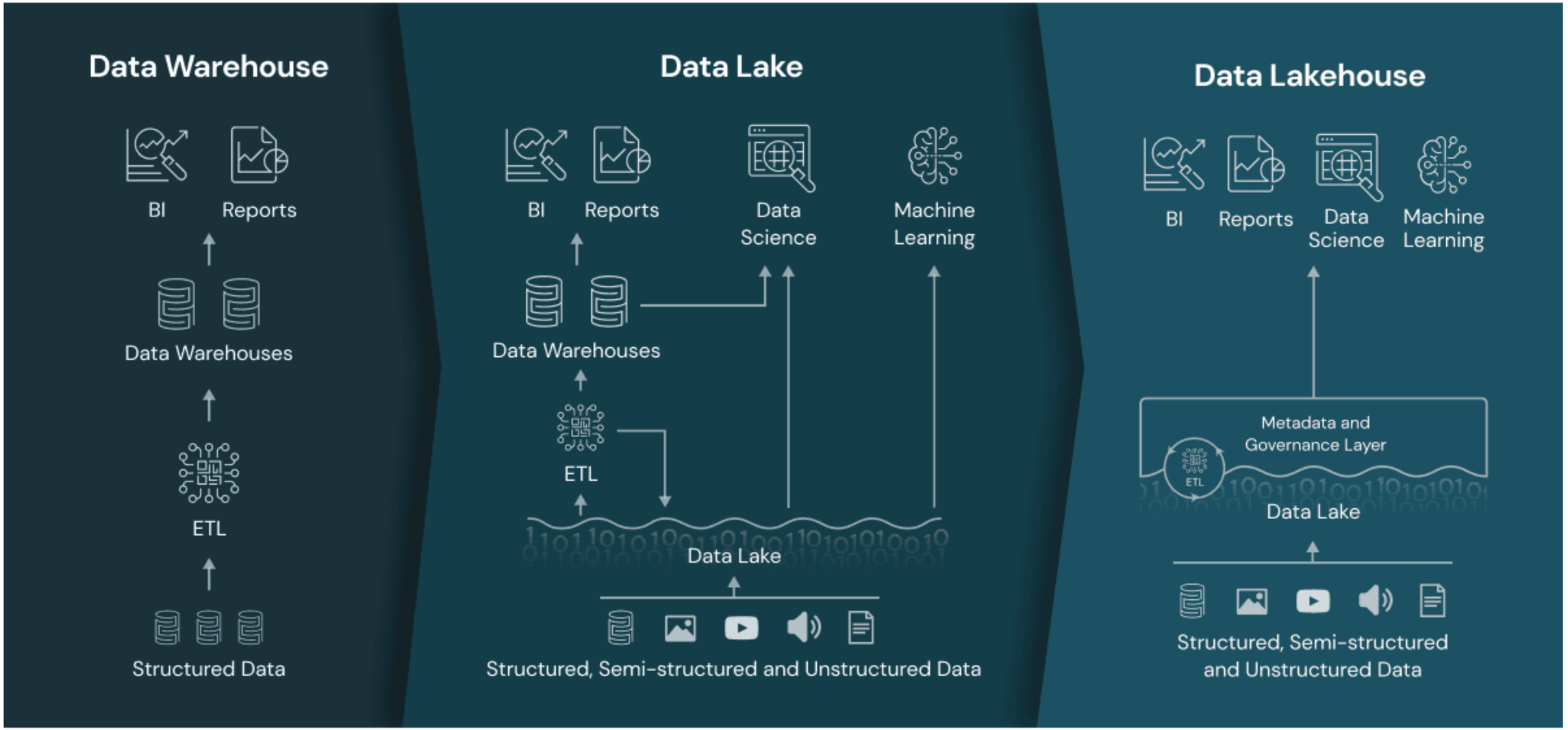
Data Lakehouse

The Problem

- **Data Warehouse issues:**
  - **Rigid Schemas:** Cannot handle diverse or unstructured data types (e.g., videos, logs).
  - **Cost:** High storage costs due to storing only processed data.
  - **Latency:** Time-consuming ETL processes delay data availability.
- **Data Lake issues:**
  - **Lack of Governance:** Risk of becoming a “data swamp” with unorganized, poor-quality data.
  - **Performance Issues:** Querying raw data directly is slow and inefficient.
  - **No Unified Analytics:** Separates raw data storage and structured analytics, requiring complex integrations.

The Solution

- **A Data Lakehouse bridges this gap**
- A data lakehouse combines the best of data lakes and data warehouses into a single system.
  - Handles structured, semi-structured, and unstructured data.
  - Provides high-performance analytics and supports AI/ML workloads.
  - Enforces data governance while retaining the scalability and flexibility of data lakes.



https://www.databricks.com/glossary/data-lakehouse

Small Details

- **Unified Architecture: Single system for storage and analytics.**
  - **Open Storage Format:** Supports Parquet, ORC, or Delta for easy access and compatibility.
  - **Real-Time Data Processing:** Enables faster insights with streaming capabilities.
  - **Integrated Governance:** Ensures quality, lineage, and compliance across all data.

The Tradeoffs

- **Advantages:**
  - **Flexibility:** Handles all data types in a single platform.
  - **Cost-Efficiency:** Combines low-cost storage of data lakes with the performance of data warehouses.
  - **Unified Workflows:** Simplifies data pipelines by eliminating the need for separate systems.
  - **Real-Time Insights:** Supports streaming data and near-real-time analytics.
- **Disadvantages:**
  - **Complexity:** Requires expertise to design and implement a lakehouse.
  - **Integration Challenges:** Existing tools and workflows may need significant adaptation.
  - **Evolving Technology:** Still a relatively new concept, with varying levels of maturity across implementations.
  - **Performance Overhead:** Additional layers (e.g., metadata, indexing) may introduce slight overhead compared to pure warehouses.

The Technologies

- **Databricks Lakehouse:** Built on Delta Lake for unified analytics and AI.
- **Snowflake:** Combines data warehouse and data lake features with support for unstructured data.
- **Google BigLake:** A unified storage engine combining BigQuery and Google Cloud Storage
- **Presto/Trino:** Distributed SQL query engine for querying raw data.
- **Apache Spark:** Processes both raw and structured data for analytics and AI/ML.

Data Streaming

The Problem

- **Data Delays:** Batch processing introduces latency, leading to delayed insights.
- **Real-Time Demands:** Modern applications require immediate data updates (e.g., stock prices, IoT sensors).
- **Scalability Issues:** Traditional systems struggle with high-velocity and high-volume data ingestion.
- **Event Complexity:** Processing events sequentially or maintaining event order can be challenging.
- **Disconnected Systems:** Difficulty in syncing data across distributed systems in real-time.

The Solution

- Data streaming is the real-time, continuous flow of data from various sources to enable immediate processing, transformation, and analysis.
- Focuses on handling data in motion as opposed to static, batch data processing.
- **Low latency:** Near real-time data ingestion and processing.
- **Continuous:** Uninterrupted flow of data.
- **Scalable:** Handles large volumes of data generated at high velocity.
- Examples:
  - Financial transactions, stock market feeds.
  - IoT sensor data, telemetry from devices.
  - Application logs, user activity streams (e.g., clicks, searches).

The Ingredients

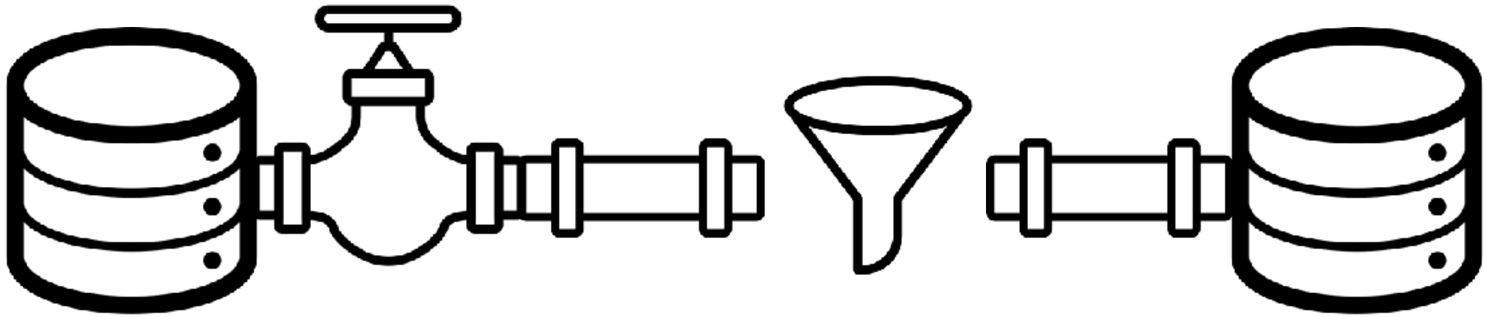
- **Event Production and Consumption:** Events generated by producers (e.g., IoT devices, logs, applications) are consumed by downstream services.
- **Message Brokers and Topics:** Streams organized into topics to allow pub/sub or point-to-point models.
- **Windowing and Aggregation:** Real-time analytics via sliding or tumbling windows.
- **Fault Tolerance:** Guarantees data durability and replay in case of failure.
- **Data Partitioning:** Ensures scalability by distributing streams across partitions.
- **Integrations:** Connects to data lakes, warehouses, and external services for downstream processing.


NY
100.00

Filtering

NY

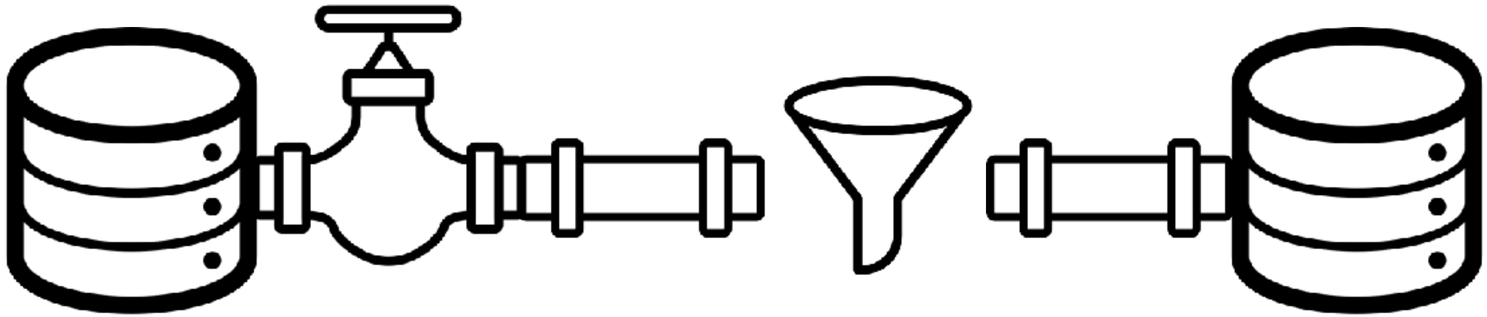
100.00



```
value > 10000
```

OH

20000

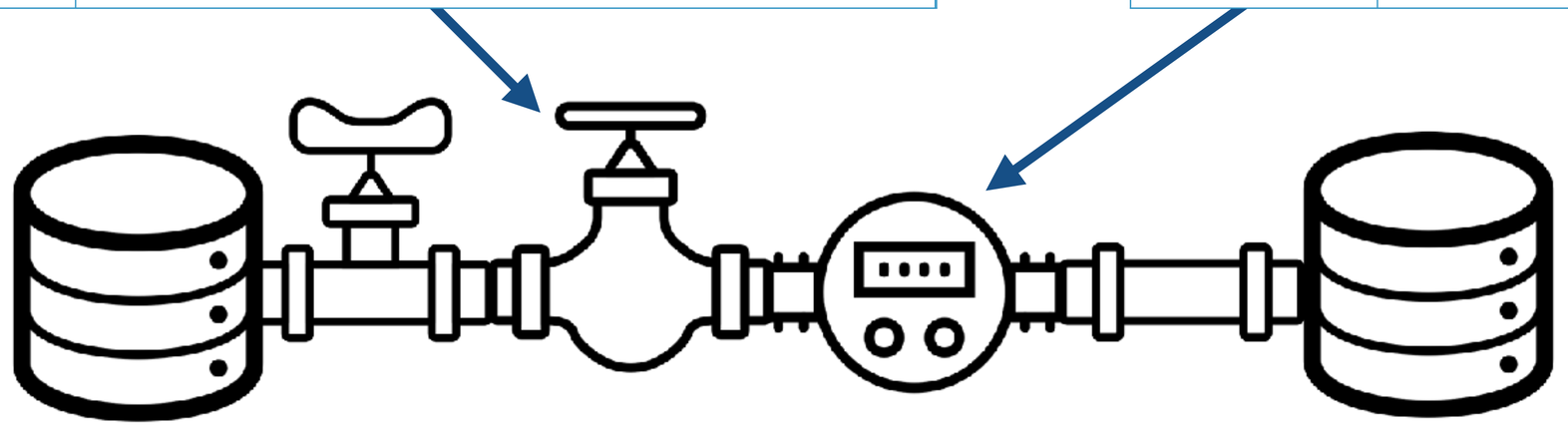


value > 10000

Aggregating

Key	Value
OH	100.0

Key	Value
OH	100.0



OH

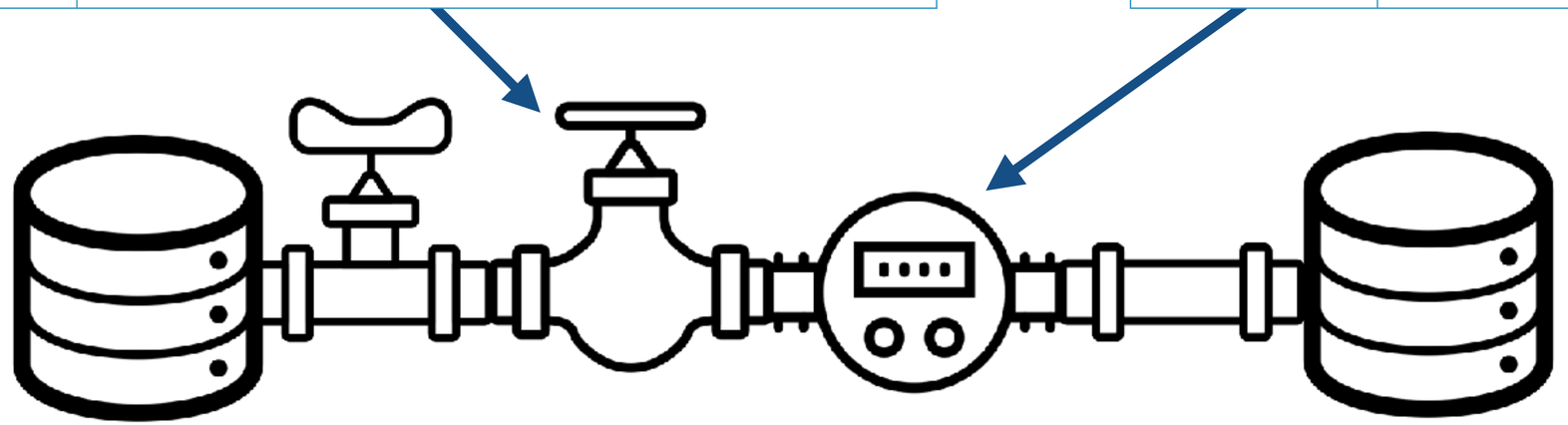
100.00

OH

100.00

Key	Value
OH	100.0
NY	20.0

Key	Value
OH	100.0
NY	20.0



NY

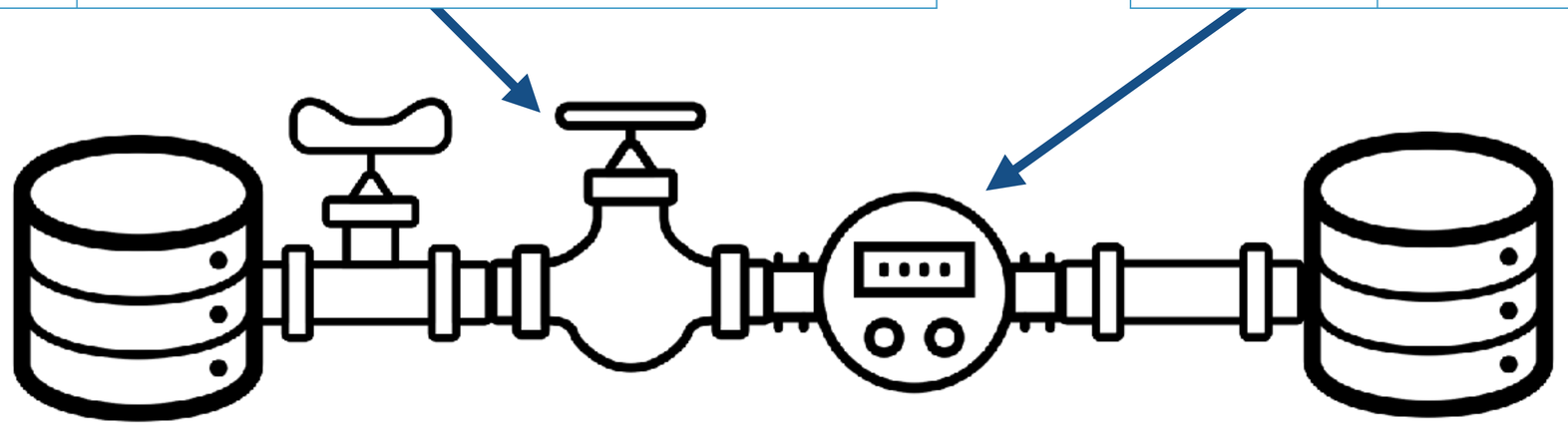
20

NY

20

Key	Value
OH	100.0
NY	20.0, 60.0

Key	Value
OH	100.0
NY	80.0

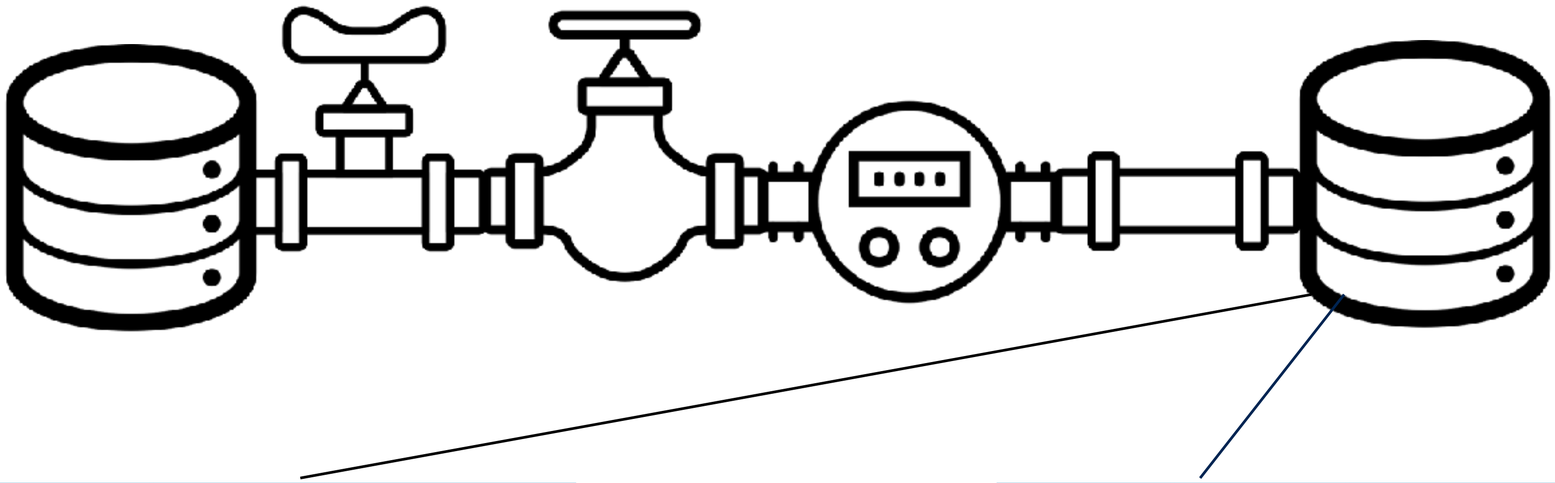


NY

60

NY

80.0

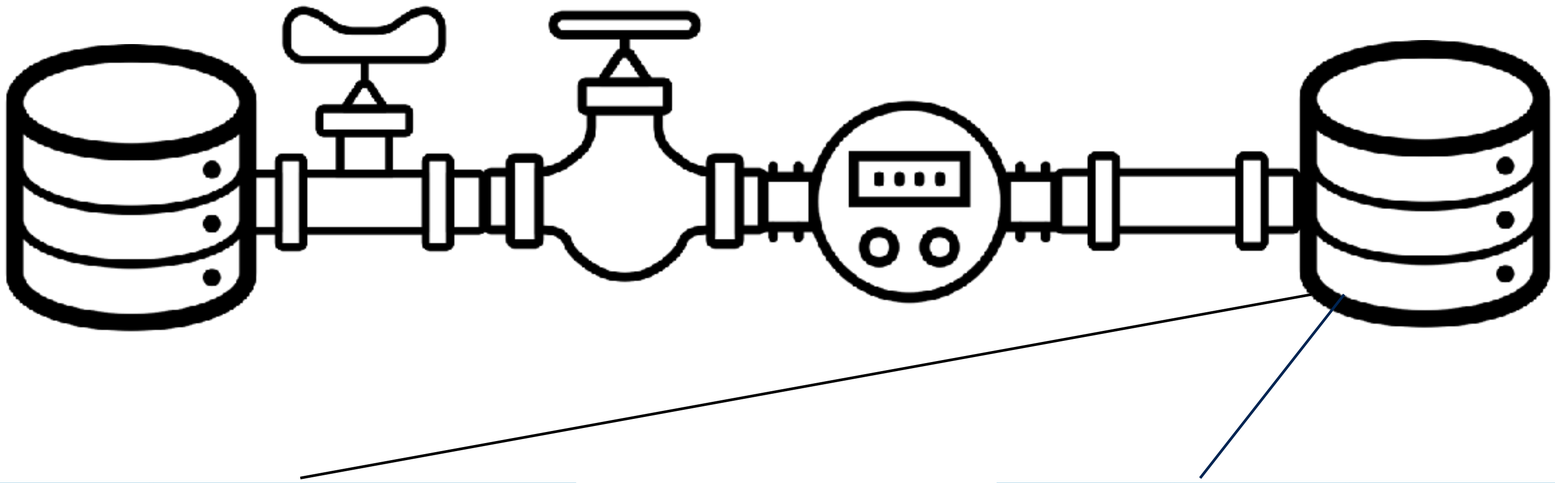


Offset	Key	Value
0	**OH**	100.0

Partition 0

Offset	Key	Value
0	**NY**	20.0
1	**NY**	80.0

Partition 1



Offset	Key	Value
0	**OH**	100.0

Partition 0

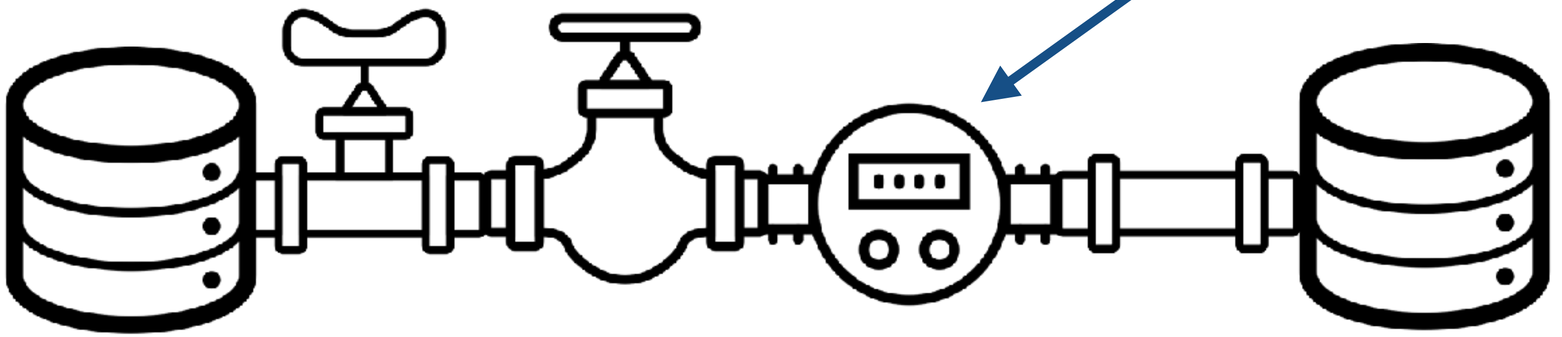
Offset	Key	Value
1	**NY**	80.0

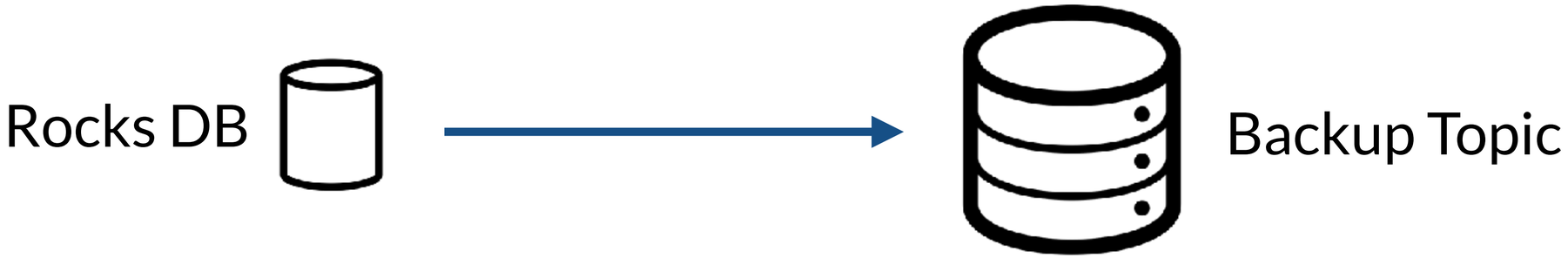
Partition 1

After Compaction

Rocks DB

Key	Value
OH	100.0
NY	80.0




Rocks DB
Backup Topic

Windowing

Tumbling Window

Hopping Window

Session Window

Session Window

the last element has a occurred; now we wait

Session Window

waited 5 minutes = make it a window

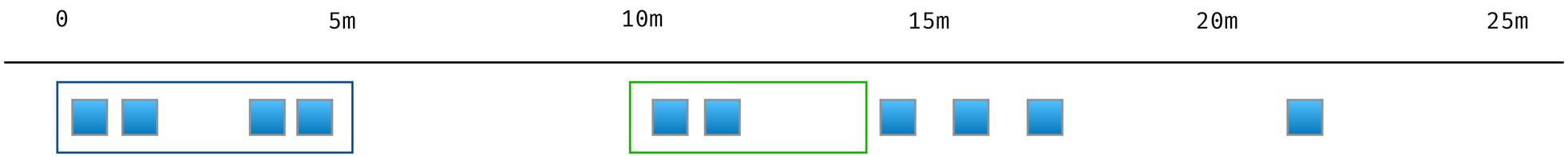
Session Window

we have a new element; we start a new window

Session Window

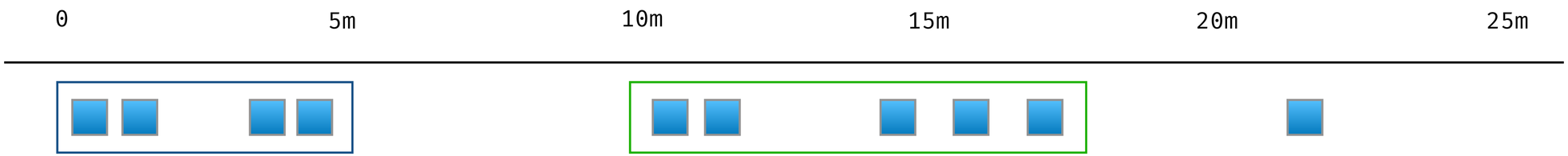
we wait for 5 minutes

Session Window



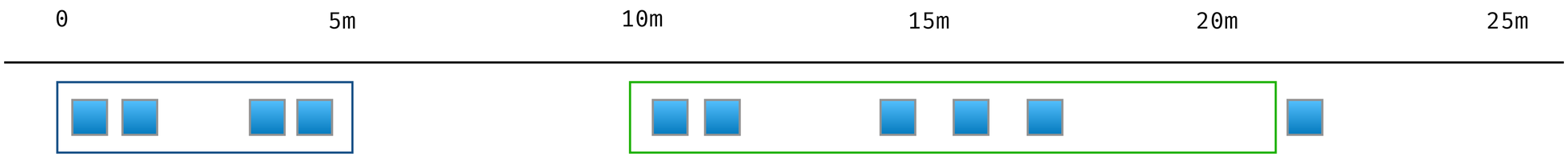
2 minutes elapsed; no windowing yet

Session Window



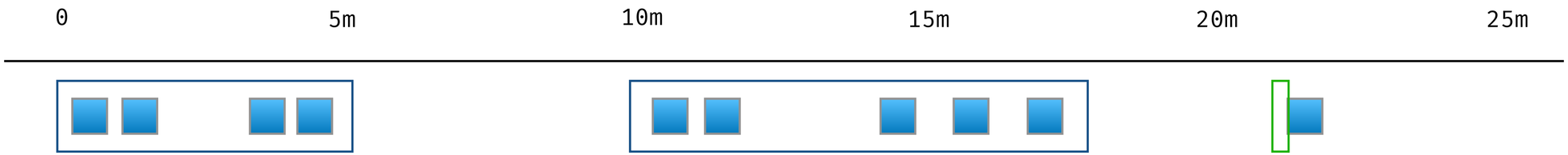
we wait five minutes

Session Window



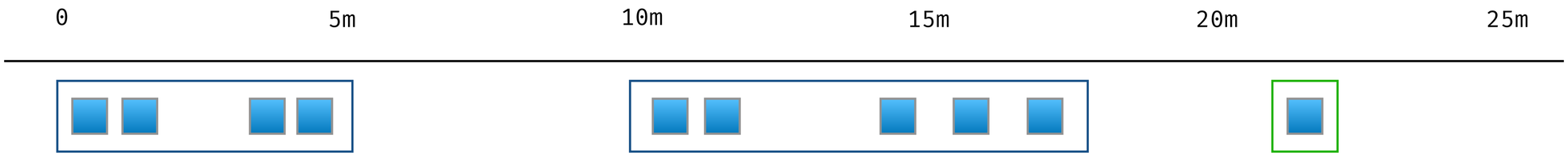
no new elements; we make it a window

Session Window



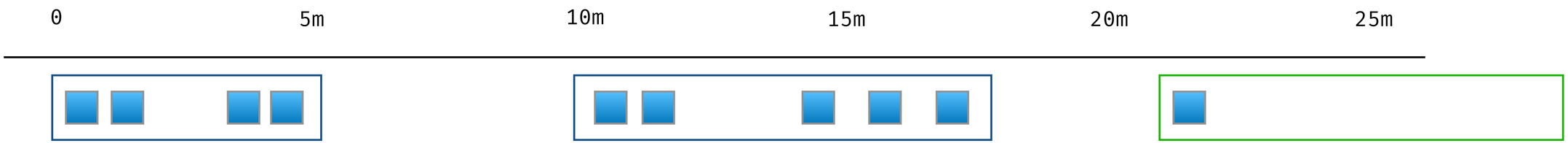
we see something new

Session Window



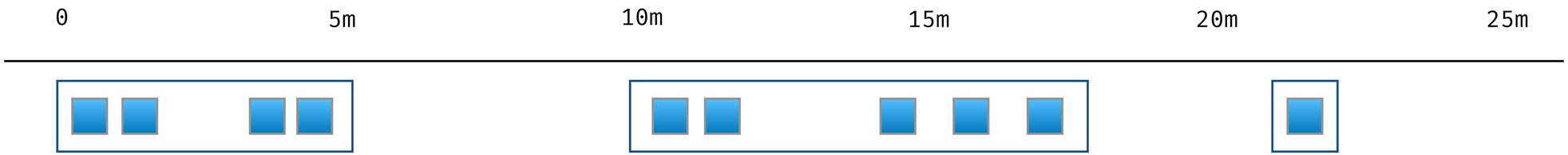
last element; we wait five minutes

Session Window



nothing else; we wait 5 minutes

Session Window



that's a new window

The Technology

- **Open-Source Platforms:** Apache Kafka, Apache Pulsar, Apache Flink, RabbitMQ (with stream plugin)
- **Cloud-Native Streaming Services:** AWS Kinesis, Google Pub/Sub, Azure Event Hubs, Confluent Cloud (Managed Kafka)
- **Analytics and Integration Tools:** Spark Streaming, Apache Storm, Dataflow (Google Cloud)
- **IoT and Specialized Use Cases:** MQTT Brokers (e.g., Mosquitto), Kafka Streams, Amazon IoT Core

Data Glaciers

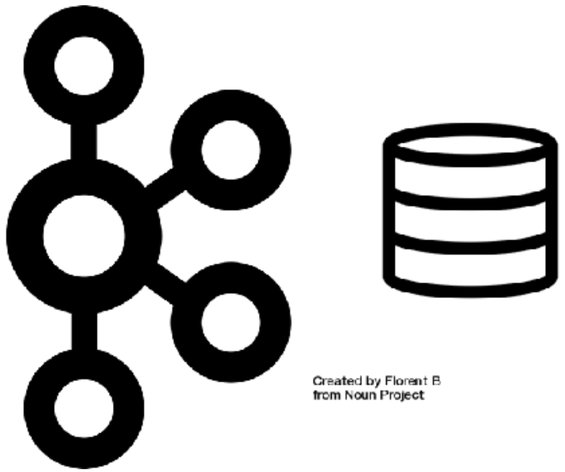
The Problem

- Keeping infrequently accessed data in high-performance storage results in unnecessary expenses.
- **Example:** Using SSDs for decades-old compliance logs.
- Inactive data stored alongside operational data can slow down query performance in primary systems.

The Solution

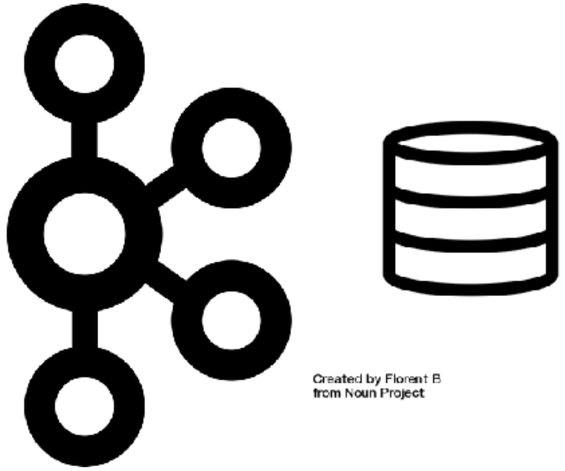
- A “data glacier” is an archived or rarely accessed data stored for long-term retention and compliance purposes.
- Shift rarely accessed data to low-cost, high-capacity storage solutions optimized for archival purposes.
- **Separation of Hot and Cold Data:** Distinguish frequently accessed data (hot) from rarely accessed data (cold).
- **Tiered Storage:** Move cold data to a dedicated storage tier designed for long-term retention.
- **Compression and Deduplication:** Optimize space by reducing storage requirements for archival data.
- **Lifecycle Management:** Automate movement of data to glacier storage based on access patterns and age.
- **Governance and Compliance:** Ensure archived data meets regulatory standards for retention and auditability.

Tiered-Storage



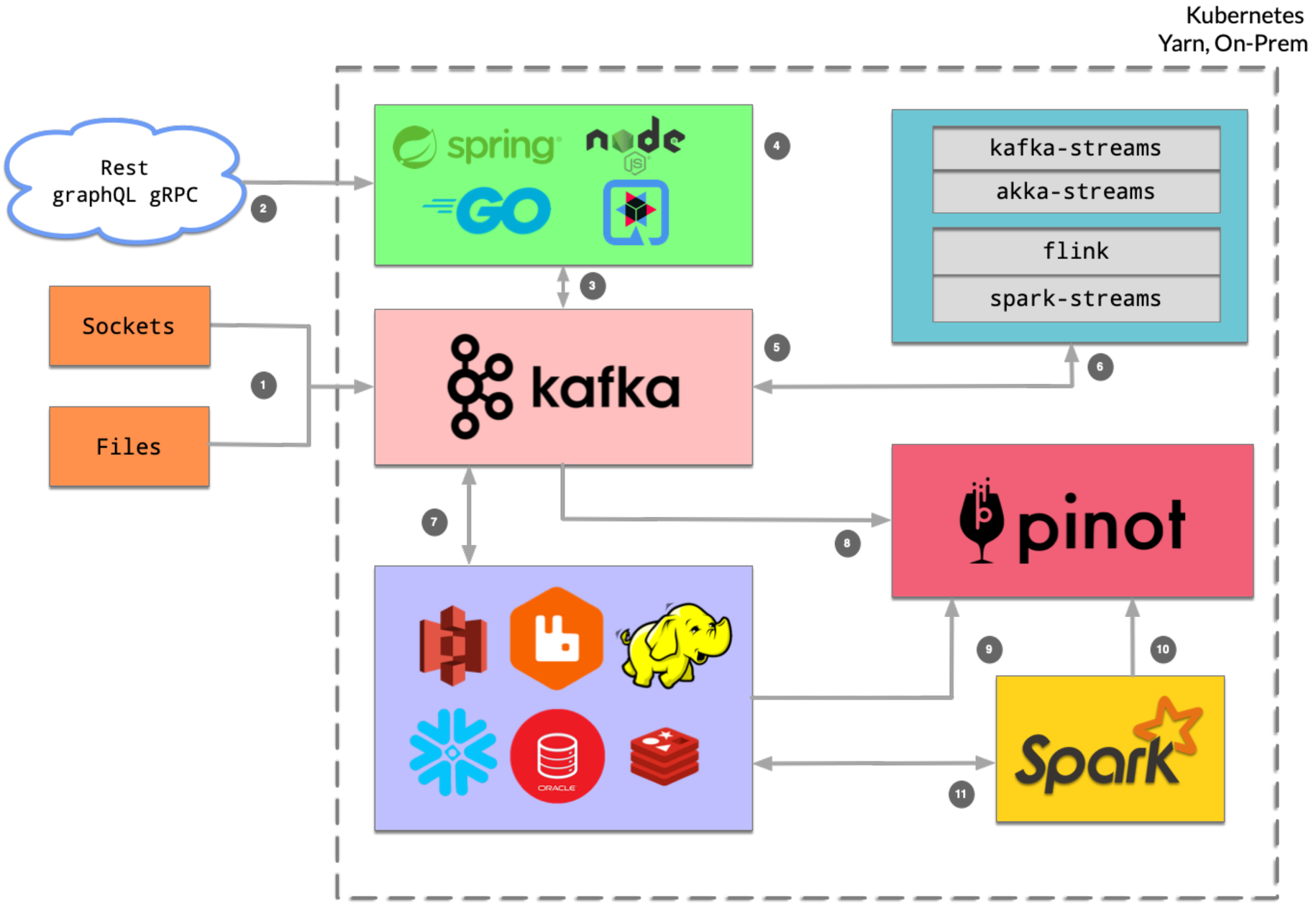
Created by Florent B
from Noun Project

Long Term back up in the Cloud



Have active data in the cluster for Kafka with whatever retention time you would like

Big Data Architecture


Kubernetes
Yarn, On-Prem
Rest
graphQL gRPC
Sockets
Files
spring
node
GO
kafka
kafka-streams
akka-streams
flink
spark-streams
pinot
ORACLE
Spark
1
2
3
4
5
6
7
8
9
10
11

Data Mesh

The Problem

- Data can be siloed Database per Service. To circumvent, we created a data lake or shared database
- Problem is that the data lake, ended up as a data monolith
- By using a shared storage, we lost governance, and ownership
- We wanted to apply bounded context patterns, and did so with services, but lost the point with data.

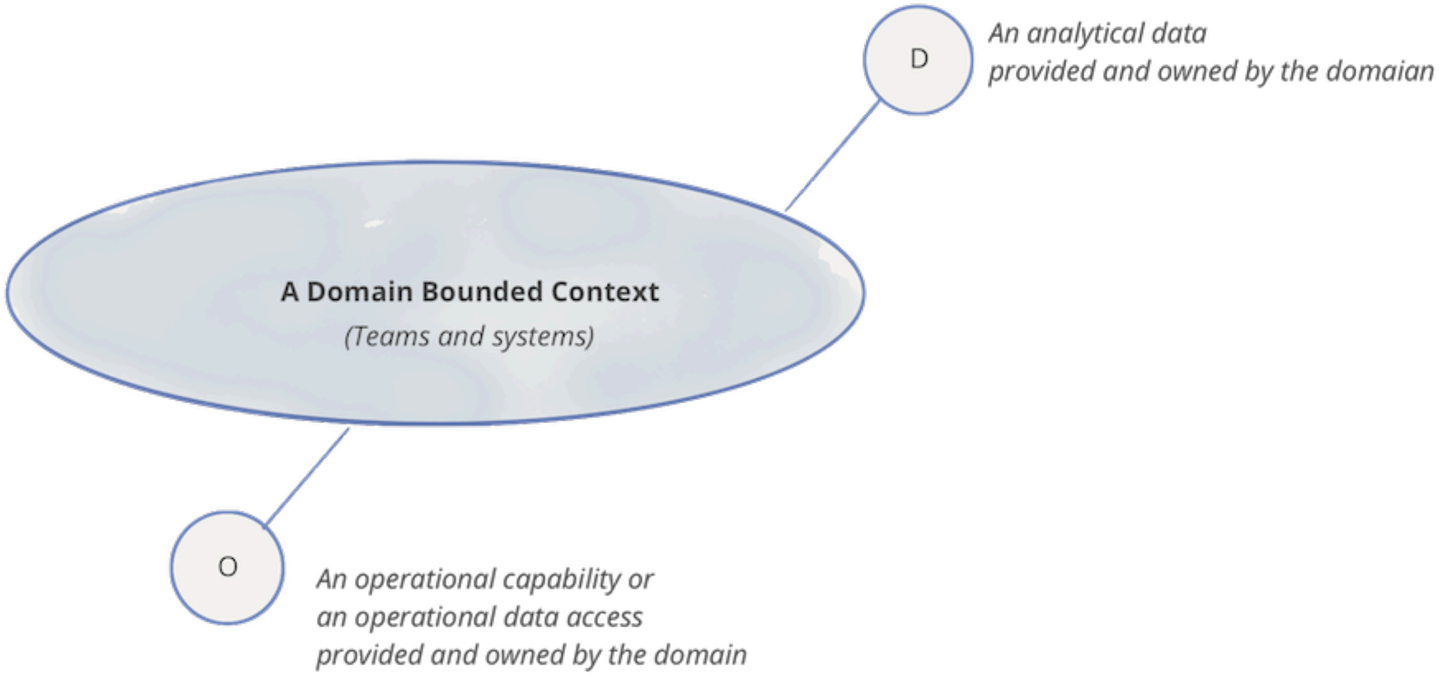
The Solution

- There are two patterns: Database per Service, and Shared Database
- Data Mesh is an attempt at bringing not only the database but owning all data
- Data Mesh is about providing data ownership per context, and follows these principles:
- Domain-oriented decentralized data ownership and architecture
- Data as a product
- Self-serve data infrastructure as a platform
- Federated computational governance
- The accountability of data quality shifts upstream as close to the source of the data as possible.

The corps was composed of all arms of the service, was **self-sustaining**, and could fight on its own until other corps could join in the battle. The corps itself was a headquarters to which units could be attached. **It might have attached two to four divisions of infantry with their organic artillery, it had its own cavalry division and corps artillery, plus support units**. With this organization a corps was expected to be able to hold its ground against, or fight off an enemy army for a least a day, when neighboring corps could come to its aid. "Well handled, it can fight or alternatively avoid action, and maneuver according to circumstances without any harm coming to it, because an opponent cannot force it to accept an engagement but if it chooses to do so it can fight alone for a long time."

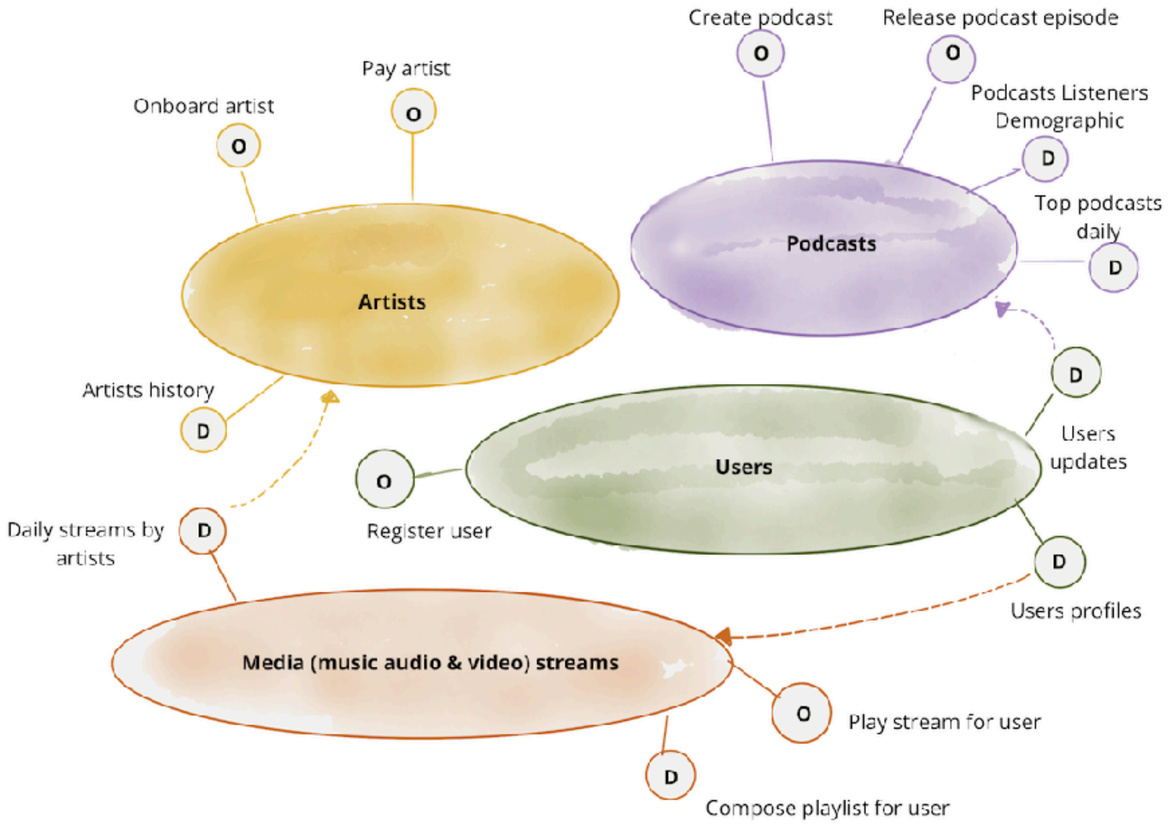
https://www.napoleon-series.org/military-info/organization/c_armycorps.html

The Diagram



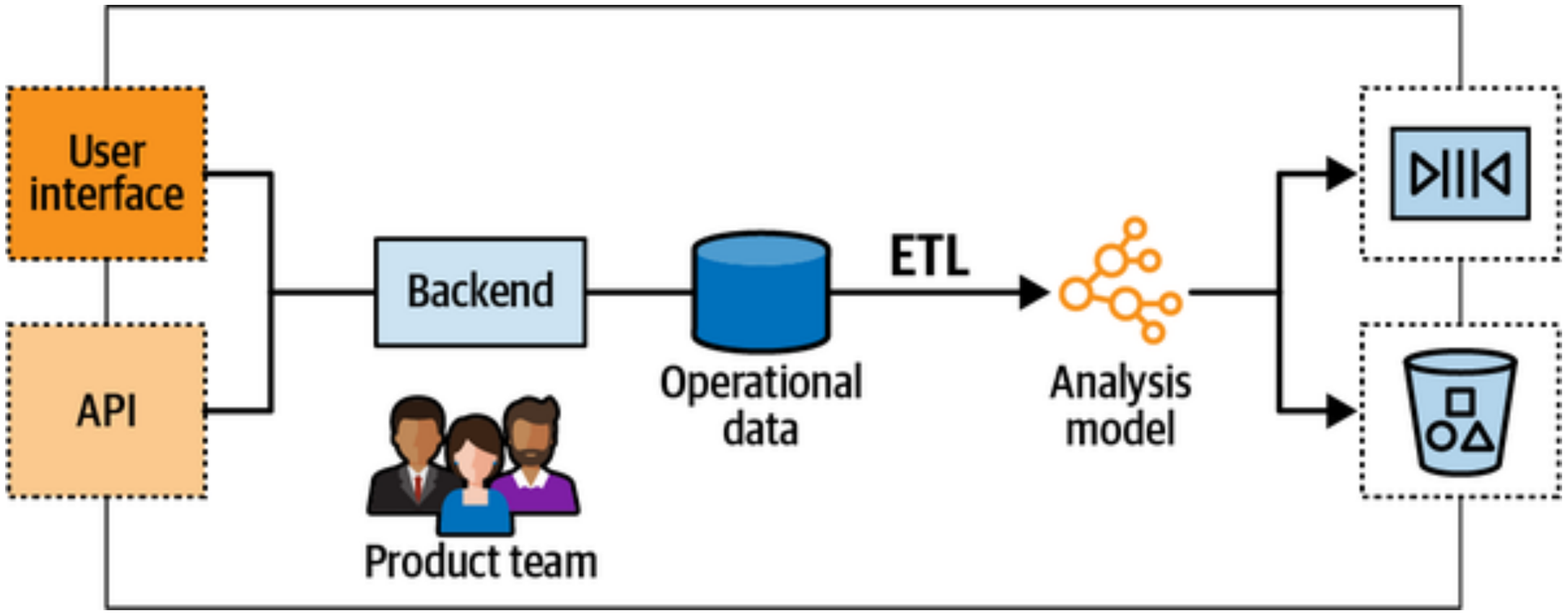
https://martinfowler.com/articles/data-mesh-principles.html

The Diagram



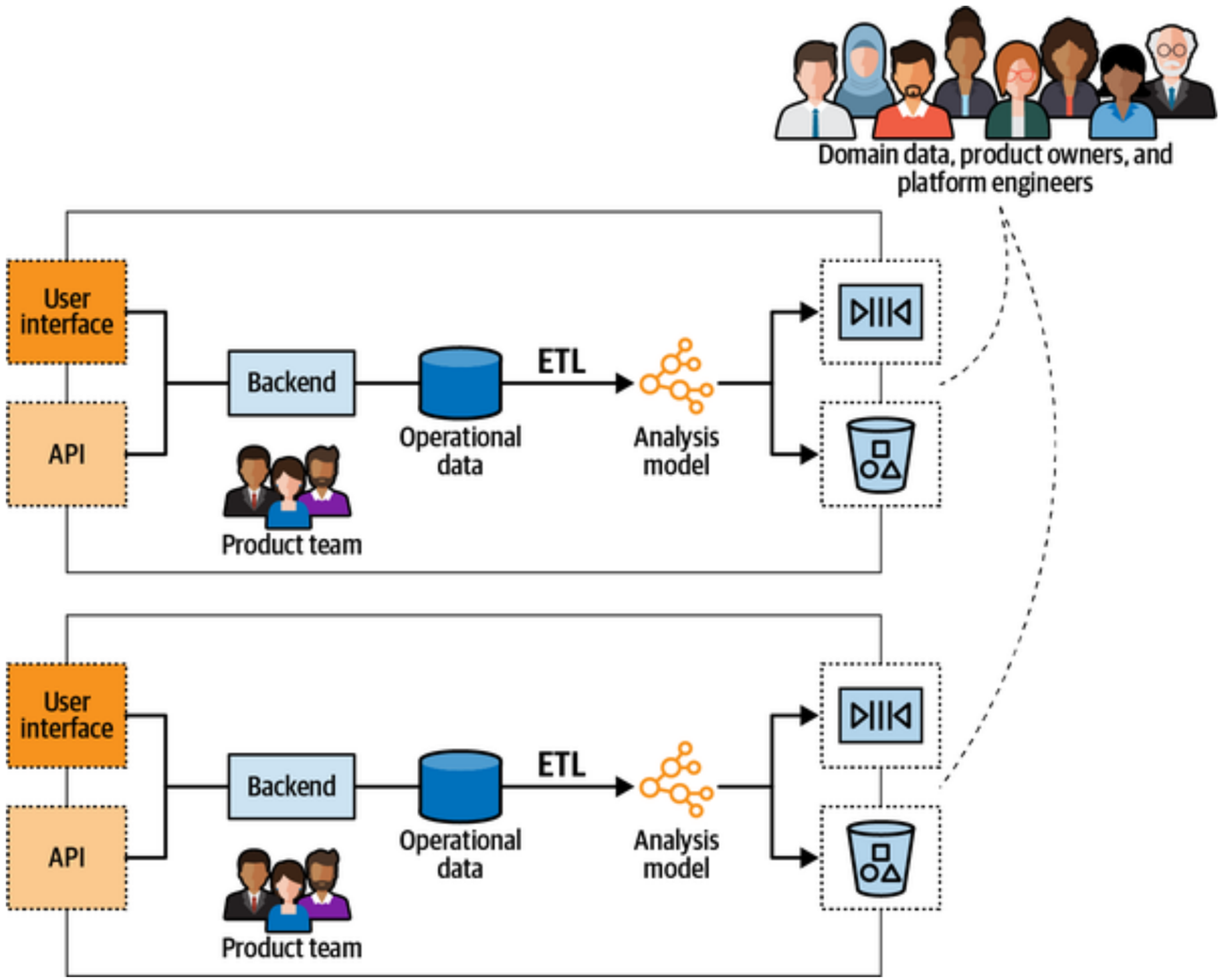
https://martinfowler.com/articles/data-mesh-principles.html

The Diagram



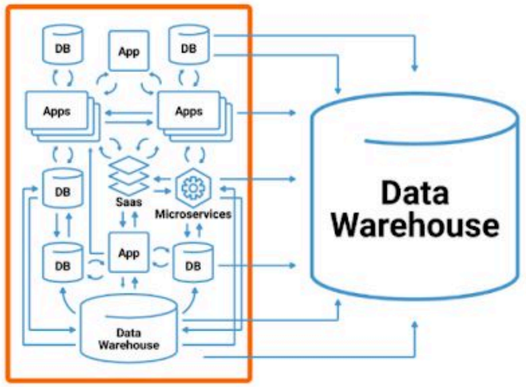
Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy - Vlad Khononov

The Diagram


Domain data, product owners, and platform engineers
User interface
API
Backend
Operational data
ETL
Analysis model
Product team
User interface
API
Backend
Operational data
ETL
Analysis model
Product team

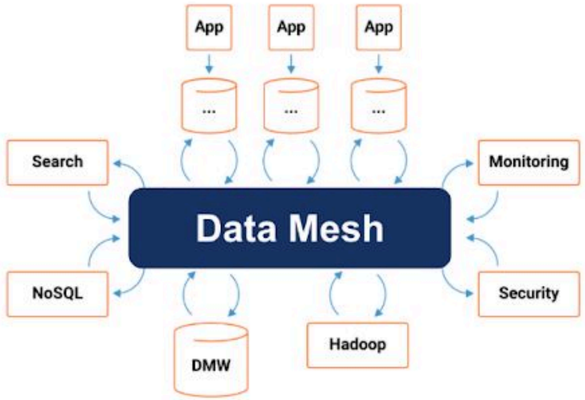
The Diagram

Anti-pattern: responsibility for data becomes the domain of the DWH team



Centralized Data Ownership

Pattern: Ownership of a data asset given to the "local" team that is most familiar with it



Decentralized Data Ownership

https://www.confluent.io/blog/how-to-build-a-data-mesh-using-event-streams/

The Diagram

Domain-oriented decentralized data ownership and architecture	So that the ecosystem creating and consuming data can scale out as the number of sources of data, number of use cases, and diversity of access models to the data increases; simply increase the autonomous nodes on the mesh.
Data as a product	So that data users can easily discover, understand and securely use high quality data with a delightful experience; data that is distributed across many domains.
Self-serve data infrastructure as a platform	So that the domain teams can create and consume data products autonomously using the platform abstractions, hiding the complexity of building, executing and maintaining secure and interoperable data products.
Federated computational governance	So that data users can get value from aggregation and correlation of independent data products - the mesh is behaving as an ecosystem following global interoperability standards; standards that are baked computationally into the platform.

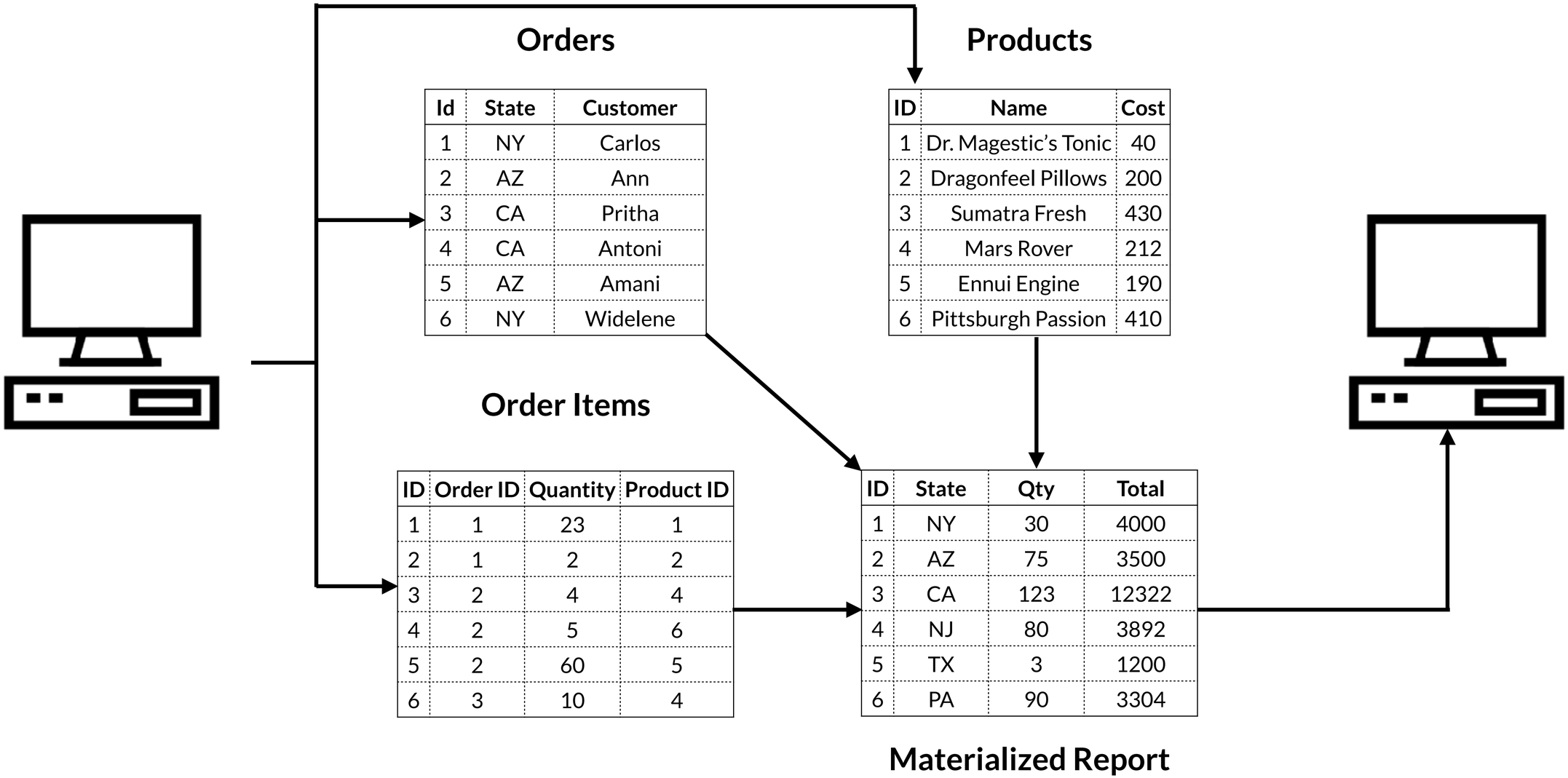
https://martinfowler.com/articles/data-mesh-principles.html

Materialized Views

The Problem

- Currently much of the data revolves on how it is stored, not how it is read
- Data when read needs to transformed and prepared
- Each entity typically has too much information for it to be queryable and usable
- For example, one data entry in a document style database can have other aggregates that are not absolutely necessary
- Data may also need to be joined, cleansed, or engineered for a particular purpose, like machine learning

The Diagram

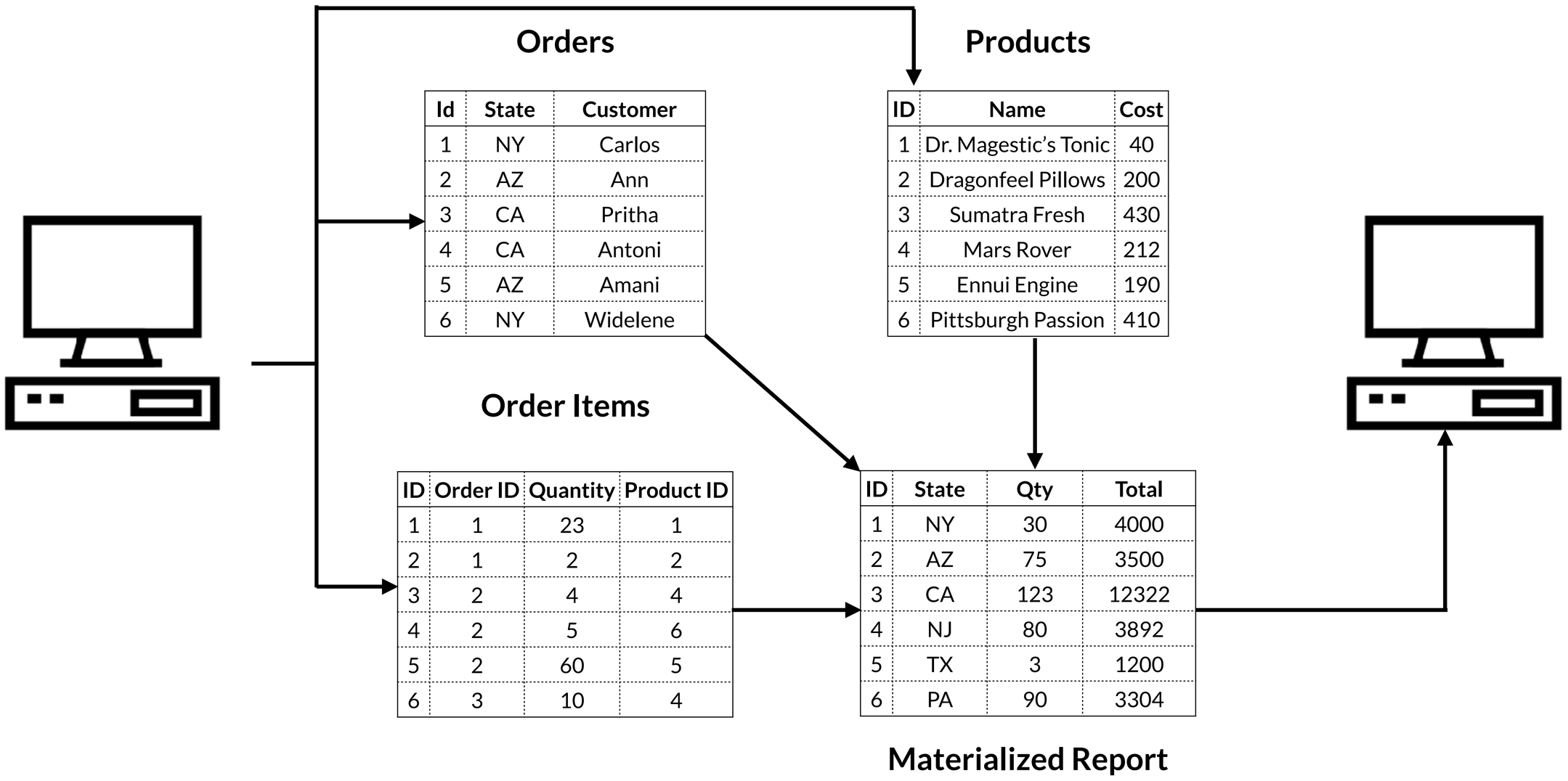


Id	State	Customer
1	NY	Carlos
2	AZ	Ann
3	CA	Pritha
4	CA	Antoni
5	AZ	Amani
6	NY	Widelene

ID	Name	Cost
1	Dr. Magestic's Tonic	40
2	Dragonfeel Pillows	200
3	Sumatra Fresh	430
4	Mars Rover	212
5	Ennui Engine	190
6	Pittsburgh Passion	410

ID	Order ID	Quantity	Product ID
1	1	23	1
2	1	2	2
3	2	4	4
4	2	5	6
5	2	60	5
6	3	10	4

ID	State	Qty	Total
1	NY	30	4000
2	AZ	75	3500
3	CA	123	12322
4	NJ	80	3892
5	TX	3	1200
6	PA	90	3304



Id	State	Customer
1	NY	Carlos
2	AZ	Ann
3	CA	Pritha
4	CA	Antoni
5	AZ	Amani
6	NY	Widelene

ID	Name	Cost
1	Dr. Magestic's Tonic	40
2	Dragonfeel Pillows	200
3	Sumatra Fresh	430
4	Mars Rover	212
5	Ennui Engine	190
6	Pittsburgh Passion	410

ID	Order ID	Quantity	Product ID
1	1	23	1
2	1	2	2
3	2	4	4
4	2	5	6
5	2	60	5
6	3	10	4

ID	State	Qty	Total
1	NY	30	4000
2	AZ	75	3500
3	CA	123	12322
4	NJ	80	3892
5	TX	3	1200
6	PA	90	3304

The Solution

- Create a different perspective of the data that you need
- This can be implemented by the datastore itself
  - Oracle
  - PostgreSQL
- Can be performed by Stream Processing Frameworks
  - Kafka
  - Spark
  - Flink

The Tradeoffs

- If your source is already simple and easy to query, there is no need
- If immediate consistency is required, then query direct

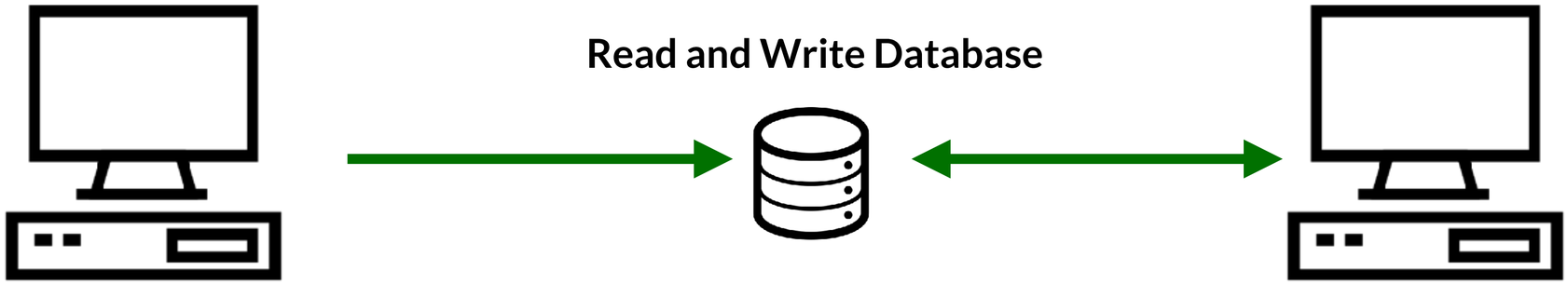
CQRS

CQRS

- **Command and Query Responsibility Segregation**
- Separating reads, add-updates, for a databases
- Benefits include better performance, scalability, and security
- Better evolution over time
- Prevents Merge Conflicts:
  - Database locking ensures that the updates don't change the same data concurrently, but it doesn't ensure that multiple independent changes result in a consistent data model.

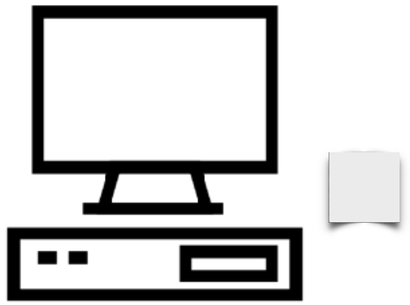
The Diagram

What if we need to query...
`SELECT state, count(*) from orders group by state`, over and over?



State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani

The Diagram



Write Database

State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani

Read Database

State	Total
NY	4000
AZ	2433
CA	1340

The Diagram

Write Database

State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani
NY	900	Widelene

Read Database

State	Total
NY	4000
AZ	2433
CA	1340

The Diagram

Write Database

State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani
NY	900	Widelene

Read Database

State	Total
NY	4900
AZ	2433
CA	1340

Demo: CQRS

- Next we will use Kafka to implement the following behaviors
  - Event Sourcing
  - Materialized Views
  - Outbox Pattern
  - CQRS (Command Query Responsibility Segregation)

Business Intelligence

Business Intelligence

- Not an architectural pattern, domain or methodology focused on collecting, analyzing, and presenting business data to support decision-making.
- However, certain **architectural patterns** are commonly used to implement BI systems effectively.
- BI Systems are designed to aggregate, process, and analyze data from multiple sources, including databases and messaging systems, to provide comprehensive insights.
  - **Tableau:** Can connect to relational databases, NoSQL databases, cloud warehouses, and even live streams via integrations.
  - **Apache Superset:** Open-source BI tool capable of connecting to databases and querying streaming systems.

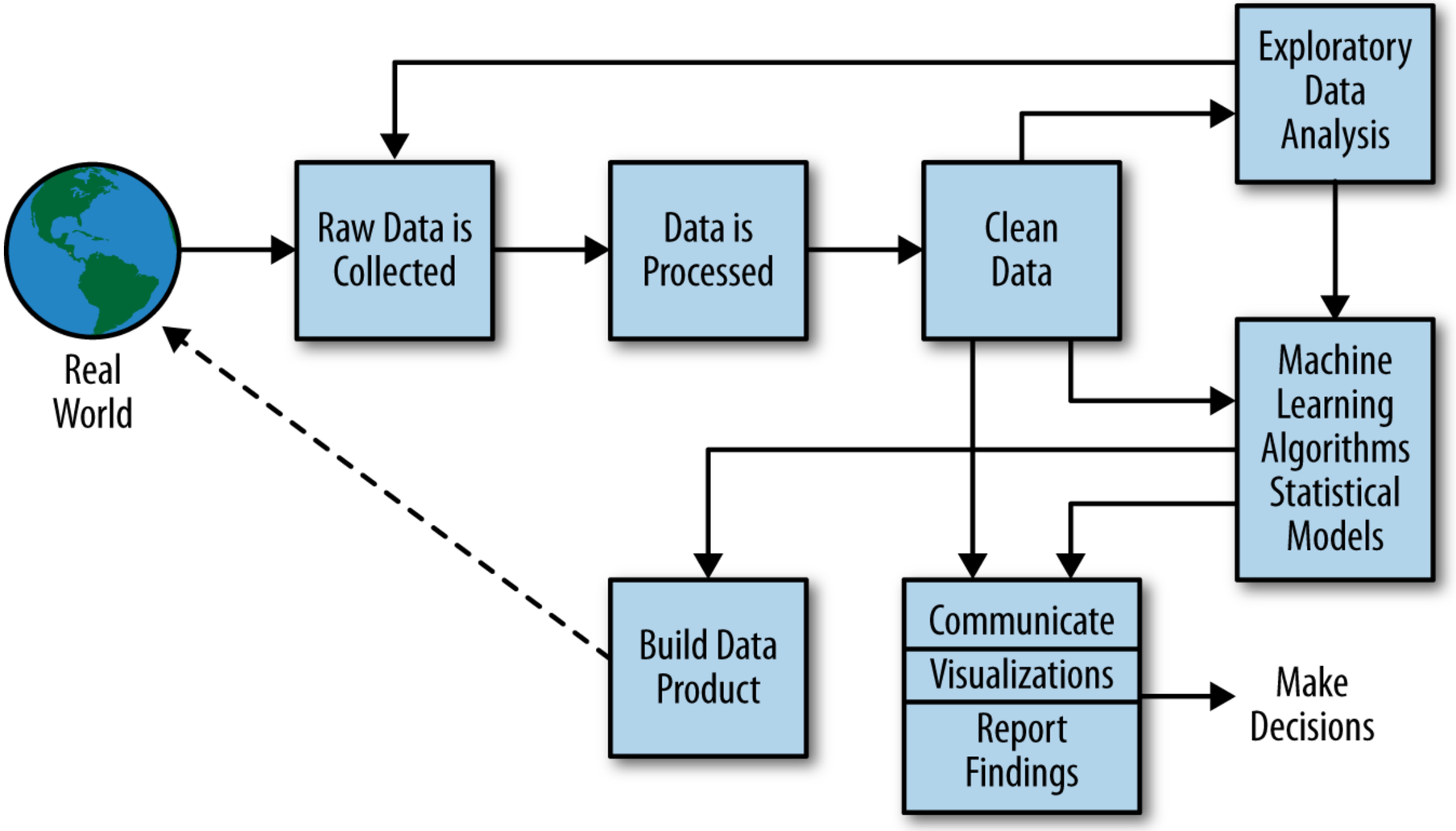
Architectural Patterns and BI

- **Data Warehousing Pattern**: Centralized storage for structured data, optimized for querying and reporting (OLAP).
- **Extract, Transform, Load (ETL)** A pipeline for extracting data from various sources, transforming it to match schema requirements, and loading it into a data warehouse or data lake.
- **Star Schema or Snowflake Schema Pattern:** Data modeling patterns for structuring data in a warehouse.
- **CQRS Pattern:** Separates write and read operations for optimized analytics.
- **Data Lake Pattern:** Stores unstructured or semi-structured data for future analysis and BI Querying

Technologies

- **Tableau -** Interactive data visualizations and dashboards.
- **Microsoft Power BI -** Seamless integration with the Microsoft ecosystem.
- **Apache Superset** - Open Source BI Tool

Making Data Available for ML/Ai



Doing Data Science Cathy O'Neil, Rachel Schutt Published by O'Reilly Media, Inc.

Making Data Available for AI

- Making data available for Machine Learning (ML) involves preparing, managing, and provisioning data in a way that meets the needs of ML workflows.
- While this is not an architectural pattern itself, it relies on several architectural patterns and best practices that ensure data is accessible, clean, and suitable for ML models.

Architectural Patterns and ML/AI

- **Data Lake Pattern:** A centralized repository that stores raw data in its native format. Handles both structured and unstructured data, providing flexibility for ML workflows.
- **Data Warehouse Pattern**: Stores structured data optimized for querying and analysis. Suitable for training models on historical or aggregated data.
- **Feature Stores:** A specialized storage layer for ML features. Ensures reusability, consistency, and availability of features across teams and pipelines. Feast, Databricks Feature Store.
- **Data Streaming:** Extract data from sources, transform it (e.g., encoding categorical data), and load it into ML-friendly storage. Real-time pipelines (e.g., Kafka, Spark Streaming) enable online learning.
- **Data Lakehouse Architecture:** Optimized for both batch and real-time ML workflows: Delta Lake, Apache Iceberg.

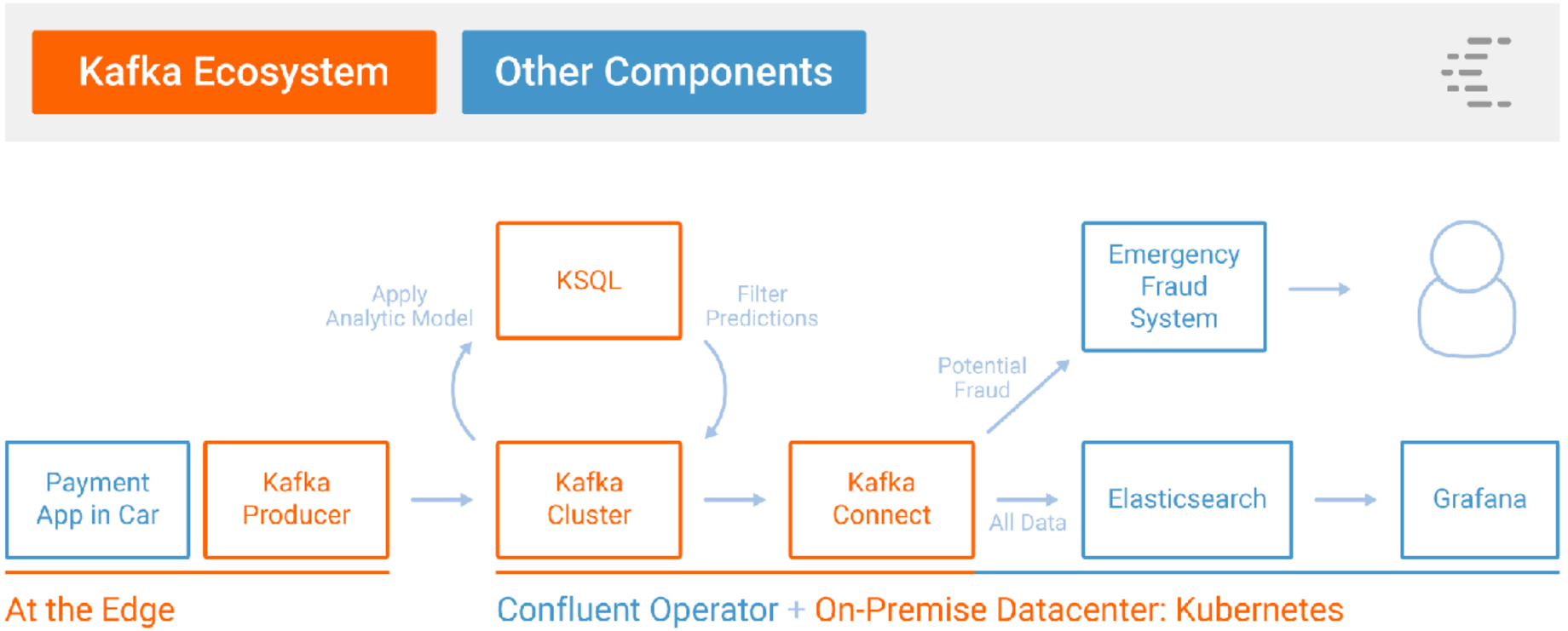
ML Specific Tools

- **Metadata Management:** Use tools to track metadata about datasets, models, and experiments. MLflow, Pachyderm.

- **Data Preprocessing and Pipeline Automation:** Orchestrate preprocessing tasks with tools like Apache Airflow or Kubeflow Pipelines. Ensure consistency in data transformations used across ML models.

KSQLDB User Defined Functions

```java
package com.example.ksql.functions;

import io.confluent.ksql.function.udf.Udf;
import io.confluent.ksql.function.udf.UdfDescription;
import io.confluent.ksql.function.udf.UdfParameter;

@UdfDescription(
    name = "reverse",
    description = "Example UDF that reverses an object",
    version = "0.1.0",
    author = ""
)
public class ReverseUdf {
  @Udf(description = "Reverse a string")
  public String reverseString(
      @UdfParameter(value = "source", description = "the value to reverse")
      final String source
  ) {
    return new StringBuilder(source).reverse().toString();
  }
}
```



https://www.confluent.io/blog/build-udf-udaf-ksql-5-0/

Creating a UDF Model with H2O

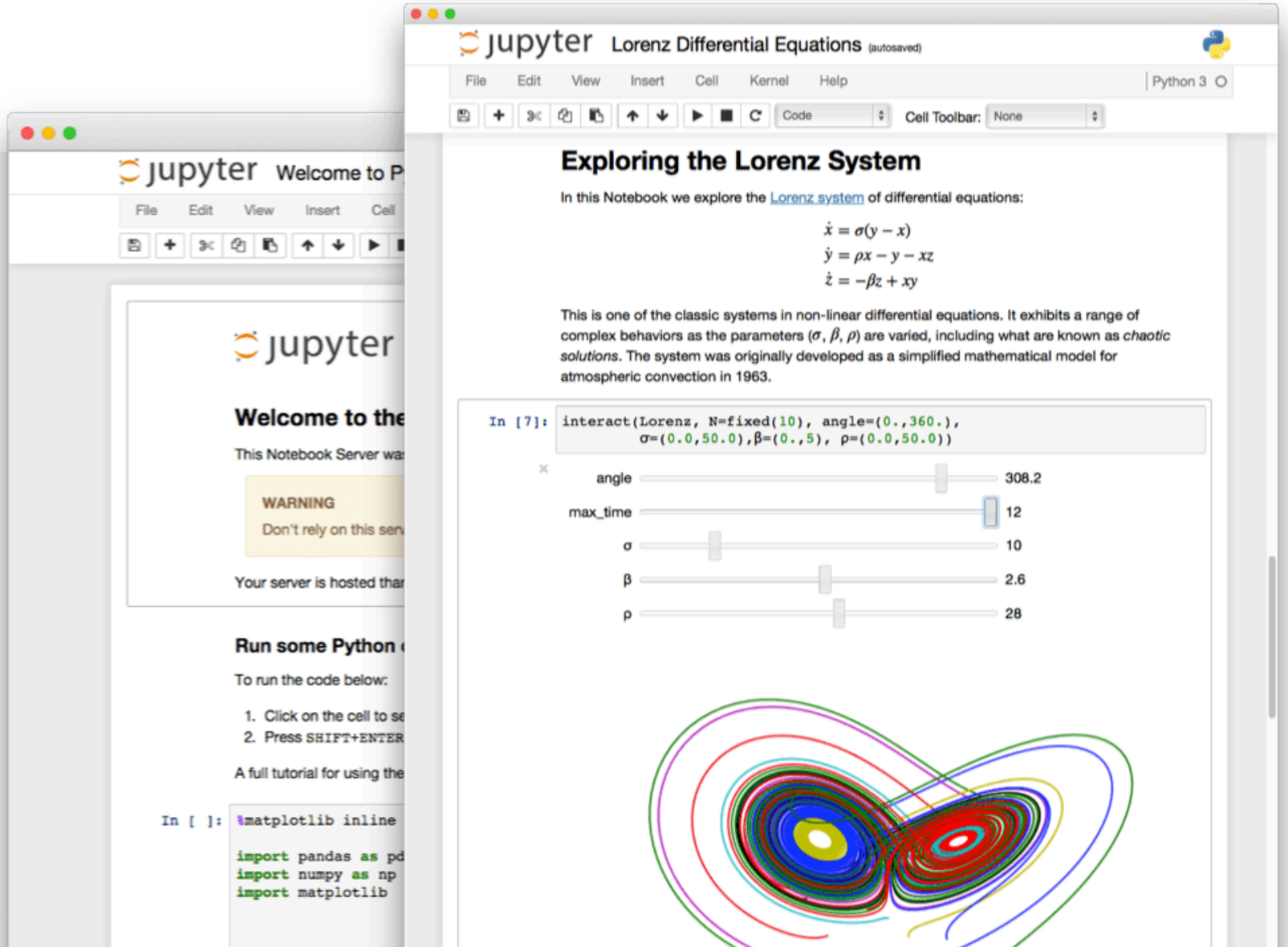
```
GenModel rawModel;

rawModel = (hex.genmodel.GenModel)
Class.forName(modelClassName).newInstance();

EasyPredictModelWrapper model = new
   EasyPredictModelWrapper(rawModel);
```

H2O.ai

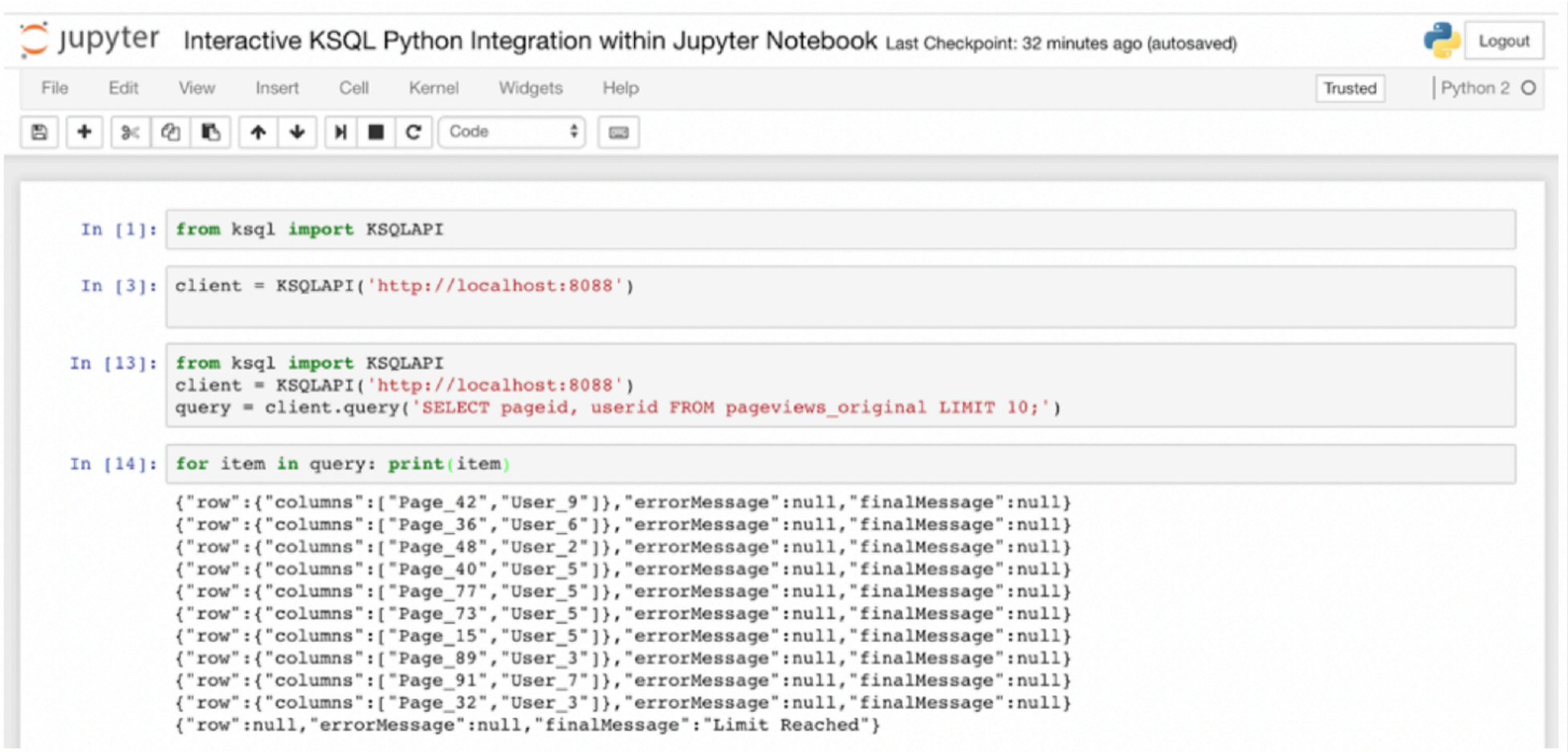
```
SELECT car_id, event_id, ANOMALY(sensor_input) FROM car_sensor;
```

Source: https://bit.ly/32zEgB0

Integrating Results to Jupyter



Viewing Real-time results in KSQLDB



This enables live use of data through the notebook to experiment with possible modelling

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Mastodon: https://mastodon.social/@dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext