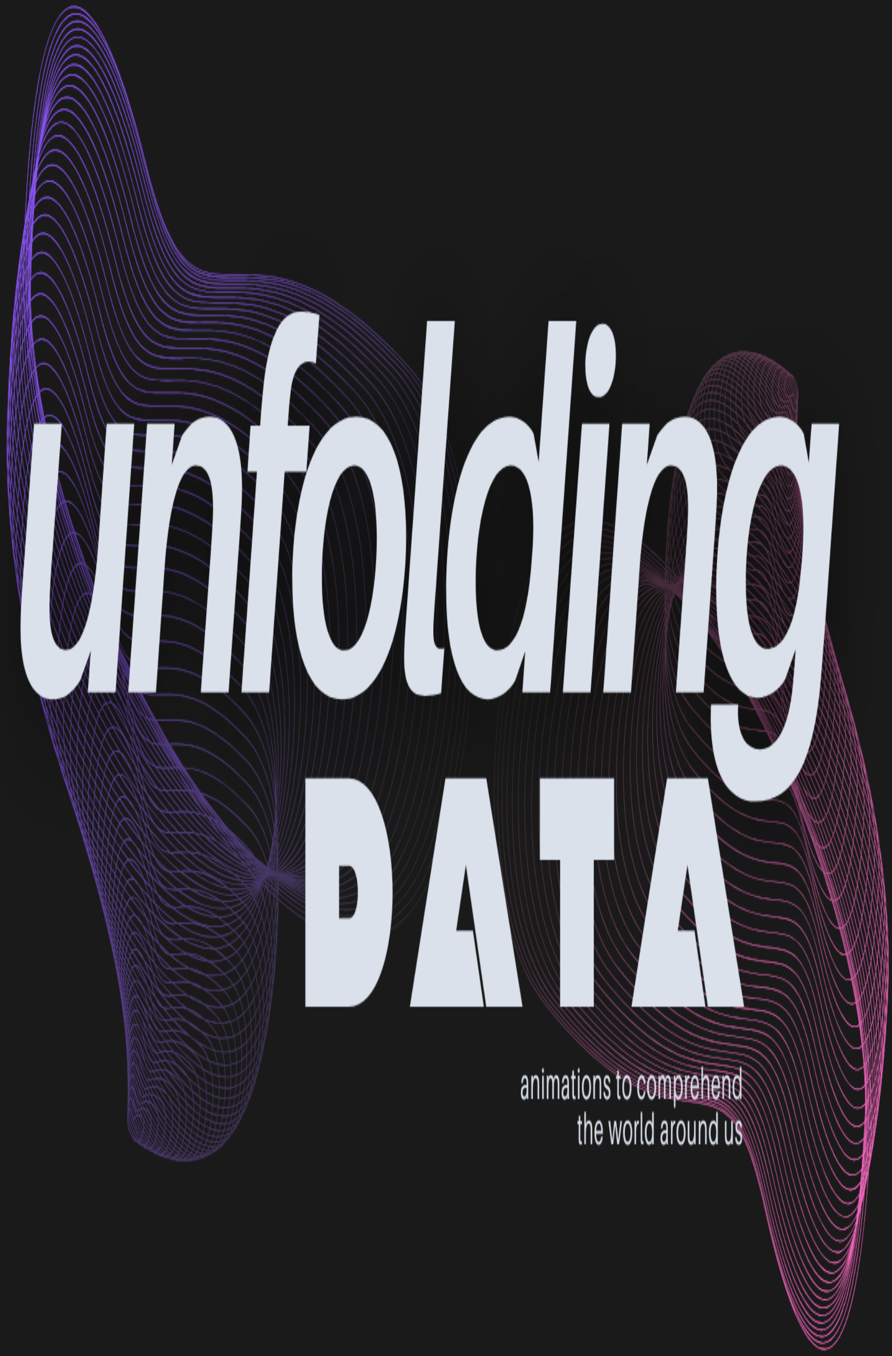
# unfolding DATA

animations to comprehend
the world around us



{{< details title="The Big Data Tech Stack, more clear than ever  " closed="true" >}}

1. Data Storage

This category compares the foundational storage layers of each stack. It's about where your raw data physically resides.

| Criterion | **HDFS** | **S3 / GCS** | **MinIO** |
|---|---|---|---|
| **Type** | Distributed File System | Cloud Object Store | Self-hosted Object Store |
| **Architecture** | Coupled Storage & Compute. Data is stored on the same nodes that process it. | Decoupled Storage & Compute. Storage and processing scale independently. | Decoupled Storage & Compute. Runs on your infrastructure but separates the two functions. |
| **Deployment** | On-premise or in the cloud (IaaS) requiring manual management. | Fully managed cloud service. | Self-hosted on-premise or in any cloud (IaaS). |
| **Scalability** | Scales horizontally by adding more servers to the cluster. | Virtually infinite scalability. You simply use as much as you need. | Horizontally scalable by adding more hardware to your cluster. |
| **Cost** | High initial hardware cost and ongoing operational overhead. | Pay-as-you-go based on storage volume and data access. | Low hardware cost (commodity servers) but high management overhead. |
| **Primary Use Case** | Legacy on-premise batch processing and big data analytics. | Modern cloud data lakes and object-based storage for any data type. | Building a private cloud data lake or avoiding cloud vendor lock-in. |

2. Data Transformation & Orchestration

This table compares the tools that handle the "T" (transformation) and "E" (orchestration) parts of the ELT/ETL process.

| Criterion | **Dataform** | **dbt** | **PySpark** | **ADF** |
|---|---|---|---|---|
| **Core Function** | SQL-based transformation for BigQuery. | SQL-based transformation for any cloud warehouse. | Code-based (Python) distributed data processing. | Visual ETL/Orchestration. |
| **Orchestration** | Basic built-in scheduler. Best used with Cloud Composer. | Basic built-in scheduler. Best used with Airflow. | Requires a separate orchestrator like Airflow or a managed service like Dataproc. | Has a robust, built-in orchestrator with a GUI. |
| **Primary Language** | SQLX (Superset of SQL). | SQL (with Jinja). | Python (with Spark SQL). | Visual GUI (with some SQL/code). |
| **Flexibility** | Limited to BigQuery and SQL-based transformations. | Highly flexible; works with most modern data warehouses. | Highly flexible; can handle complex logic and ML pipelines. | Highly integrated with Azure. Can connect to many sources. |

| Criterion | **Dataform** | **dbt** | **PySpark** | **ADF** |
|---|---|---|---|---|
| **User** | Data Analysts & Analytics Engineers on GCP. | Analytics Engineers & Data Analysts. | Data Engineers & Data Scientists. | Data Engineers & Business Users. |

3. Query Engines & Data Warehouses

This compares the tools that serve as the compute and storage layers for analytics.

| Criterion | **BigQuery** | **Snowflake** | **Trino** | **Spark** |
|---|---|---|---|---|
| **Type** | Serverless Data Warehouse | Cloud Data Warehouse | SQL Query Engine | General-Purpose Analytics Engine |
| **Architecture** | Separated storage and compute, managed by Google. | Separated storage and compute, managed by Snowflake. | Separate compute layer that queries multiple data sources. | Separate compute layer for distributed processing. |
| **Deployment** | Fully managed service on GCP. | Fully managed service (multi-cloud). | Self-hosted on-premise or in the cloud. | Self-hosted on a cluster or via managed service (e.g., GCP Dataproc). |
| **Primary Use Case** | Scalable, serverless analytics on a massive scale. | Enterprise-grade data warehousing for structured data. | Federated queries across diverse data sources. | ETL, machine learning, and complex transformations. |

4. BI & Visualization

This category compares the tools used by end-users to create dashboards and reports.

| Criterion | **Looker** | **Superset / Redash** | **Grafana** |
|---|---|---|---|
| **Core Function** | Semantic Modeling & Visualization. | Visualization & Ad-hoc Querying. | Time-series Monitoring & Visualization. |
| **Data Philosophy** | **Code-first semantic layer**. Data is modeled in LookML for consistency. | **SQL-based visualization**. Users often write their own queries or work from a pre-defined set of queries. | **Monitoring-focused**. Optimized for time-series data from a variety of data sources. |
| **User** | Data Analysts & Business Users. | Data Analysts & SQL-savvy users. | DevOps & Engineers (for system metrics). |
| **Collaboration** | Highly collaborative via Git integration. | Collaborative dashboards via sharing. | Collaborative dashboards and alerting. |
| **Ecosystem** | Google Cloud's official BI tool. | Open-source, flexible, and database-agnostic. | Open-source, widely used for observability and metrics. |

5. Table Format & Catalog

Apache Iceberg and Project Nessie are key components of a modern, open-source data lakehouse architecture. They don't fit neatly into the previously defined categories because they represent a different layer of the stack: **the table format and the catalog**.

See the tools that bring structure, transactions, and version control to a data lake.

| Criterion | **Apache Iceberg** | **Project Nessie** |
|---|---|---|
| **Type** | Open-source **Table Format** | Open-source **Data Catalog** |
| **Core Function** | Defines how data files in a data lake are organized to act as a database-like table. | A Git-like central registry for managing metadata and versions for all your tables. |
| **Primary Use Case** | Enables ACID transactions, time travel, and schema evolution on a single table in a data lake. | Manages multi-table transactions, data branching, and version control for the entire data lake. |
| **Data Philosophy** | Brings database-like reliability and performance to data lakes. | Brings software development best practices (Git) to data management and governance. |
| **Relationship** | **A table format.** It's the "engine" that enables a single table to have ACID properties and time travel. | **A catalog.** It's the "control plane" that manages the state of many Iceberg tables using branches, commits, and tags. |
| **Deployment** | Not a service. It's a library or a specification used by engines like Spark, Flink, and Trino. | Can be run as a Docker container, on Kubernetes, or as a managed service. |
| **Who Uses It** | Data Engineers & Analytics Engineers. | Data Engineers, Analytics Engineers, & Data Scientists (for branching). |

{{< /details >}}

{{< details title="Nessie and Apache Iceberg were really cool " closed="true" >}}

Project Nessie and Apache Iceberg are designed to work very well together, and Nessie can indeed be used on top of Iceberg.

- **Apache Iceberg:**
  - This is a table format for massive analytic datasets. It brings table-like capabilities to data lakes, enabling features like ACID transactions, schema evolution, and time travel.
  - Essentially, Iceberg helps organize your data lake into tables that behave more like traditional database tables.
- **Project Nessie:**
  - Nessie provides Git-like version control for your data lake. It allows you to create branches, commits, and tags for your data, making it easier to manage changes and collaborate.
  - In the context of Iceberg, Nessie acts as a catalog that tracks the metadata of your Iceberg tables. This allows you to version the state of your tables, not just the underlying data files.

**How They Work Together:**

- Nessie enhances Iceberg by providing a way to manage the metadata of Iceberg tables. This means you can:
  - Create branches to experiment with data transformations without affecting the production data.
  - Roll back to previous versions of your tables if needed.
  - Collaborate with others by merging changes from different branches.

  In essence, Iceberg provides the table format, and Nessie provides the version control layer on top of those tables.

  This combination creates a powerful and flexible **data lakehouse architecture**.

{{< /details >}}

{{< details title="Nessie and Apache Iceberg were really cool | HDFS and PySpark " closed="true" >}}

Project Nessie and Apache Iceberg are designed to work very well together, and Nessie can indeed be used on top of Iceberg.

- **Apache Iceberg:**
  - This is a table format for massive analytic datasets. It brings table-like capabilities to data lakes, enabling features like ACID transactions, schema evolution, and time travel.
  - Essentially, Iceberg helps organize your data lake into tables that behave more like traditional database tables.
- **Project Nessie:**
  - Nessie provides Git-like version control for your data lake. It allows you to create branches, commits, and tags for your data, making it easier to manage changes and collaborate.
  - In the context of Iceberg, Nessie acts as a catalog that tracks the metadata of your Iceberg tables. This allows you to version the state of your tables, not just the underlying data files.

**How They Work Together:**

- Nessie enhances Iceberg by providing a way to manage the metadata of Iceberg tables. This means you can:
  - Create branches to experiment with data transformations without affecting the production data.
  - Roll back to previous versions of your tables if needed.
  - Collaborate with others by merging changes from different branches.

  In essence, Iceberg provides the table format, and Nessie provides the version control layer on top of those tables.

  This combination creates a powerful and flexible **data lakehouse architecture**.

- **HDFS + PySpark:**
  - This is a more traditional big data stack, often used in on-premises deployments. HDFS stores the data, and PySpark allows for Python-based data processing.
- **Databricks + S3:**
  - This is a common cloud-based stack. Databricks provides the processing power (Spark), and S3 serves as the scalable data lake.
- **MinIO + Apache Iceberg + Nessie:**
  - This stack focuses on open-source technologies. MinIO provides object storage, Iceberg enables efficient data lake management, and Nessie adds version control (and time travel)

{{< /details >}}

{{< details title="Core Components of a Data Analytics Tech Stack " closed="true" >}}

- **Data Storage:**: This is where the raw and processed data resides. Common options include:
  - **Cloud Storage:** Amazon S3, Google Cloud Storage (GCS), Azure Blob Storage.
    - These are highly scalable and cost-effective **object stores** that serve as the foundation of cloud-native **data lakes**.
    - **Object Storage:** MinIO, which is an open source object storage solution that is S3 API compatible.
  - **Data Lakes:** HDFS (Hadoop Distributed File System), which is often used in on-premises or hybrid environments.
    - You will find format like: `avro`, `parquet` and `delta`.
  - **Data Warehouses:** Snowflake, Google BigQuery, Amazon Redshift or Azure Synapse. These are optimized for analytical queries.
- **Data Processing:** These tools transform and prepare data for analysis:
  - **Apache Spark:** A powerful distributed processing engine used for large-scale data transformation and analysis. It includes modules like **Spark SQL** for structured data, **Spark Streaming** for real-time data, and **MLlib** for machine learning.
  - **Databricks:** A cloud-based platform built on Apache Spark, providing a collaborative environment for data engineering, data science, and machine learning.
    - It also offers features like **Delta Lake** for data reliability and **MLflow** for managing the machine learning lifecycle.
  - **SQL-based tools:** These are essential for querying and manipulating data in data warehouses. This category includes modern tools that use SQL for transformations:

        * **dbt (data build tool):** A transformation tool that enables data analysts and engineers to transform data in their data warehouses more effectively. It allows for modular SQL code and follows software engineering best practices.
            - Dbt wont move the data for you, so you need an orchestration tool like ADF or TalenD
        * **Dataform:** Google's equivalent of dbt, used for transforming data in BigQuery using **SQLX**.
- **Data Orchestration:**
    - These tools manage the flow of data through the pipeline:
        * **Apache Airflow:** A platform for programmatically authoring, scheduling, and monitoring workflows.
            - It defines pipelines as **Directed Acyclic Graphs (DAGs)**.
        * **Cloud Composer:** Google Cloud's fully managed version of Apache Airflow, which simplifies its deployment and management.
        * **Azure Data Factory (ADF):** Microsoft's visual ETL and orchestration service, offering a code-free approach to building data pipelines within the Azure ecosystem.
- **Data Catalog and Governance:** These tools help manage and govern data assets:
    - **Apache Iceberg:** An open table format for large analytic datasets. It provides features like **ACID transactions**, schema evolution, and time travel.
        * See also Delta Lake Format: https://delta.io/ which is some kind of .parquet with enhance time travel capabilities
    - **Project Nessie:** Provides **Git-like capabilities** (branches, commits, tags) for data lakes, working in conjunction with table formats like Apache Iceberg to provide full data versioning and governance.
    - **Apache Hive Metastore:** A central repository for metadata, primarily used in the Hadoop ecosystem. It stores schema and location information for Hive tables.

```
df = spark.read
     .format("delta")
     .option("timestampAsOf", "2025-12-01")
     .load("/path/to/my/table")
```

- **Data Visualization and Business Intelligence (BI):**
    - These tools allow users to explore and visualize data:
        * **Tableau, Power BI, Looker:** Popular BI platforms for creating dashboards and reports.

**Key Considerations:**

- The choice of tech stack depends on factors such as:
    - The volume and velocity of data.
    - The specific analytical needs of the organization.
    - Budget constraints.
    - Whether the deployment is on-premises, cloud-based, or hybrid.

{{< /details >}}

{{< callout type="info" >}} You can plug to those stacks interesting tools like superset or (Open Data Hub) {{< /callout >}}

{{< details title="DWH vs Databases " closed="true" >}}

The data modeling approach is a primary differentiator between the two systems: DB vs DWH.

A data warehouse can be thought of as a specialized database with its tables designed in a way that prioritizes analytical performance over transactional efficiency.

You can turn a PostgreSQL database into a data warehouse with proper data modeling and optimization, especially for small to medium-sized data volumes. While it wasn't designed for this purpose, its flexibility and extensibility make it a viable option for many use cases.

The key is to change how you approach the database from a **transactional (OLTP)** mindset to an **analytical (OLAP)** one.

**Data Modeling and Schema**

The most critical step is to use an analytical data model.

- **Star Schema**: Instead of the highly normalized schema used for OLTP, you would model your data using a star schema. This design separates data into a central **fact table** (which contains measurements like sales or transactions) and multiple, smaller **dimension tables** (which contain context like product, customer, or time). This structure makes analytical queries much more efficient.
- **Denormalization**: You would also intentionally denormalize your data. This means duplicating some data to avoid complex joins, which are performance-killers in an analytical context.

**Facts vs. Dimensions**

The core of data warehousing is separating data into **facts** and **dimensions**.

Think of it as answering the questions “What happened?” and “In what context did it happen?”.

- **Facts** are the **measurable events** or metrics of a business process. They are the **“what happened”** of your data. Facts are typically numerical values that can be counted, summed, or averaged. They are stored in a **fact table**, which is often very large and contains a row for every single event or transaction.
  - **Example**: In a retail company, a fact would be a **sale**. The fact table would store the **sales amount**, `quantity sold`, or `profit` for each transaction.
- **Dimensions** provide the **context** for the facts. They are the **“who, what, where, when, and how”** of your data. Dimensions contain descriptive attributes that you use to filter, group, and analyze your facts. They are stored in **dimension tables**, which are usually much smaller than fact tables.
  - **Example**: For that same retail sale, the dimensions would be the `Product` (what was sold), `Customer` (who bought it), `Store` (where it was sold), and `Date` (when it was sold). The dimension tables would contain descriptive information like the `product_name`, `customer_name`, `store_location`, and `date_of_week`.

The fact table’s rows link to the dimension tables using foreign keys. This allows you to answer analytical questions like, “What was the total sales amount (`fact`) for Nike brand products (`dimension`) in the last quarter (`dimension`)?”

**Optimizations and Extensions**

In addition to data modeling, you would need to implement several optimizations and use extensions to overcome PostgreSQL’s native limitations for large-scale analytics:

- **Partitioning**: For very large tables, you would use **table partitioning** to split data into smaller, more manageable pieces based on criteria like date or region. This allows queries to scan only the relevant partitions, drastically improving performance.
- **Materialized Views**: You can create **materialized views** to pre-compute and store the results of complex queries. This is perfect for dashboards and reports that are run frequently, as users can query the pre-calculated view instead of the raw data.
- **Parallelism**: Since PostgreSQL v10, it has supported parallel query execution. You can tune settings to allow the database to use multiple CPU cores to process large, complex queries faster.
- **Extensions**: PostgreSQL’s power lies in its extensions. Extensions like **Citus** can turn a single PostgreSQL instance into a distributed database cluster, allowing it to scale horizontally for massive datasets. Other extensions, like `cstore_fdw`, can provide **columnar storage**, which is natively optimized for analytical workloads by allowing queries to read only the specific columns they need.

{{< /details >}}