# Big Data

# Data Sources

- User generated content (social/web/mobile)
  - Twitter, Facebook, Snapchat, WhatsApp
  - Clickstream
  - Financial transactions
- Internet of Things (IoT)
  - Wind turbines, oil rigs
  - Wearables
  - Smart cars
  - Home appliances

# Big Data


INSTAGRAM
USERS LIKE
1,736,111
POSTS
YOUTUBE
USERS UPLOAD
300 HOURS
OF NEW VIDEO
every
MINUTE
of the DAY
Facebook
USERS LIKE
4,166,667
POSTS
REDDIT
USERS CAST
18,327
VOTES
TWITTER
USERS SEND
347,222
TWEETS
Social Media

# Use Cases

- **Web and e-tailing**
  - Recommendation Engines
  - Ad Targeting
  - Search Quality
  - Abuse and Click Fraud Detection



- **Government**
  - Fraud Detection and Cyber Security
  - Welfare Schemes
  - Justice


- **Telecommunications**
  - Customer Churn Prevention
  - Network Performance Optimization
  - Calling Data Record (CDR) Analysis
  - Analysing Network to Predict Failure



- **Healthcare and Life Sciences**
  - Health Information Exchange
  - Gene Sequencing
  - Serialization
  - Healthcare Service Quality Improvements
  - Drug Safety


# Use Cases

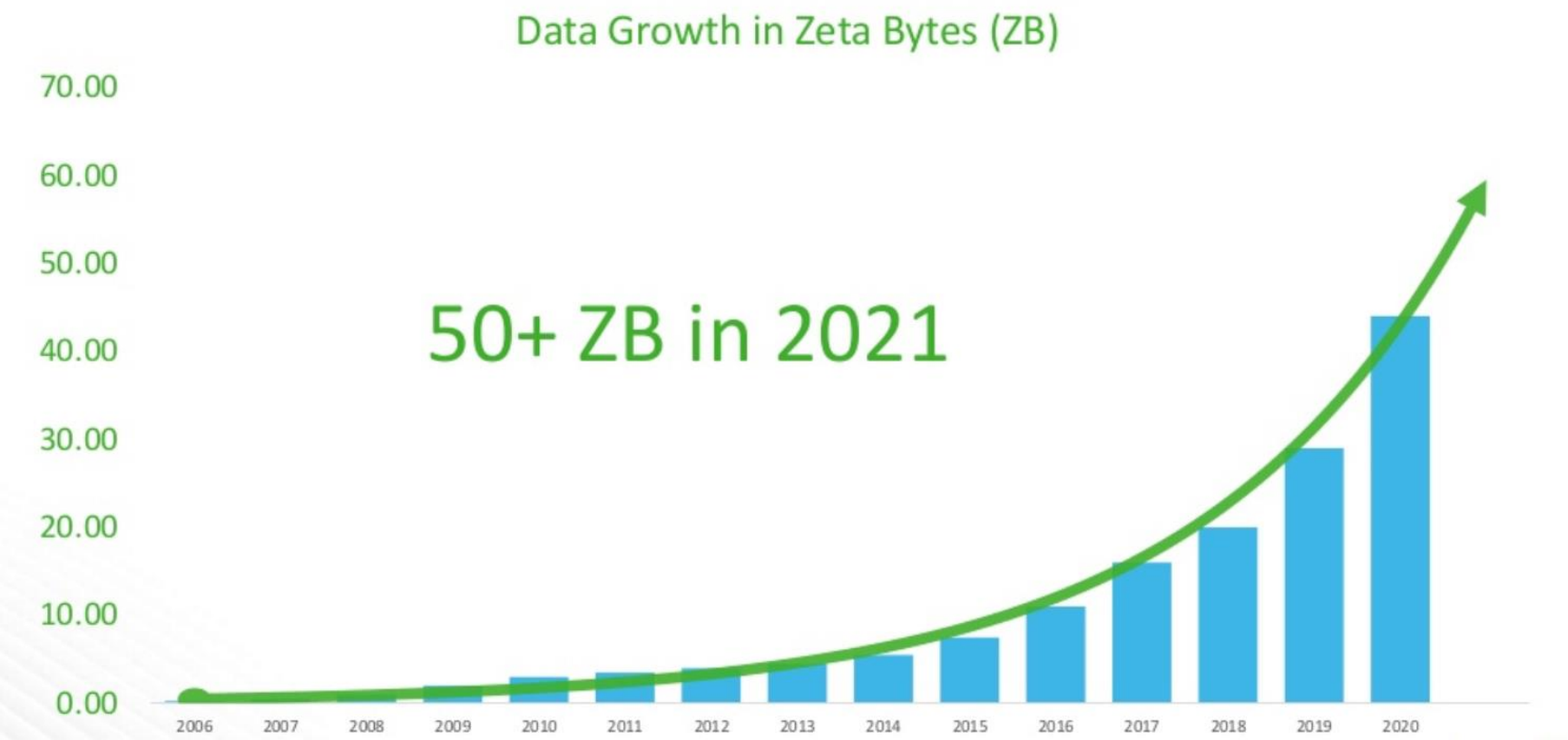
- **Banks and Financial services**
  - Modeling True Risk
  - Threat Analysis
  - Fraud Detection
  - Trade Surveillance
  - Credit Scoring and Analysis

- **Transportation**
  - Surge Pricing
  - Ride pooling
  - Revenue Management
  - Traffic control
  - Route planning
  - Logistics

- **Retail**
  - Point of Sales Transaction Analysis
  - Customer Churn Analysis
  - Sentiment Analysis

- **Hotels and Food Delivery Services**
  - Customer Demands
  - Details of Customers
  - Availability and Seasonal Data Changes

# Growth of Data



# Big Data?

- Growth in data volumes
- Limitless types of data sources
- Faster computation
- Scalability
- Support for stream processing
- Big Data / Data Science
- Cost

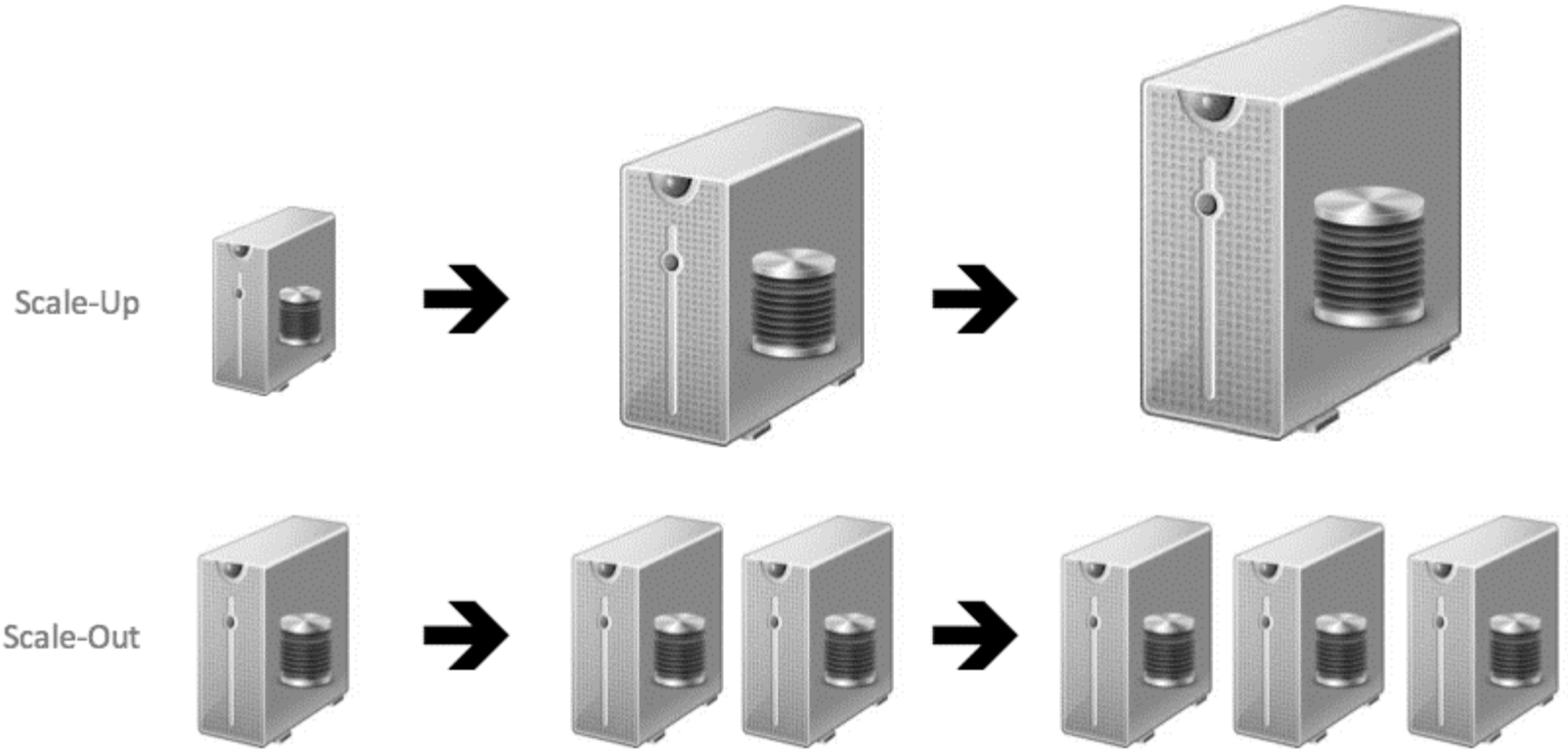
# Types of Data

- Cold / Archival Data (>4 hrs)
  - Warehouse type data
  - Historical, large data sets
  - Batch processed
  - Low cost

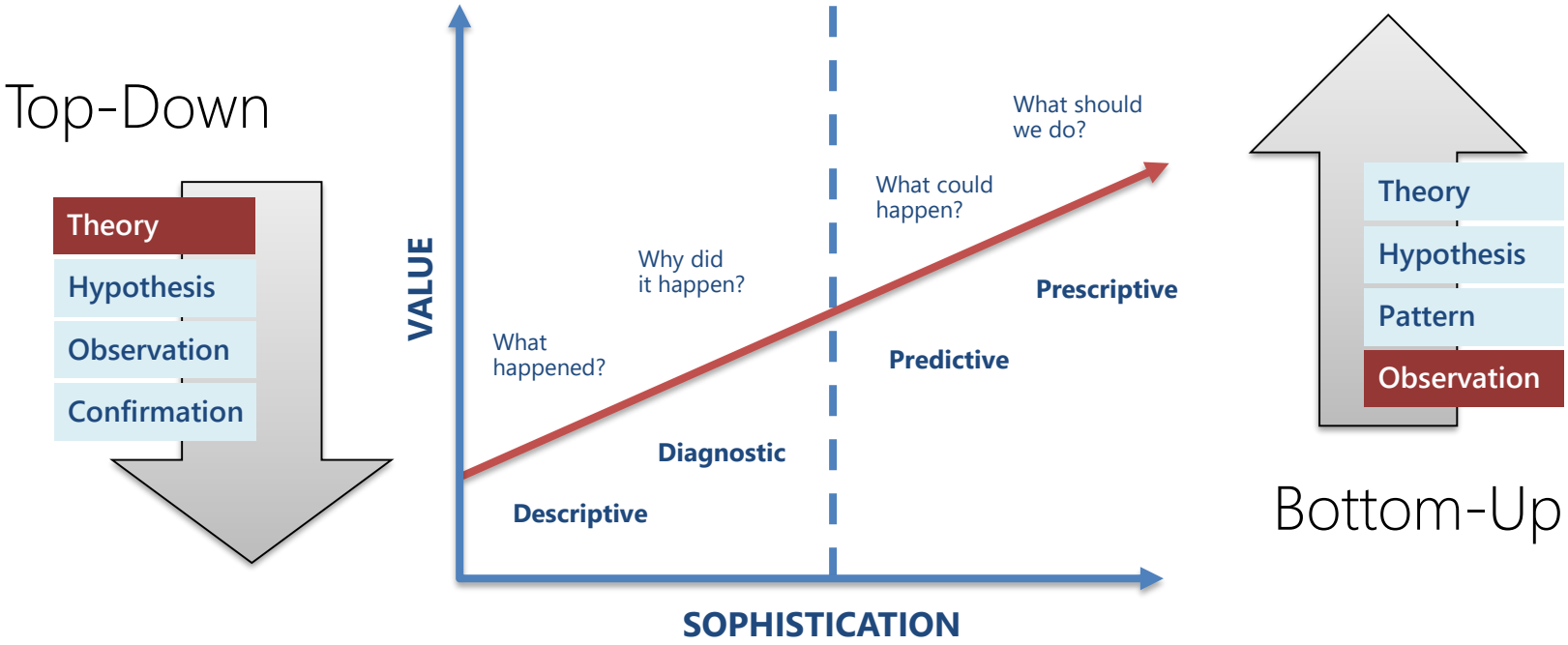
- Warm Data (<4 hrs, typically 1 hr target)
  - Operational data
  - Usually limited data
  - Latency determined by how much transformation is needed
  - Streaming or Batch processed
  - Medium cost

- Hot Data (<60 seconds)
  - Alerts, Notifications etc.
  - Very little transformations
  - Streaming data
  - High cost

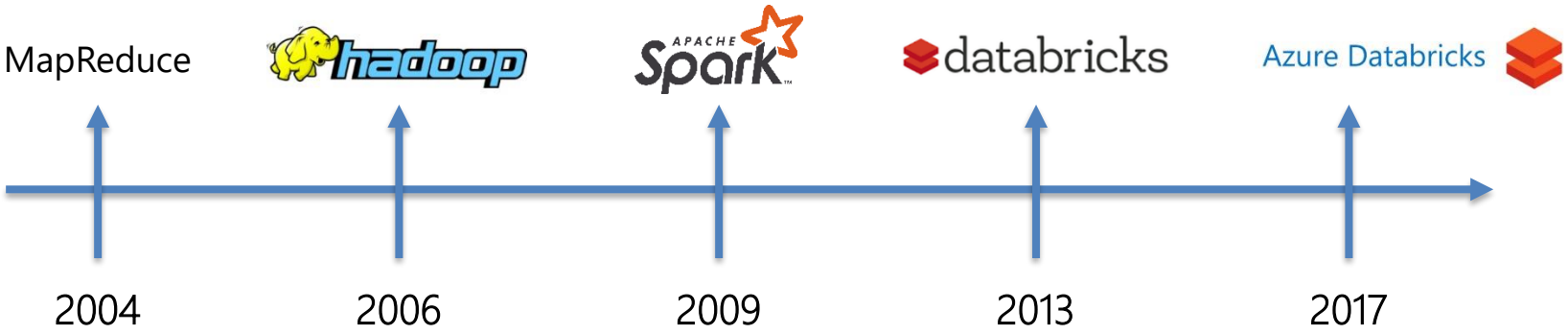
# Scale-up vs Scale-out



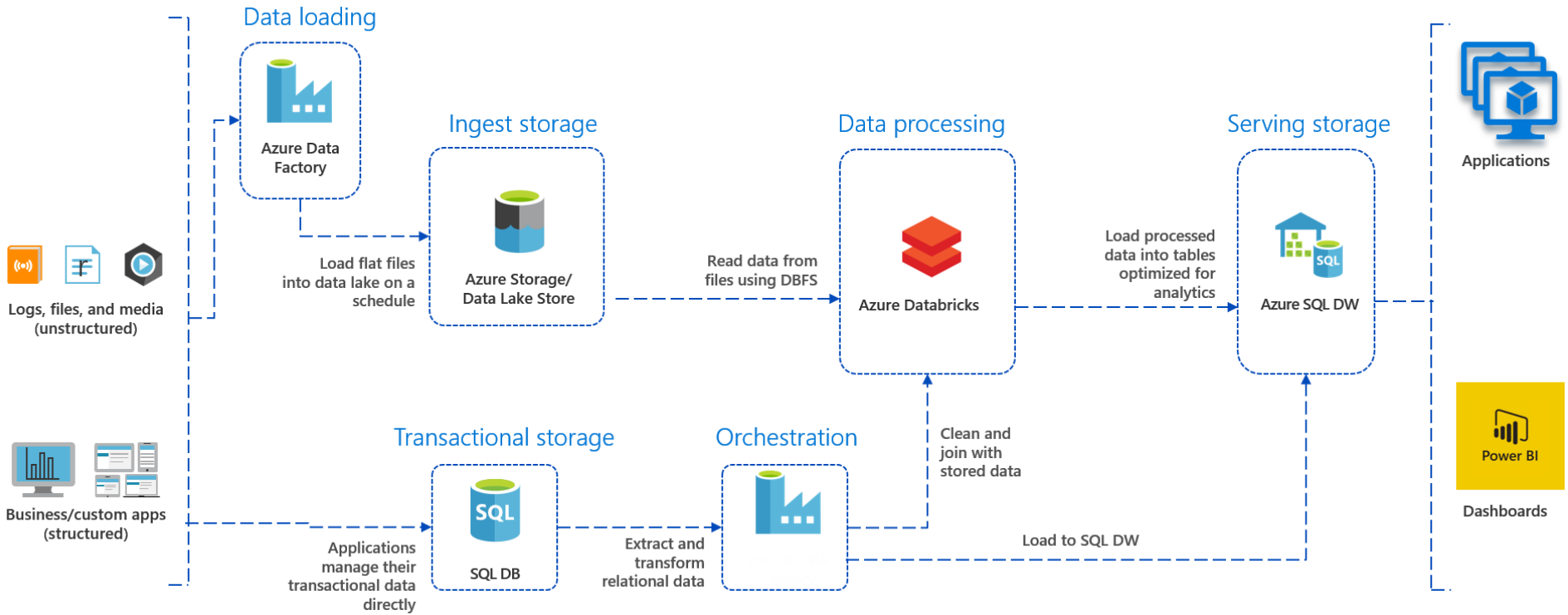
# Analytics


Top-Down
Theory
Hypothesis
Observation
Confirmation
VALUE
What happened?
Why did it happen?
What could happen?
What should we do?
Descriptive
Diagnostic
Predictive
Prescriptive
SOPHISTICATION
Theory
Hypothesis
Pattern
Observation
Bottom-Up

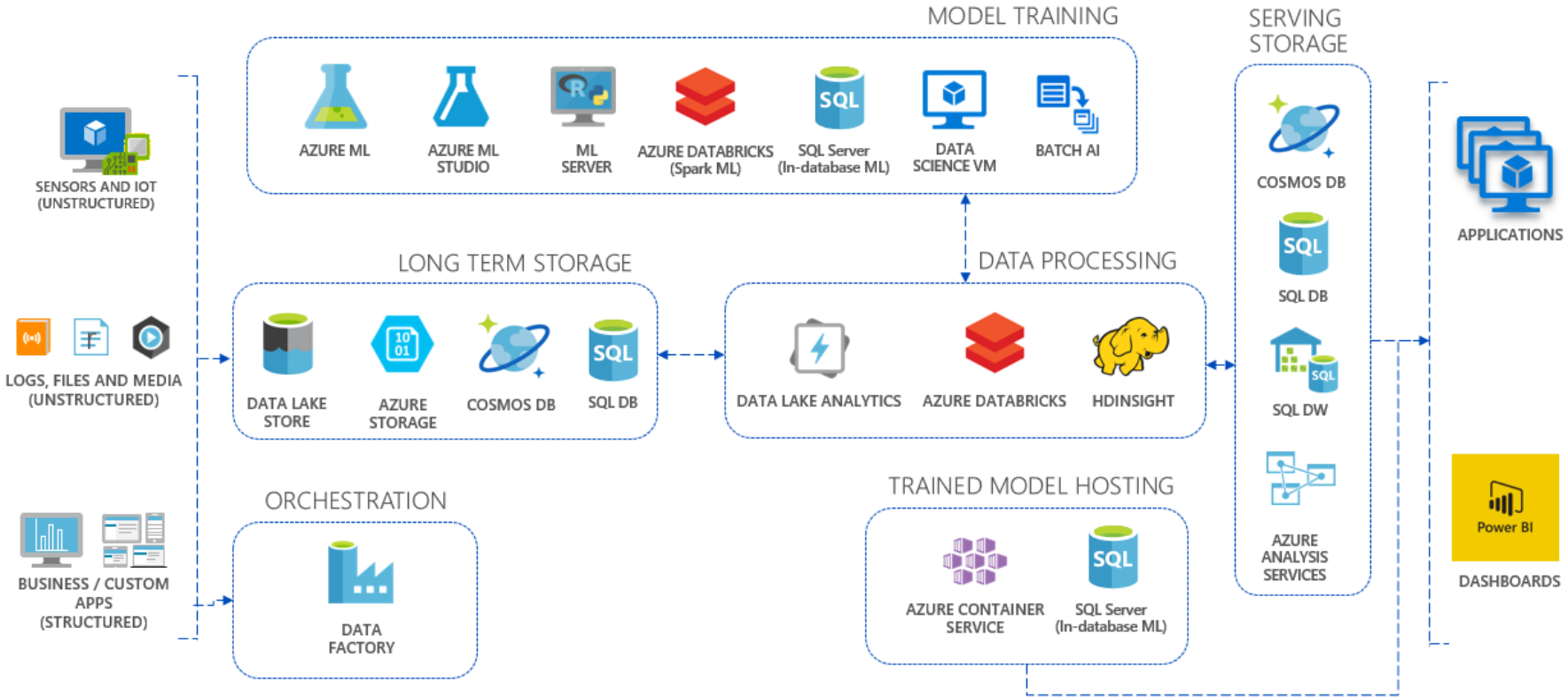
# History


MapReduce
hadoop
APACHE Spark
databricks
Azure Databricks
2004
2006
2009
2013
2017

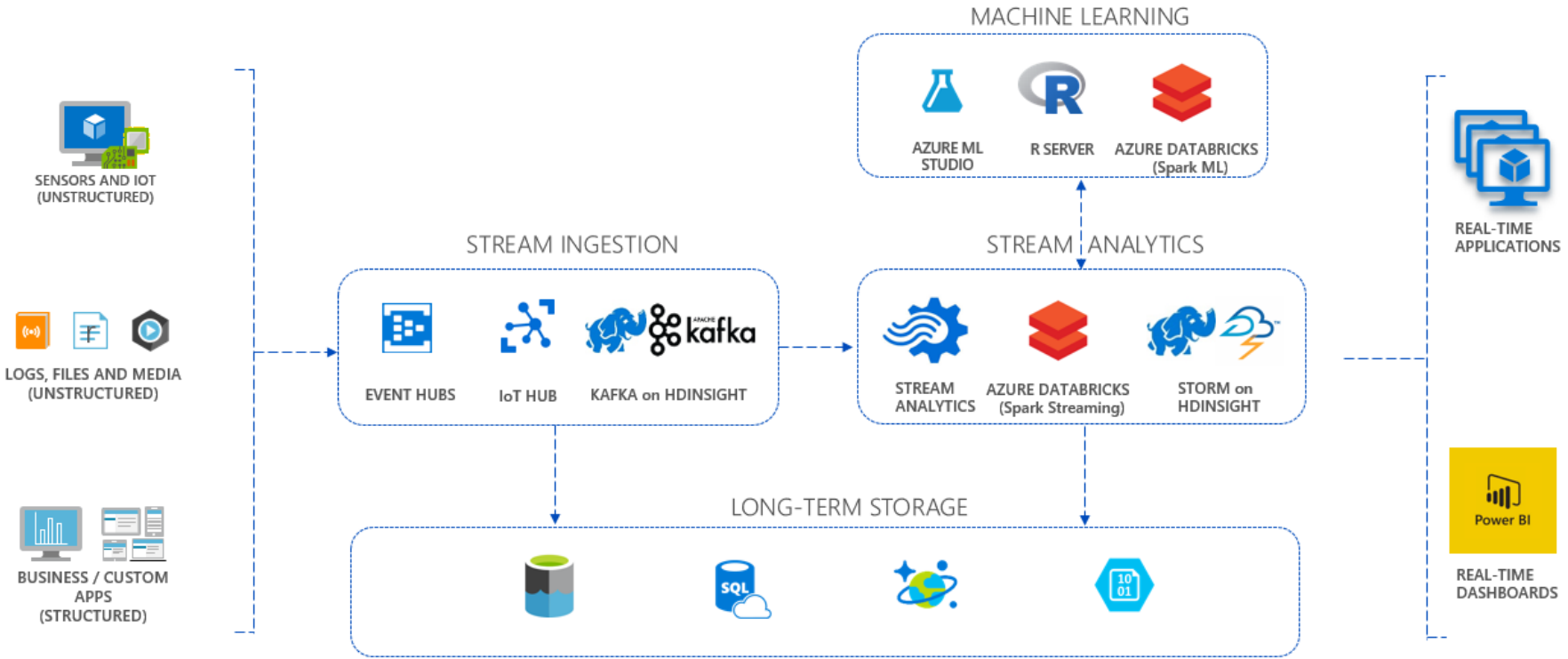
# Modern Data Warehousing



# Advanced Analytics

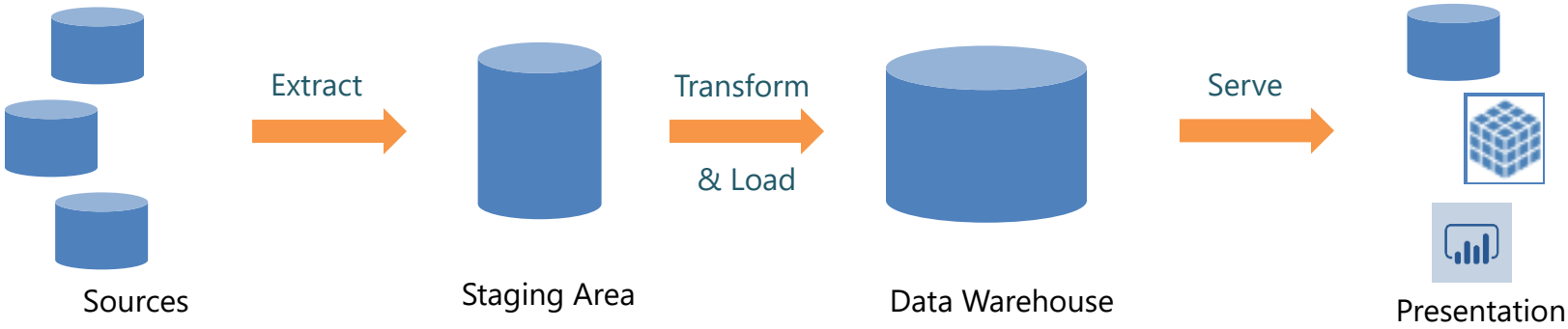


# Real Time Analytics



# ETL / ELT

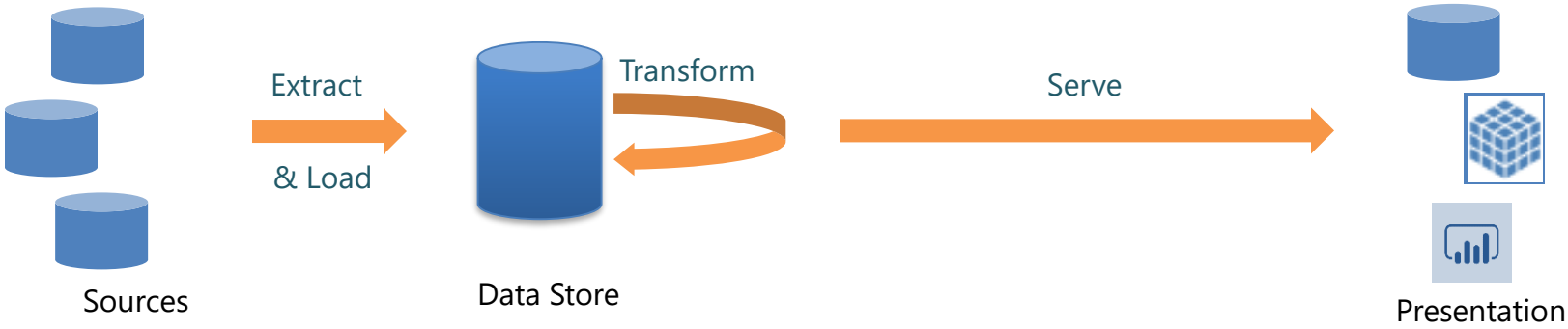
# ETL – Extract, Transform, Load




## Challenges with traditional tools

- Forced schema
- Latency
- Data duplication
- Scale up only

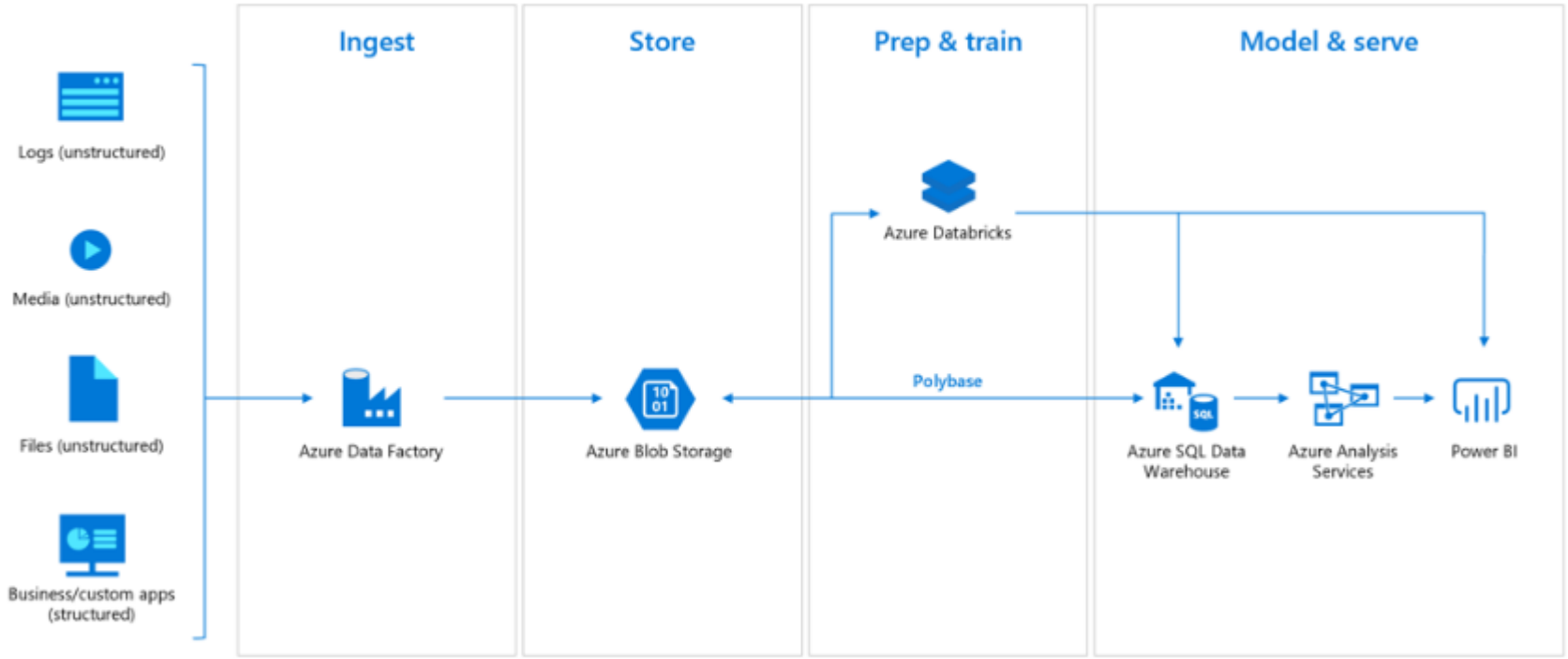
# ELT – Extract, Load, Transform




## Benefits

- Store source data as-is
- Reduced latency
- Single source of truth
- Scale out
- Store structured & unstructured data
- Support for Big Data / Data Science scenarios

# ETL using Azure Databricks

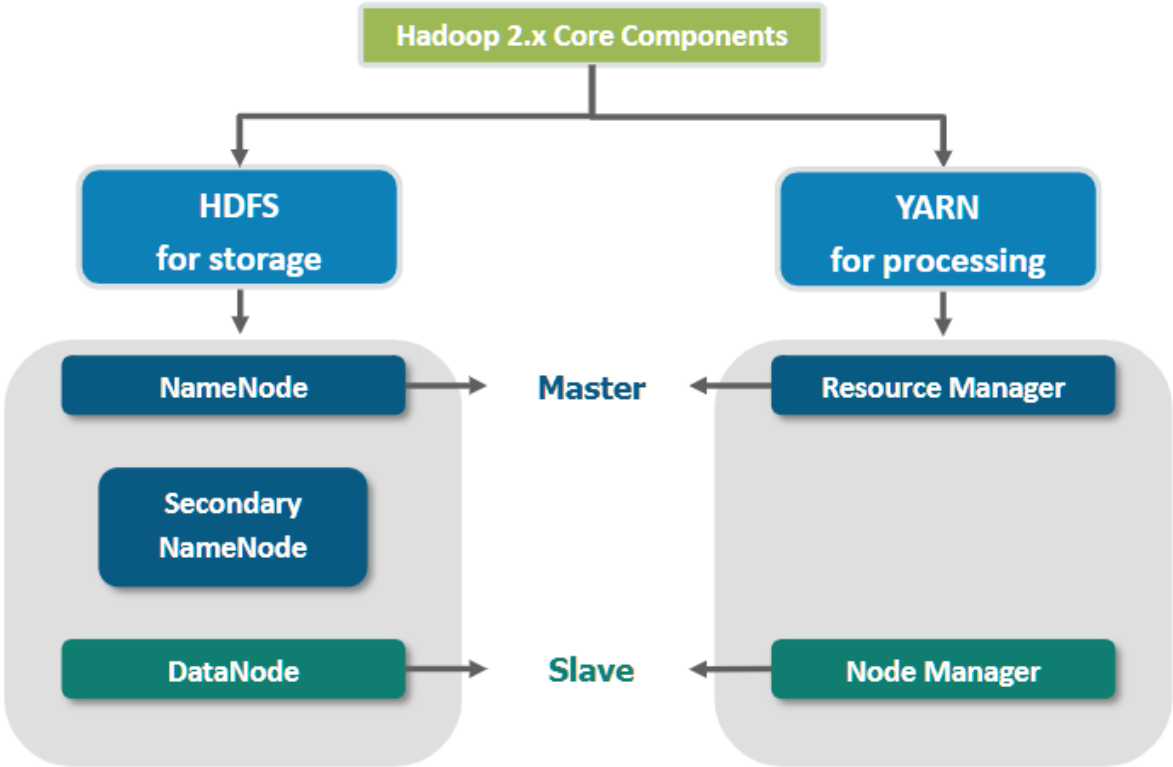


# ETL Pipeline

```
import org.apache.spark.sql.functions._

// Extract
val extractedTradingData = spark
  .read
  .option("header", "true")
  .option("inferSchema", "true")
  .csv("/mnt/storage/TradingFiles/*.csv")

// Transform
val transformedTradingData = extractedTradingData
    .select("timestamp", "instrument_token", "last_price")
    .withColumnRenamed("instrument_token", "Instrument")
    .withColumnRenamed("last_price", "TradePrice")
    .withColumn("TradeDate", $"timestamp".cast("date"))
    .groupBy($"TradeDate", $"Instrument")
    .agg(
        min("TradePrice").alias("LowPrice"),
        max("TradePrice").alias("HighPrice")
    )

// Load
transformedTradingData
  //.coalesce(1)
  .write
  .option("overwrite", "true")
  .option("header", "true")
  .json("/mnt/datalake/TransformedTradingData.json")
```

# Hadoop

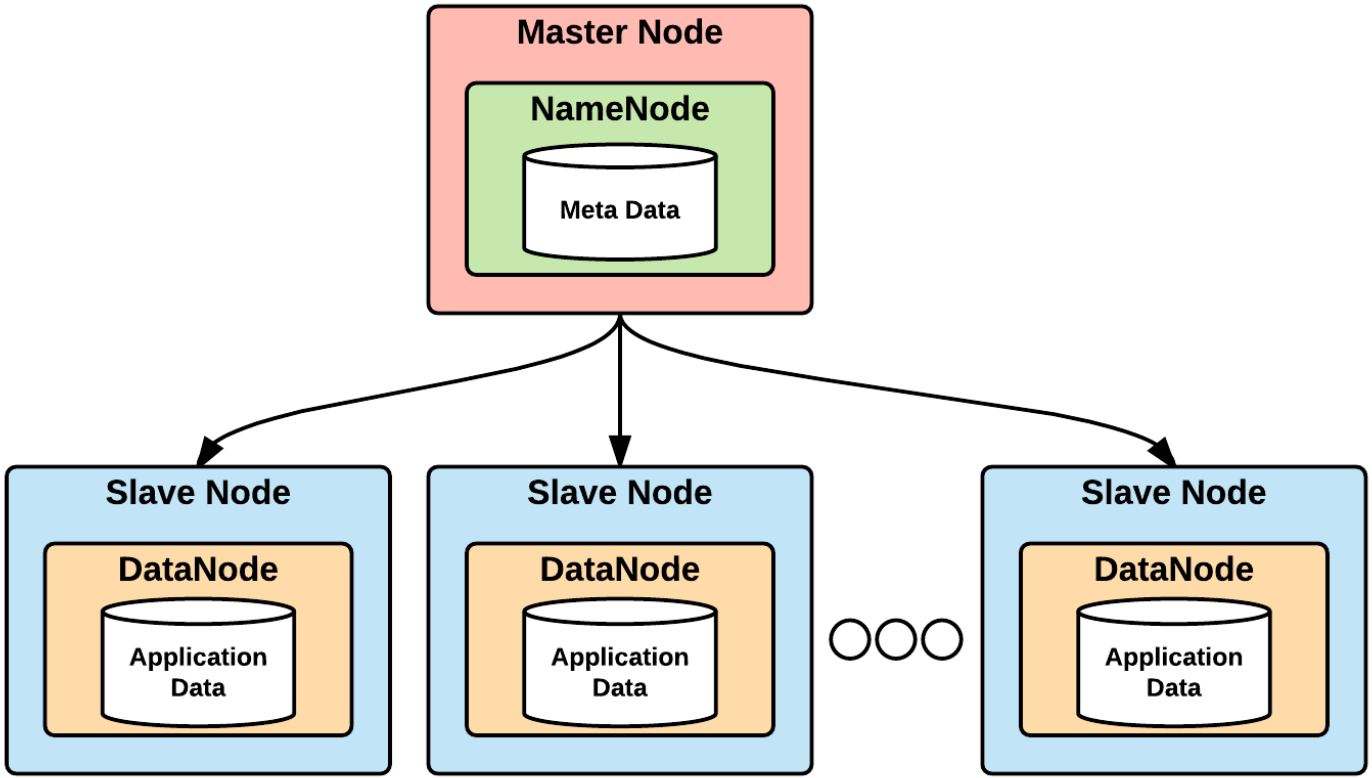
# Hadoop Core Components



# Hadoop Distributed File System

- Used to store huge volumes of data
- Distributed file system that handles large data sets running on commodity hardware
- HDFS cluster has a NameNode and DataNodes
- NameNode contains metadata
- Highly fault tolerant – Data is stored in multiple locations
- Data is divided into blocks of 128MB (default)
- Default 3 copies are created for each block

# HDFS – Master/Slave Architecture



# HDFS Architecture

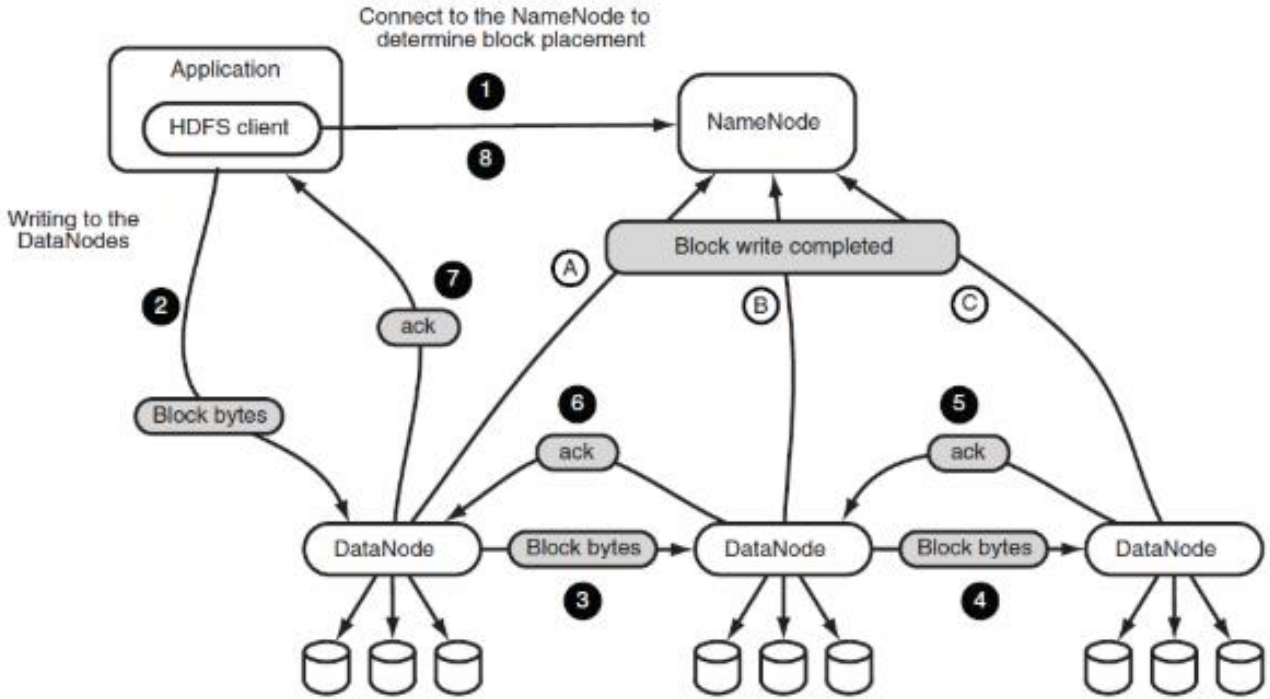
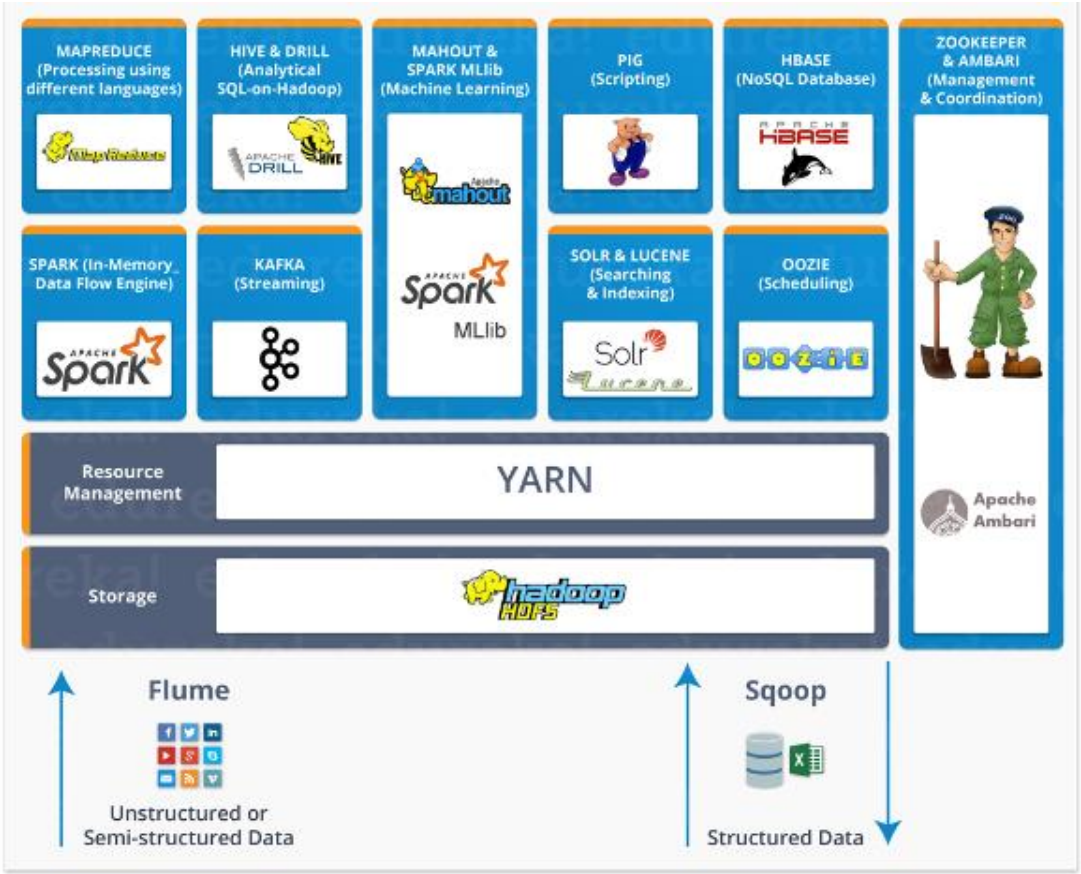



Figure C.2 **HDFS write data flow**

# Hadoop Ecosystem

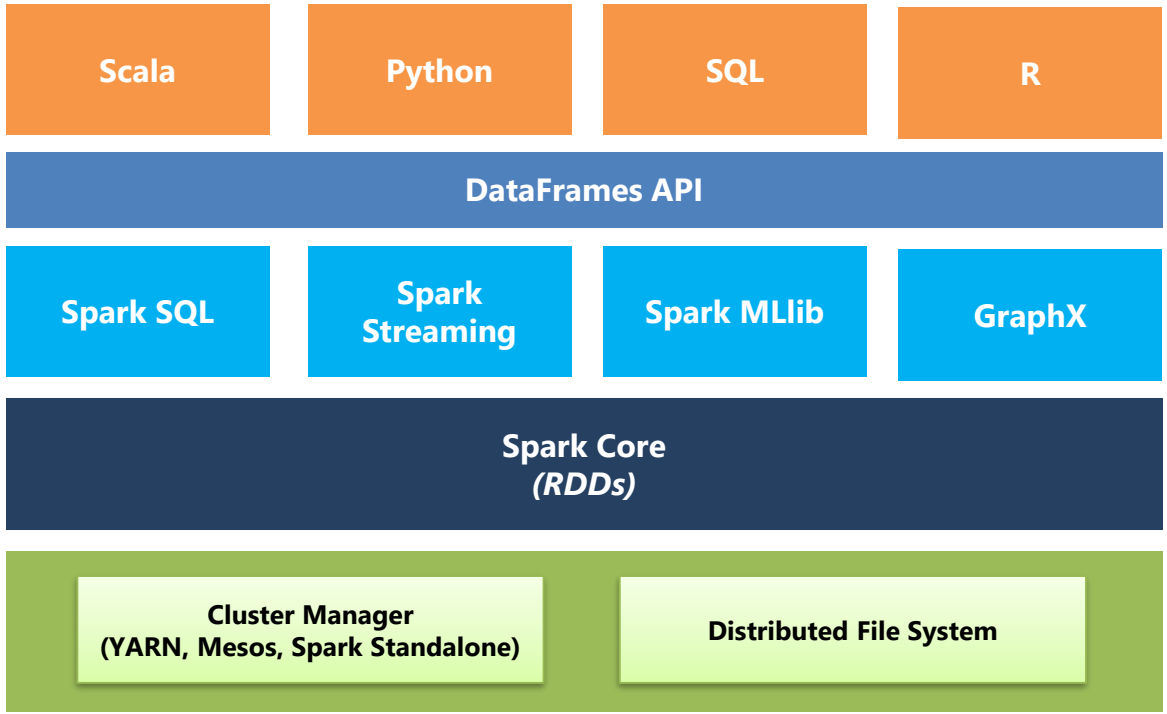


# Spark

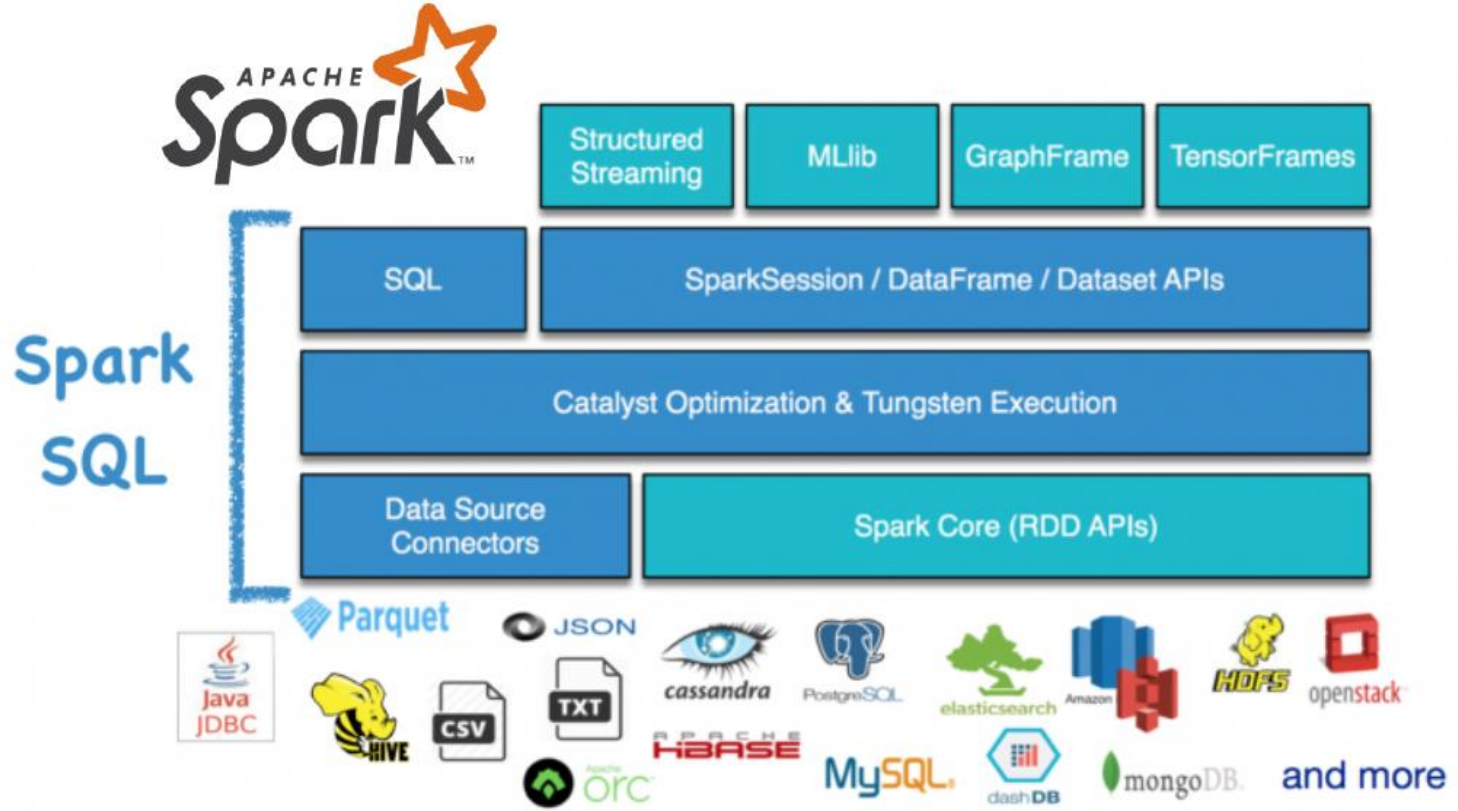
# Why Spark?

- Open-source data processing engine built around speed, ease of use, and sophisticated analytics
- Compute engine designed for distributed data processing at scale
- In-memory engine that is up to 100 times faster than Hadoop
- Largest open-source data project
- Multi-language support – Scala, Java, SQL, R & Python

# Architecture



# Architecture



# Resilient Distributed Datasets

- In-memory objects; All operations in Spark are performed on them
- Collection of entities (like rows)
  - *think of this like a Collection in Java/C#*
- Can be assigned to a variable and methods can be invoked on it
- Methods return values or apply further transformations on RDDs

# RDDs Characteristics

- In-memory
  - Resides in the memory of the cluster
- Partitioned
  - RDD (data) is split into partitions
  - Partitions are split across data nodes (machines) in a cluster
  - Data is processed in parallel on nodes
  - Data is stored in memory for each node in the cluster
- Immutable
  - Once created, cannot be changed
- Resilient
  - Can be reconstructed even if a node crashes
  - Keeps track of every transformation which led to current RDD, called lineage

# Operations on RDD

- Transformation
  - Pipelining (Narrow)
  - Shuffling (Wide)
- Action

# Transformation Operation

- Function that take an RDD as input and produce one or many RDDs as output
- Define chain of Transformations on the dataset, called Lineage Graph
- Example:
  - Load the dataset
  - Convert sales amount from INR to USD
  - Merge first and last names to full name
- These are Lazy operations

# Types of Transformations

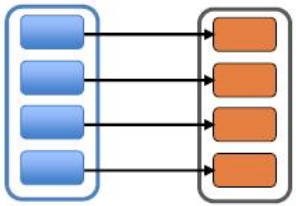
vs

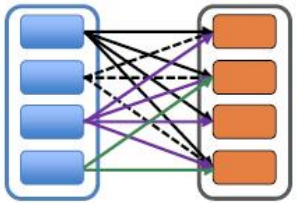
narrow

wide

*each partition of the parent RDD is used by at most one partition of the child RDD*

*multiple child RDD partitions may depend on a single parent RDD partition*

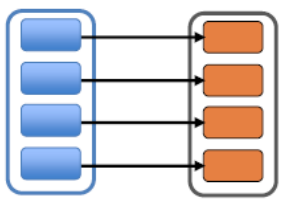




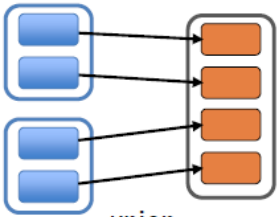
# Types of Transformations

## narrow

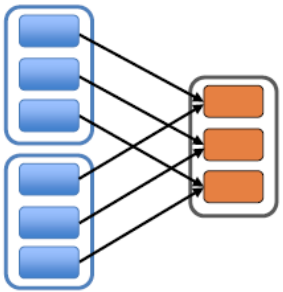
*each partition of the parent RDD is used by at most one partition of the child RDD*



map, filter



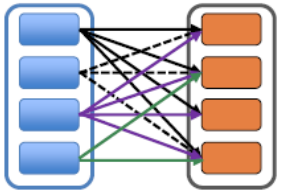
union



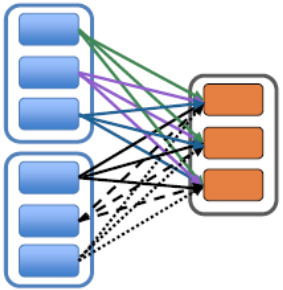
join w/ inputs co-partitioned

## wide

*multiple child RDD partitions may depend on a single parent RDD partition*



groupByKey



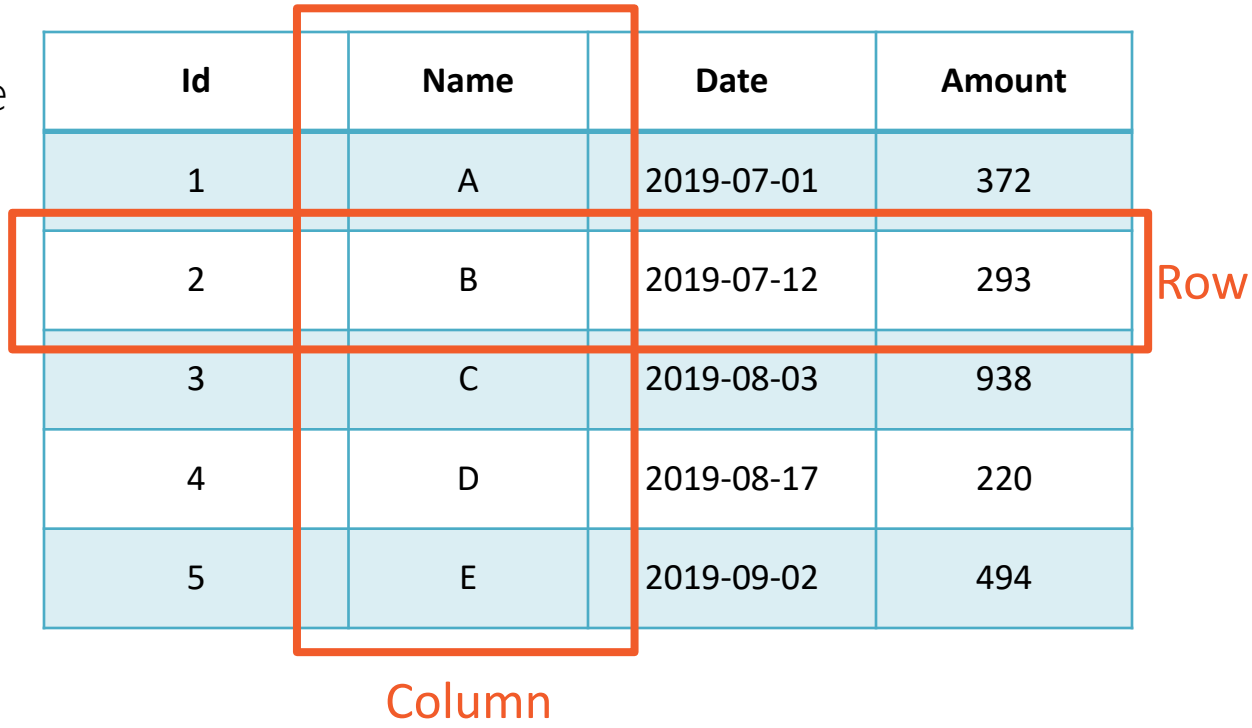
join w/ inputs not co-partitioned

# Action Operation

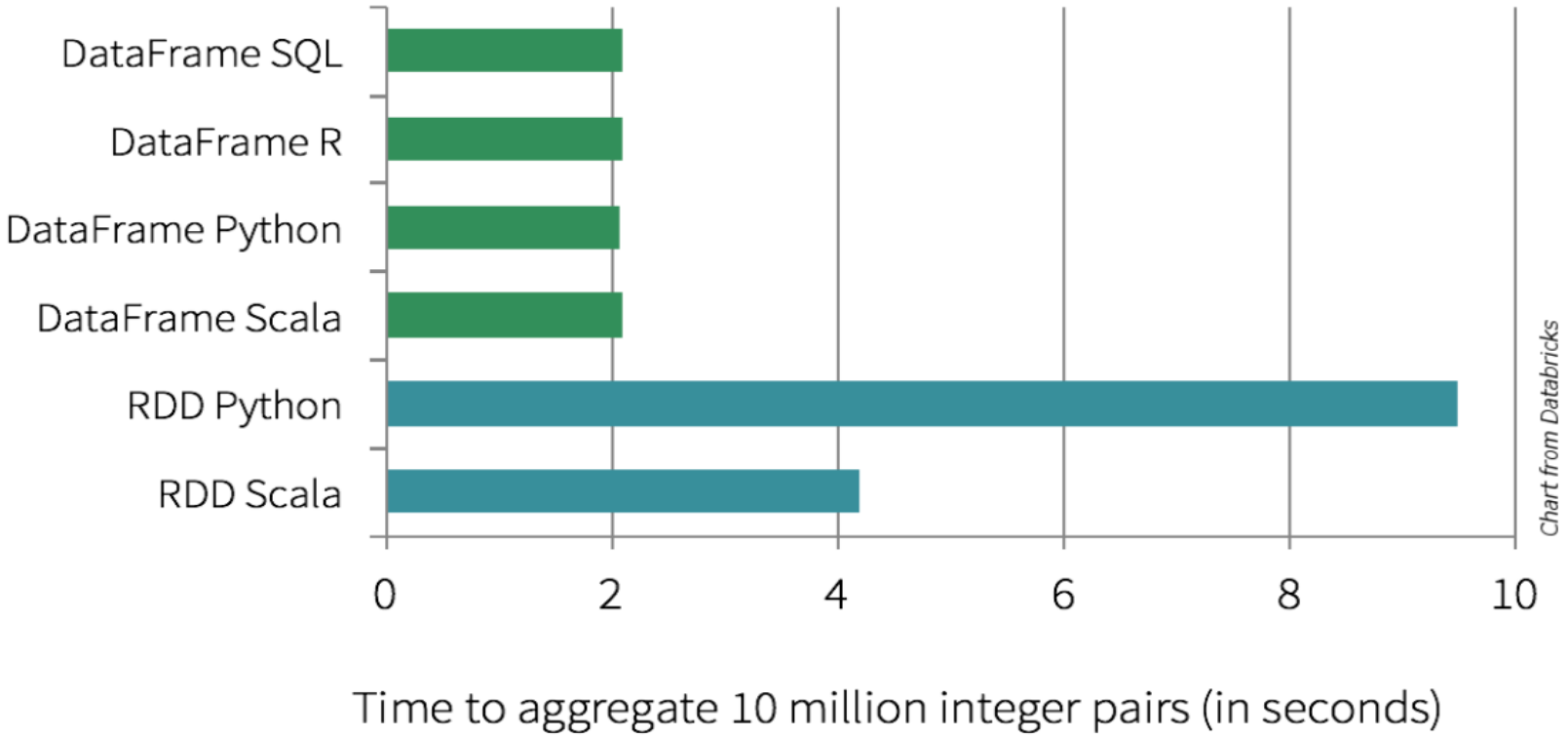
- Returns final result of RDD computations
- Triggers execution using Lineage Graph, after optimizing the transformations applied
- Example
  - Load the data into destination
  - Show the output of the data
  - Display the count

# DataFrames

- Built on top of RDDs
- Named columns
  - Very similar to table in RDBMS
- Impose a structure onto a distributed collection of data
- Optimized for execution
  - Because of Catalyst Optimizer and Tungsten Engine



# Compare Execution Times



# RDD Functions

 = easy  = medium

# Essential Core & Intermediate Spark Operations


### General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

### Math / Statistical

- sample
- randomSplit

### Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

### Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

---


- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

 = easy  = medium

# Essential Core & Intermediate PairRDD Operations


### General

- flatMapValues
- groupByKey
- reduceByKey
- reduceByKeyLocally
- foldByKey
- aggregateByKey
- sortByKey
- combineByKey

### Math / Statistical

- sampleByKey

### Set Theory / Relational

- cogroup (=groupWith)
- join
- subtractByKey
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

### Data Structure

- partitionBy

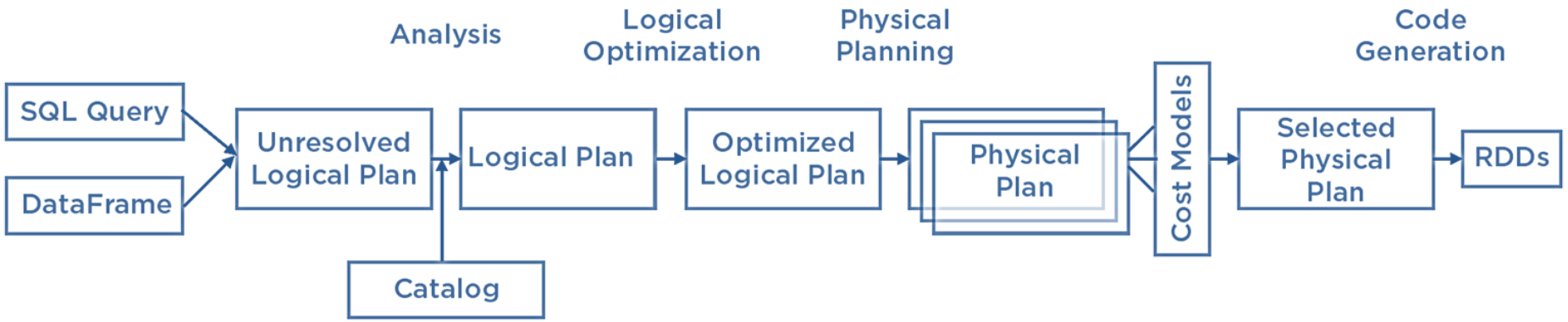
---


- keys
- values

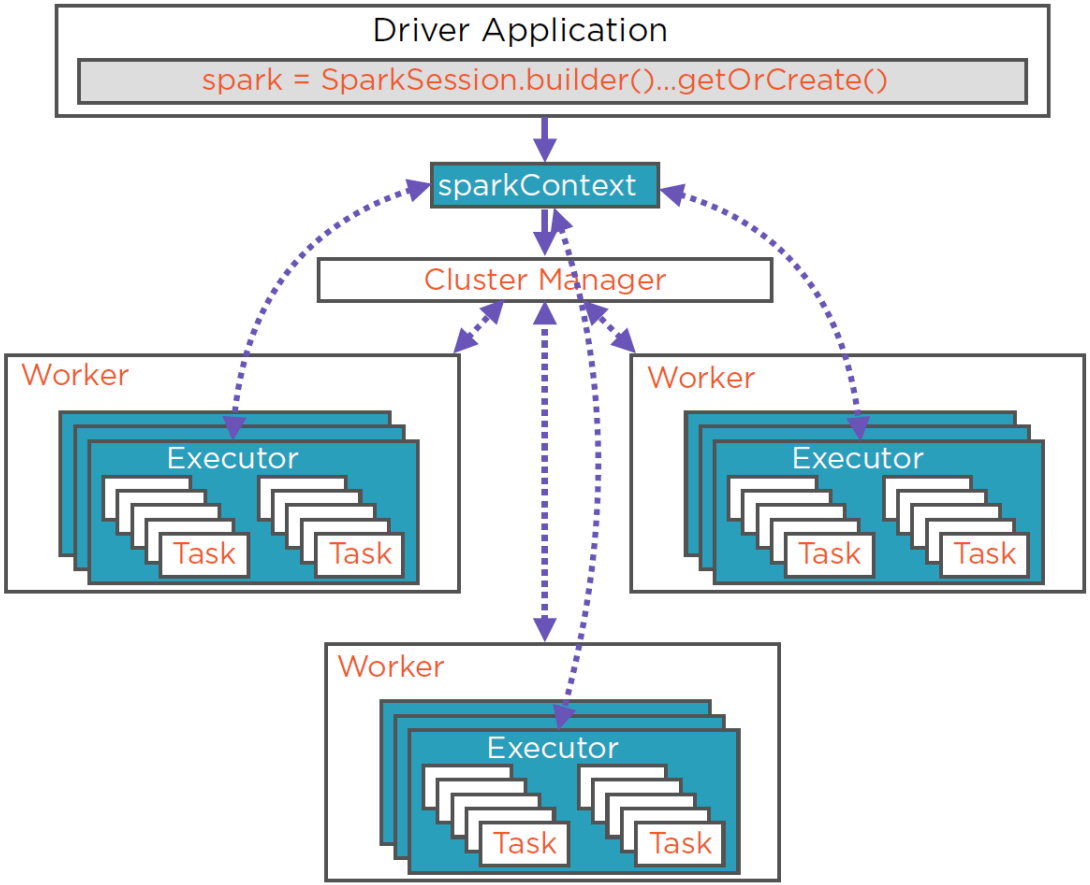
- countByKey
- countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact

# Spark Internals

# Optimization with DataFrames



# Spark Execution



# Definitions

- **Driver Node:** Machine that initiates the Spark Session
- **Driver (Executor):** JVM process that runs on driver node
- **Worker Node:** Machine that is part of cluster that run Spark jobs
- **Executor:** JVM for running tasks on a worker node
- **Cluster Manager:** Responsible for the scheduling and allocation of resources across the host machines
- **Job:** A parallel computation that starts on an Action operation
- **Stage:** A group of tasks inside a job
- **Task:** Unit of work that runs inside an executor, and is part of a stage
- **Spark session:** Unified entry point of a spark application. Provides a way to interact with various spark's functionality with a lesser number of constructs

# Important Points

- Number of jobs = Number of actions performed
- In a job
  - Number of stages = Shuffling operations + 1
- In a stage
  - Number of executors (JVMs) used = Executors specified at the time of creating cluster
  - Number of tasks performed = Number of partitions in RDD
- Number of partitions = blocks or manually specified
- Maximum parallel tasks = Cores in cluster * 2
- Ideal number of cores per executor = 5
- Maximum parallel tasks in an executor = Number of cores in executor

# Why multiple files are created?

- RDDs/DataFrames are broken into partitions, and partitions are distributed to the memory of multiple nodes in a cluster
- Each node can have multiple partitions, and processing happens on partitions – helping to achieve parallelism
- While saving, a folder is created instead of file, and each partition is written as a separate file inside the folder
- Default number of partitions is 200, so 200 part files are created inside the folder

# DataFrame Partitioning

- Specify folder name to read all the partitioned files, using Spark jobs or tools with Spark connector
- Default partitions can be changed using *spark.sql.shuffle.partitions* configuration setting
- Use *coalesce* or *repartition* methods to change partitions on a DataFrame. Changing result in movement of data between partitions
- Rule of thumb – use repartition() to increase partitions, and coalesce() to decrease partitions

# Why ETL loads are faster in Spark?

- Distributed computing platform
- Lazy evaluation of Spark SQL transformations
- Caching/broadcasting small datasets reduce the data shuffle during joins
- Applying the right set of data distribution along with partitioning helps in optimization

# Azure Databricks

# Challenges with Spark

- Infrastructure Management
- Manual Configuration
- Upgrade Challenges
- Tooling & Integration Complexity
- Lack of User Interface
- Difficult to Collaborate on Projects

# Databricks

- Managed and optimized platform for running Apache Spark
- Provides whole bunch of tools out of the box
- Provides an integrated workspace to write the code and do real time collaboration
- Allows you to setup the infrastructure with few clicks and leave the rest for Databricks to manage – scalability, failure recovery, upgrades


Azure Databricks

# What is Azure Databricks?

Apache Spark based Analytics Platform optimized for Azure


- 1st party service on Azure
- Designed and new updates are in collaboration with Databricks team
- One-click setup; streamlined workflows
- Interactive workspace enable collaboration between multiple personas
- Enterprise grade security
- Native integration with Azure services

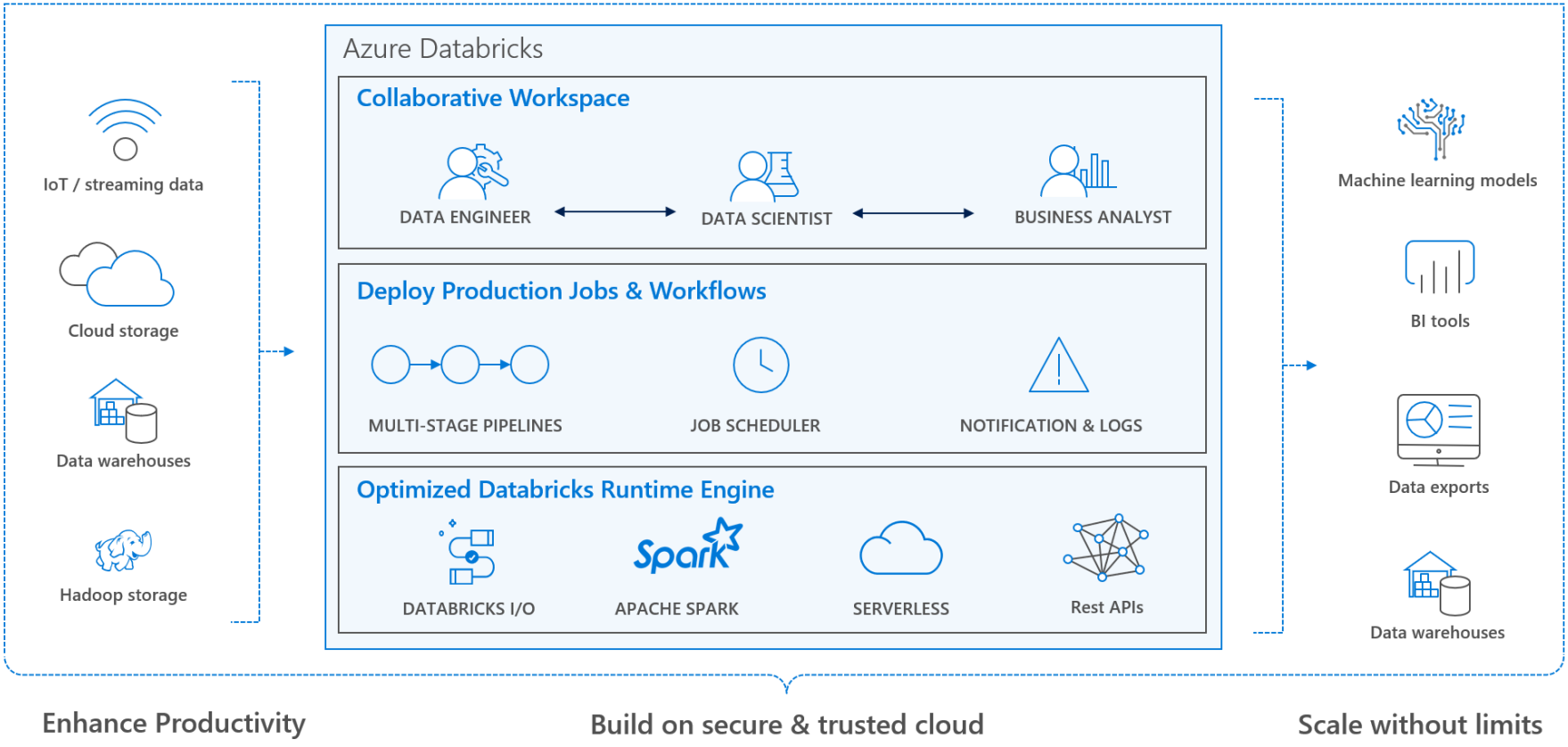
# Supports…

- Building ETL pipelines (batch processing)
- Interactive SQL
- Streaming data (Spark streaming)
- Machine learning
  - Runtime ML contains several libraries, like TensorFlow, PyTorch, Keras etc.
  - Distributed training using Horovod
  - Third party libraries (scikit-learn, DataRobot, H2O Sparkling Water)
- Deep learning
  - Runtime ML contains several libraries, like TensorFlow, PyTorch, Keras, and XGBoost
- Graph processing
  - GraphFrames & GraphX

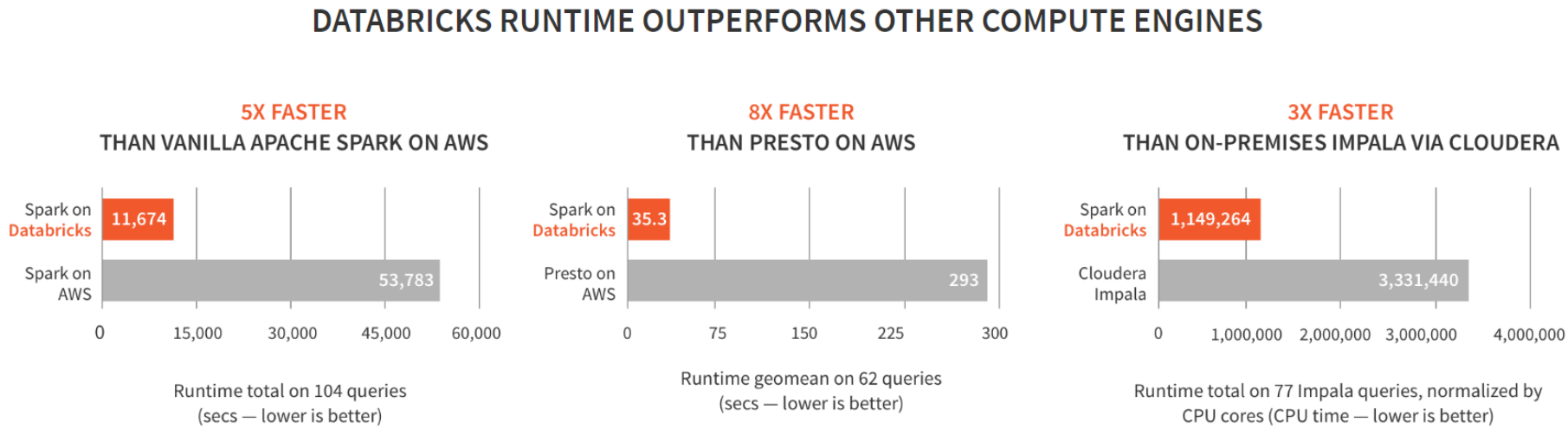
# Advantages of a Unified Platform

- Improves developer productivity
  - A single consistent set of APIs
- All systems in Spark share same abstraction (RDDs)
- Mix & match different processing in same application: ML, streaming etc.
- Performance improves as unnecessary movement of data across engines is eliminated

# Azure Databricks



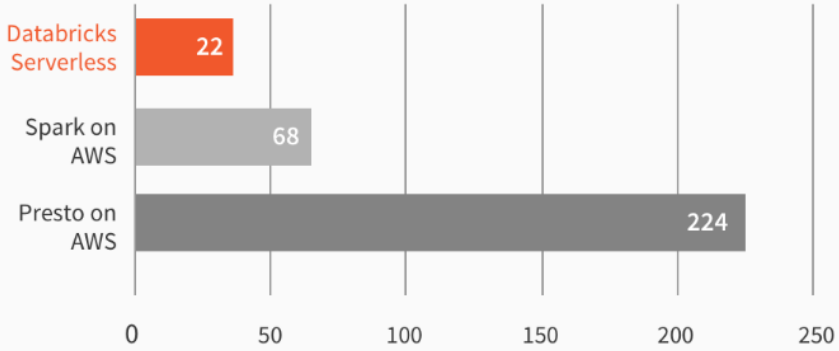
# Databricks Performance



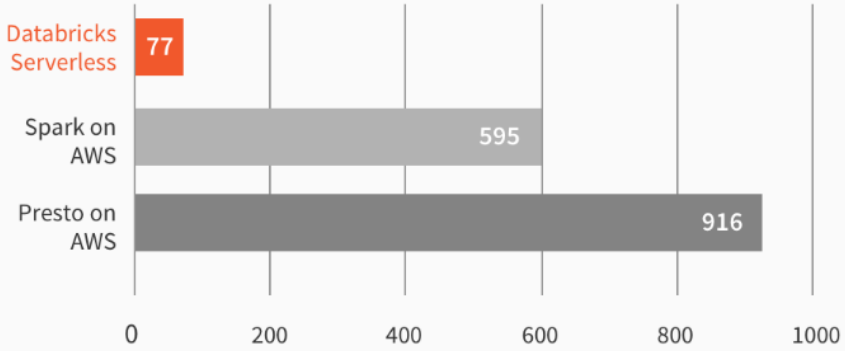
# Databricks Performance

## FASTER PERFORMANCE DURING CONCURRENT AND HETEROGENOUS LOADS

### UP TO 10X FASTER
WITH 5 CONCURRENT USERS




### UP TO 12X FASTER
WITH CONCURRENT 20 USERS + A BACKGROUND ETL JOB



# Databricks Components

# Databricks File System

- No need to provide credentials every time
- Access storage using file system semantics instead of URLs
- Files persist to storage
- Mount Azure Blob Storage using Access Key or SAS
- Mount Azure Data Lake (Gen1 & Gen2) using Service Principal

# Databricks Runtime

- Select version while provisioning cluster
- Comes bundled with:
  - Specific Apache Spark version
  - Set of optimizations
  - Ubuntu and its system libraries
  - Java, Scala, Python, R – with libraries
  - Machine Learning libraries
  - GPU libraries

# Databricks Runtime ML

- Built on top of Databricks Runtime
- Pre-installed libraries
  - TensorFlow, PyTorch, Keras, GraphFrames etc.
- Third-party libraries
  - scikit-learn, XGBoost, DataRobot etc.

# Cluster Types

| Interactive Cluster | Job Cluster |
| --- | --- |
| Interactively analyze the data | Run automated jobs |
| Created by users | Auto created when job starts |
| Manually terminate | Terminates when the job ends |
| Option to auto terminate, if inactive | Option to auto terminate not applicable |
| Low execution time | High throughput |
| Auto scale on demand | Auto scale on demand |
| Comparatively costly | Comparatively cheaper |

# Cluster Modes

| Standard Mode | High Concurrency Mode |
| --- | --- |
| Single user | Multiple users |
| No fault isolation | Fault isolation |
| No task preemption | Task preemption – fair resource sharing |
| Each user require separate cluster | Maximum cluster utilization |
| Supports Scala, Python, SQL, R & Java | Only supports Python, SQL & R |

# Security

- User based access control for jobs, notebooks, clusters etc.
- Single-sign on enabled via Azure Active Directory

# Collaborative Workspace

- Work on same notebooks in real-time
- Track changes with Git integration & revision history
- Visualize data with inbuilt visualizations. Support for 3rd party available

# Jobs & Workflows

- Build workflows in notebooks
- Create jobs based on notebooks and JARs
- Schedule the jobs (have separate clusters for jobs)
- Access audit logs for monitoring & troubleshooting
- Setup alerts for notifications

# Databases & Tables

- Create databases and tables inside them
- Table:
  - Collection of structured data
  - Equivalent to DataFrame – perform same operations on table
  - Created using files lying on storage
  - Directly query or write to tables

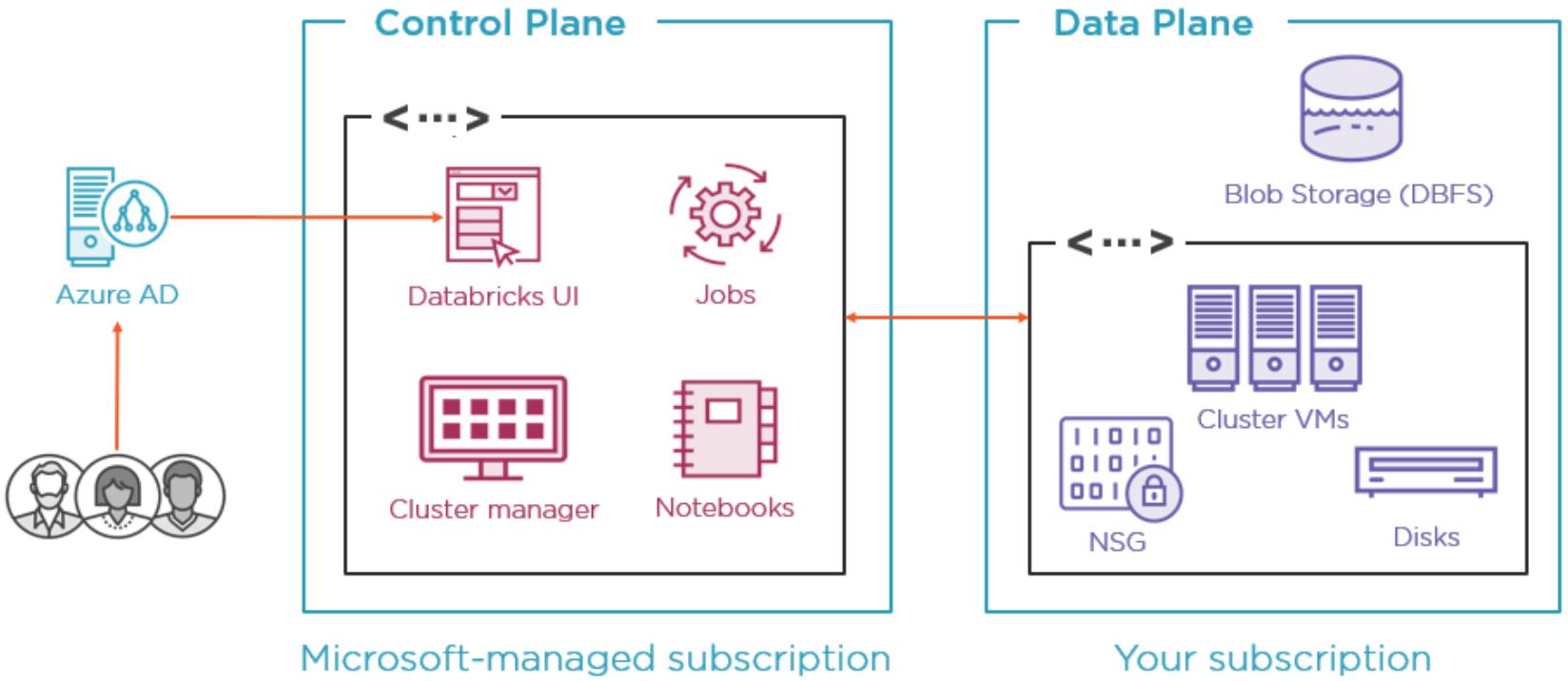
# Table Types

| Managed Table | Unmanaged Table |
| --- | --- |
| Schema and data is managed by Spark | Only schema is managed by Spark |
| Data is stored in DBFS | Data is stored in an external location |
| Stored in Parquet format | Stored in underlying dataset format |
| Dropping table deletes both the schema and the data | Dropping table deletes only the schema, data is not deleted from source |
| Useful to persist staging data | Useful to persist processed data |

# DBIO

- Databricks I/O (DBIO)
  - Layer on top of Azure Storage and is highly optimized access layer
  - Higher throughput from underlying storage
  - Transactional Writes
    - Features both appends & new writes

# Azure Databricks Architecture

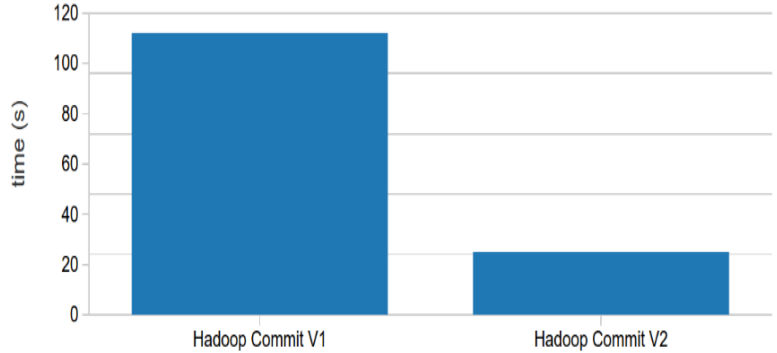


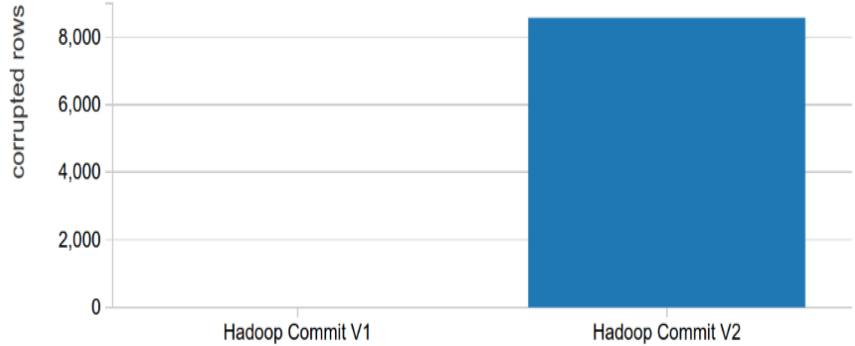
# High Speed Azure Connectors

- Azure SQL
- Azure SQL Data Warehouse
- Cosmos DB
- Azure Storage
- Azure Data Lake Store
- Power BI
- Application Insights etc.

# Transactional Writes Challenges

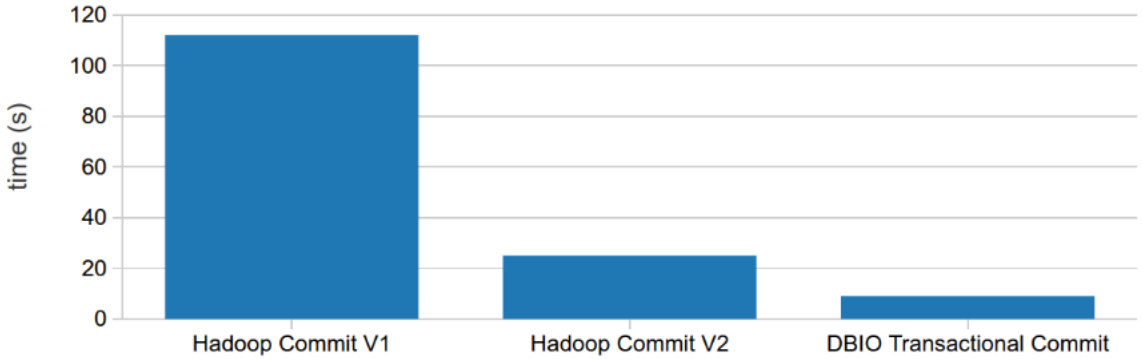
- Commit protocol determines how results should be written at the end of the job
- Only output data of successful jobs data should be visible to readers
- Partial commits/duplicates can become challenging
- Trade-off between performance & transactionality



# Successful Transactional Writes with DBIO

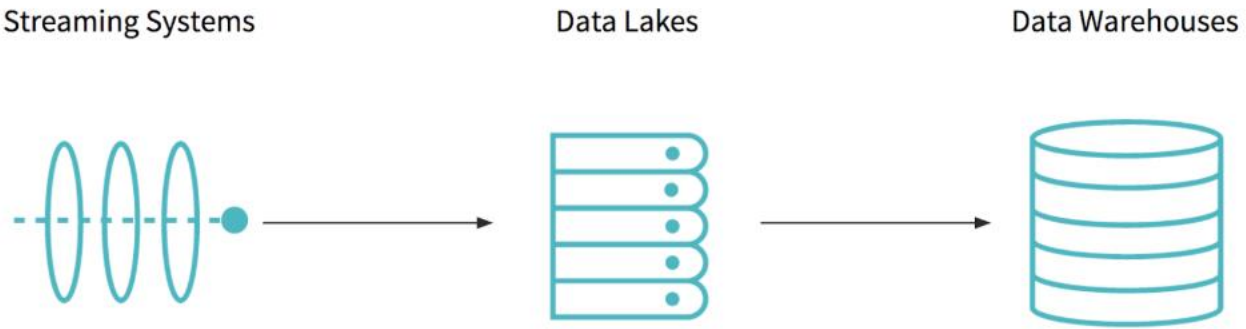
- New transactional commit protocol – DBIO Transactional Commit
- Process
  - Tag files written with the unique transaction id
  - Write files directly to their final location
  - Mark the transaction as committed when the jobs commits



# Databricks Delta

# What is Databricks Delta?

- New generation engine in Databricks built on top of Apache Spark
- Focused towards combining
  - the scale of a data lake
  - the reliability, performance & concurrency of a data warehouse
  - and the low latency of streaming in a single system



# Databricks Delta Features

- ACID transactions
  - Serializable isolation levels ensure that readers never see inconsistent data
- UPSERTS
  - Efficient handling of updates and inserts
- High throughput streaming ingestion
  - Ingest high volume data directly into query tables
- Transparent Caching
  - Accelerate read speeds through auto data caching to a node's local storage
  - Works for all parquet files and not just delta format files

# Databricks Delta Features

- Data skipping
  - Leverage statistics on data files to prune files *(partition pruning)* more effectively
- Z-Ordering
  - Co-locate related data in same set of files. Makes Data Skipping more effective
- Compaction
  - Improve the speed of read queries from a table by coalescing small files into larger ones
- Garbage Collection
  - To ensure that concurrent readers can continue reading a stale snapshot of a table, Delta leaves deleted files on DBFS for a period of time. Clean-up the files using Garbage Collection

# Databricks Delta Features

- Time Travel
  - Access historical version of data
  - Useful for auditing data changes, reproduce experiments and reports, and rollbacks

# Azure HDInsight

# What is Azure HDInsight?

- Managed, full-spectrum, open-source analytics service for enterprises
- Fully managed clusters
- Supports a broad range of scenarios


Apache Hadoop

Apache Spark

Apache Kafka

Apache HBase

Interactive Query

Apache Storm


ML Services

# Important Points

- Billing
  - Starts when cluster is created and stops when it is deleted
  - Pro-rated per minute
- Clusters types
  - Can be created only for a single workload or technology
  - Multiple type clusters are not supported
  - To do that, use VNet to connect the clusters
- Integration with Active Directory and Apache Ranger