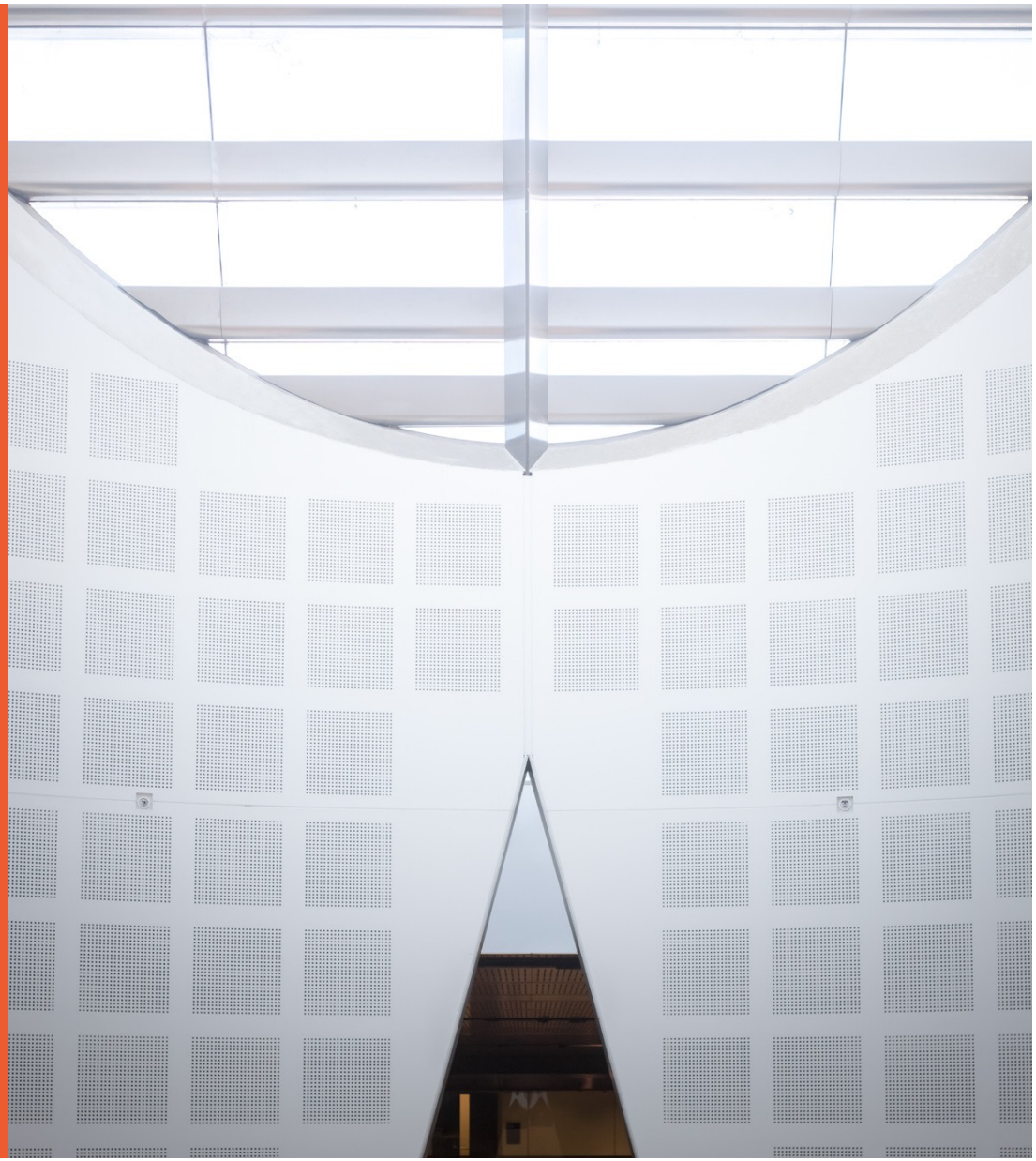
COMP5339: Data Engineering

Week 10: Scalable Data Engineering

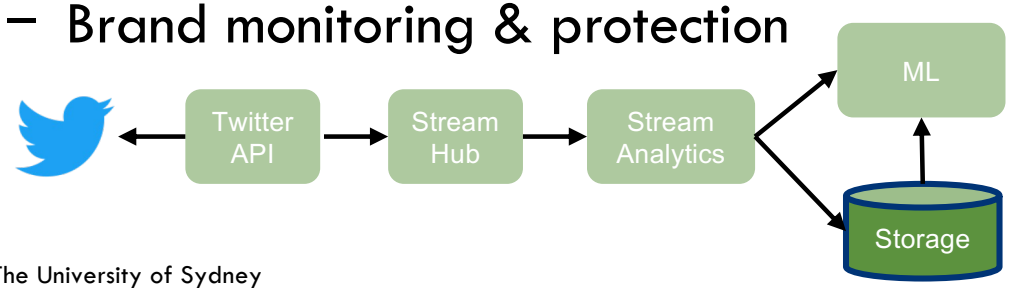
Presented by
Uwe Roehm
School of Computer Science

THE UNIVERSITY OF SYDNEY



Social Media Real-Time Analysis

- Social Media Sentiment Analysis in real-time
  - e.g. via Twitter/X monitoring
  - Sentiment analysis of tweets
    - The process of computationally identifying and categorizing opinions expressed in a piece of text, especially in order to determine whether the writer's attitude towards a particular topic, product, etc. is **positive**, **negative**, or **neutral**.
- Digital marketing
- Brand monitoring & protection



Source: Microsoft

Important Aspects

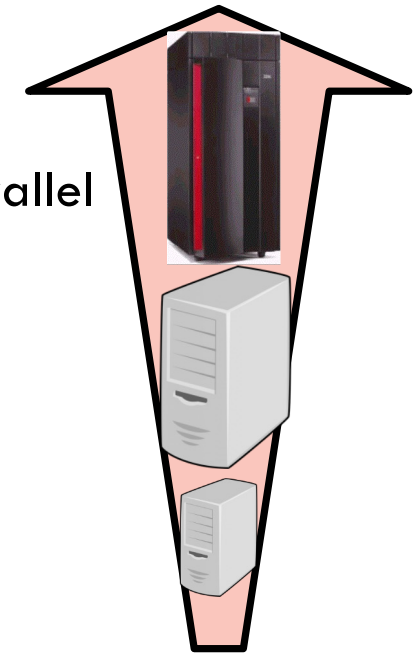
- How to design a scalable data pipeline able to process huge amounts of data and that allows for continuous growth?

- How to scale the data management layer accordingly?

Scale-Up vs. Scale-Out

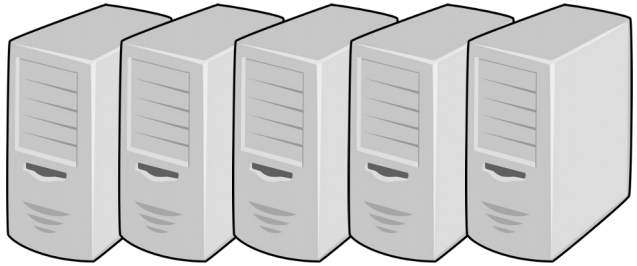
Scale-Up

- To scale with increasing load, buy more powerful, larger hardware
  - from single server to large massive-parallel server



Scale-Out

- A single server has limits…
  For Big Data processing, need to **scale-out** to a cluster of multiple servers (nodes):



[Source: Server.png from PinClipart.com]

State-of-the-Art: **shared-nothing architecture** of multiple storage or compute nodes

Scale-out… …Data Center Style

http://cdn.techsnapr.com/wp-content/uploads/2011/04/FB_SERVER_040_x900.jpg

Shared Nothing

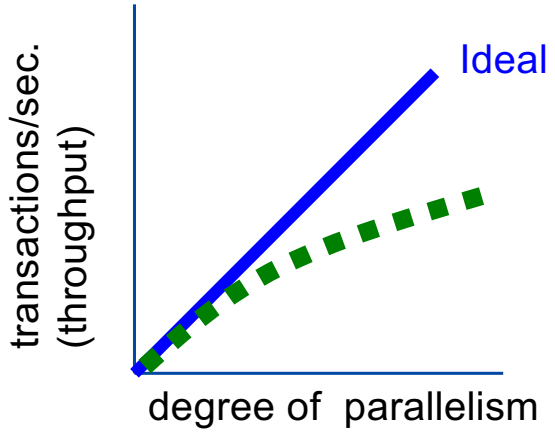
每个节点都有自己独立的 CPU、内存和磁盘，节点之间不共享任何硬件资源，只通过网络通信。

- Node consists of a processor, memory, and one or more disks. Processors at one node communicate with another processor at another node using an interconnection network. A node functions as the server for the data on the disk(s) the node owns.
  - Examples: Teradata, Tandem, Oracle-n CUBE
- Data accessed from local disks (and local memory) do not pass through interconnection network, thereby minimizing the interference of resource sharing.
- Shared-nothing multiprocessors can be scaled up to thousands of processors without interference.
- Main drawback: cost of communication and non-local disk access; sending data involves software interaction at both ends.

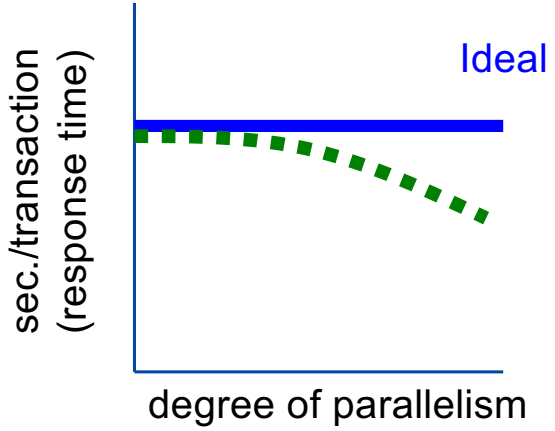
The Goals of Scalability

现实中两者都会受到 overhead（额外开销） 影

- Speed-Up 数据量不变，资源增加，运行时间应按比例减少
  - More resources means proportionally less time for a given amount of data.
    - but: coordination overhead



- Scale-Up 数据量和资源同时按比例增加，运行时间应保持不变
  - If resources increased in proportion to increase in data size, response time should remain constant
    - again: modulo overhead



Source: Ramakrishnan/Gehrke Book

Design Principles for Scalable Systems

- Scalability best achieved via **loosely-coupled systems**
- **Separation of compute and storage nodes**
  - Allows to grow storage independent of compute / data analysis needs – and vice versa

- Main drawback: **Availability** considerations
  - The more nodes, the higher the probability of some failure
- Further drawback: cost of communication and non-local disk access
  - latency can increase (how long a request takes to complete)

Goals

- **Scale-Agnostic Data Management**
  - *data sharding* for performance
  - *replication* for availability
  - ideally such that applications are unaware of underlying complexities
- **Scale-Agnostic Data Processing**
  - need to parallelize across hundreds/thousands of CPUs, can grow
    - if it fits on your machines, multiply by 10, if that fits, multiply by 1000…
  - **Performance**: parallel query processing
  - **Availability**: Ideally, the system is never down; can handle failures transparent
  - **Elasticity**: Change scale as needed even as system is in operation …

  => Map/Reduce processing paradigm

Scale-Agnostic Data Management

THE UNIVERSITY OF SYDNEY

Storage Optimisations

- Single Machine
  - Indexing
    - Index structure of key attributes to support filtering and joins
  - Data Partitioning
    - Partition larger table / dataset across multiple disks
  - Column vs Row Store
  - RAID (Redundant Array of Independent Disks)

- Cluster of Machines
  - Data Sharding, Data Replication, Caching (e.g. memcached)

Column Store

- **Row-oriented databases** organize data by **record** (row).
  - The traditional way of organizing data and still provide some key benefits for storing data quickly.
  - Optimized for reading and writing rows efficiently.
  - Postgres and MySQL are common row-oriented databases

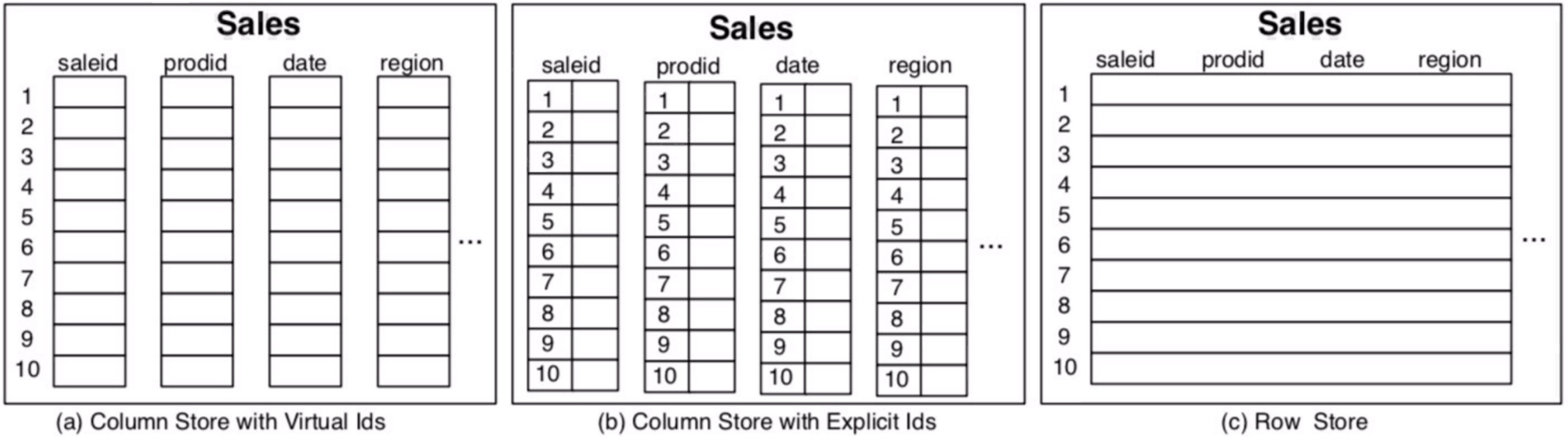
```
Row 1: [1, John, 30, 5000]
Row 2: [2, Alice,28, 6000]
Row 3: [3, Bob,  35, 5500]
```

- **Column-oriented databases** organize data by **field** (column).
  - Columnar databases are popular and provide performance advantages to querying data.
  - Optimized for reading and computing on columns efficiently.
  - Google BigQuery is a commonly used column-oriented database.

```
Column 1: [1, 2, 3]
Column 2: [John, Alice, Bob]
Column 3: [20, 28, 35]
Column 4: [5000, 6000, 5500]
```

Column Store

- Idea: store data organised by attributes rather than by row



(a) Column Store with Virtual Ids (b) Column Store with Explicit Ids (c) Row Store

- Various optimisation and design choices, such as:
  - Pure column organisation: homogeneous columns per page
  - Mixed column organisation: one page can contain values from different columns

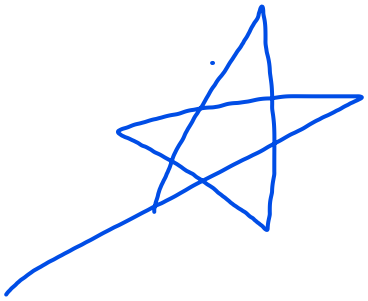
Source: Abadi, Boncz, Harizopoulos, Idreos, Madden:
"The Design and Implementation of Modern Column-Oriented Database Systems", 2012.

Apache Parquet

Source: https://parquet.apache.org/docs/

- open source, column-oriented data file format
- files are organised by column rather than row
- Supports complex data types & nested data structures
- Supports efficient data compression and encoding schemes

Column Store – Pros and Cons



- Advantages
  - Non-needed attributes do not have to be read
    - less I/O
    - more efficient scan for OLAP-style aggregation queries
  - Better compression possibility
    - same data types, more uniform data

    => even less I/O
  - Better parallel processing
    - different columns can be stored at different disks, allowing for parallel I/O
    - possibility to use SIMD instructions rather than tuple-at-a-time
- Cons
  - updates
  - full-row access
  - Not suited for small-table data

Distributed Data Management

Two main physical design techniques:

- **Data Partitioning / Data Sharding**
  - Storing subsets of the original data set at different places
    - can be in different tables in schema on same server, or at remote sites
  - Goal is to query smaller data sets & to gain scalability by parallelism
  - Subsets can be defined by
    - columns: ***Vertical Partitioning***
    - rows: ***Horizontal Partitioning***
      (if each partition is stored on a different site, also called ***Sharding***)
- **Data Replication**
  - Storing copies ('replicas') of the same data at more than one place
  - Goal is fail safety / availability
  - Several variants, basically distinguishing between which replica(s) we are allowed to read or update

Data Partitioning

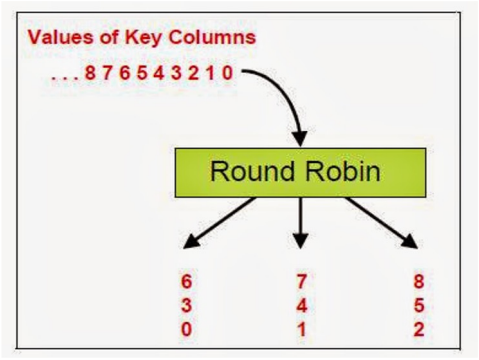
- **Horizontal Partitioning:** set of rows (known as a *shard*)
- **Vertical Partitioning:** set of columns

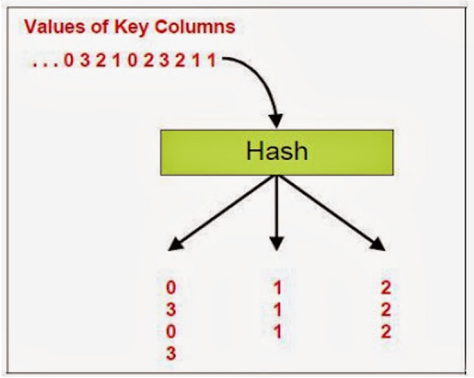
Stations

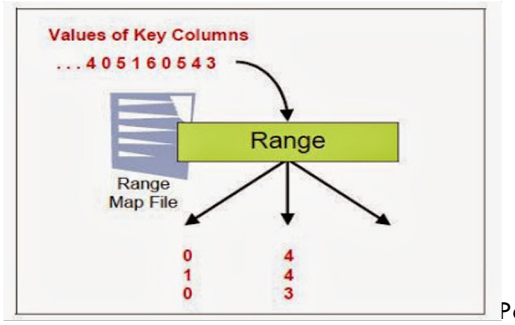
stationid	name	commence	state
409001	Murray River at Albury	14.04.1892	NSW
409002	Murray River at Corowa	01.04.1894	NSW
409003	Murray River at Denuquin	01.09.1899	NSW
409004	Murray River at Barham	01.01.1905	NSW
409202	Murray River at Tocumwal	06.06.1886	VIC
409204	Murray River at Swan Hill	01.01.1909	VIC
219018	Murray River at Quaama	07.12.1966	VIC

How to place data into partitions?

- **Round-robin**
  - Placement of partitions is going through all nodes in rounds
- **Hash partitioning**
  - Target node is determined by a hash function on the tuple id or key
- **Range partitioning**
  - Each node stores a partition defined by a range predicate





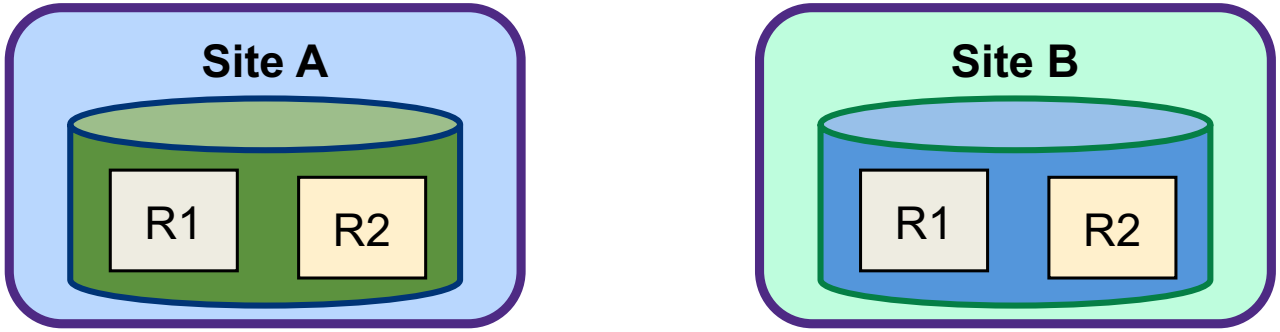


Advantages of Partitioning

- Easier to manage than a large table
- Better **availability**
  - If one partition is down, others are unaffected **if** stored on different disks.
- Helps with bulk loading, e.g., for data warehouse applications
- Queries faster on smaller partitions; can be evaluated in parallel
  - Separate nodes do not need to scan whole dataset, but only local data partition
  - Inter-Query Parallelism (different Q independently on separate places)
  - Intra-Query Parallelism (same Q accesses several places in parallel)

Data Replication

- Data Replication – data items are stored in more than one place
  - Typically, in different databases at different physical locations
  - Gives increased **availability**
    - Important as in distributed systems, chance of *some* failures increases!
    - General idea: **Read Any Copy** => if one site is down, another replica can still be read
  - Side-effect: Load balancing for faster query (read-only!) evaluation
  - When data doesn't change, all is well – updates become slower though…
    - Core question: Do we have to write all the copies? 下一页2种方法



When to Propagate Updates to Replicas?

Synchronous ('Eager') Replication

- Synchronous Replication – update all replicas inside original transaction
- Good for consistency
- Bad for performance

Asynchronous ('Lazy') Replication

- Asynchronous Replication
  - Update one copy
  - Apply those writes that are relevant to replicas in a separate "copier" txn
  - Original txn may be entirely local
- Good for performance
- May be bad for consistency

Primary-Copy vs. Multi-Leader Replication

Primary Copy (Single Leader)

- One replica of each item is authoritative, which is always updated first
- Bad for flexibility

Multi-Leader

- "update anywhere"
- Different transactions can update replicas in different orders
- Good for flexibility
- If eager propagation, then deadlock is very common;
- If lazy propagation, then need conflict resolution to ensure convergence

Primary Copy Asynchronous Replication

- Exactly one copy of relation is designated primary / leader. Replicas cannot be directly updated.
  - All writers have to update the primary-copy
  - Readers can read *any* replica.
  - The primary copy is published.
  - Other sites subscribe to (fragments of) this relation; these are secondary copies.
  - Two Steps.
    1. capture changes made by committed transactions
    2. apply changes

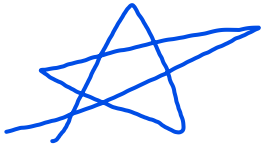
Multi-Leader Asynchronous Replication

- More than one of the copies of an object can be a leader
  - Updaters can modify any of these leaders
  - Readers again can read any replica
- Changes must be propagated to other copies
- If two leader copies are changed in a conflicting manner, this must be resolved.
- Best used when conflicts do not arise:
  - E.g., Each leader site is responsible for a disjoint fragment.
  - E.g., Updating "permissions" owned by one leader at a time.

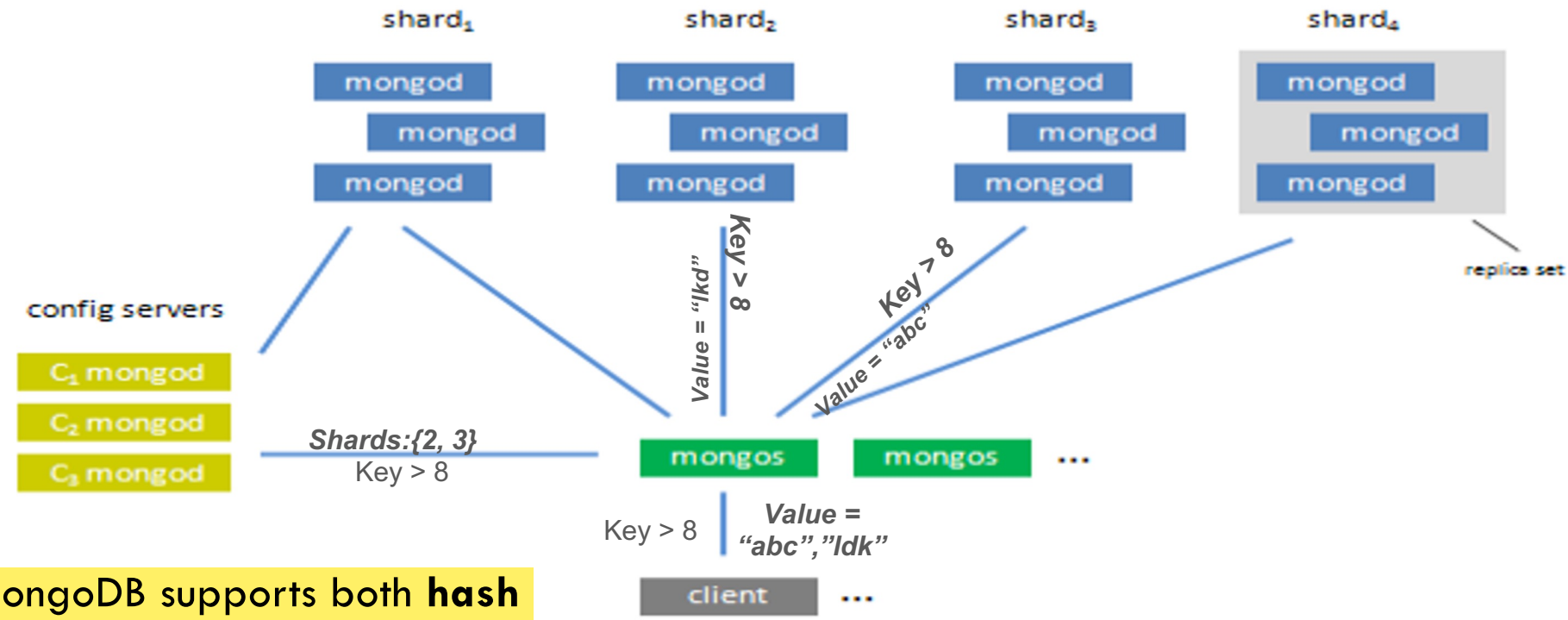
Replication Design Space Summary

- In practice, want performance and simple system design
  - lazy propagation and single-leader
- In theory, want consistency and application generality
  - eager propagation, multi-leader ('update anywhere')

	Primary Copy	Update Anywhere
Lazy Propagation (Asynchronous)	most practice approaches	
Eager Propagation (Synchronous)		ideal world



Example: MongoDB Architecture

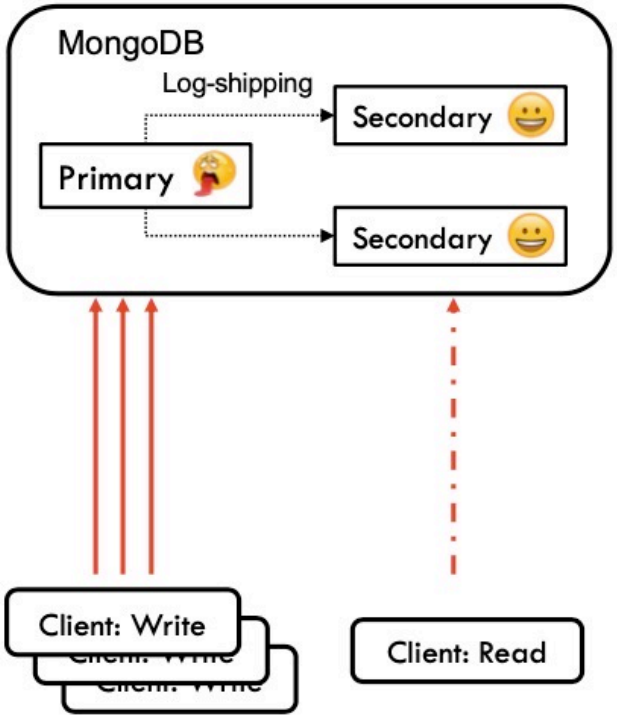


MongoDB supports both **hash** and **range** partitioning

[Source: http://www.mongodb.org/display/DOCS/Sharding+Introduction]

MongoDB Example (cont'd)

- MongoDB internally uses single-leader asynchronous replication with log shipping
  - All writes go to Primary.
  - The data on the secondary nodes can be somewhat stale.
- MongoDB client can adjust a parameter called *Read Preference*, so reads can be directed either to the primary node or to secondary nodes.



Goals of Distributed Data Management

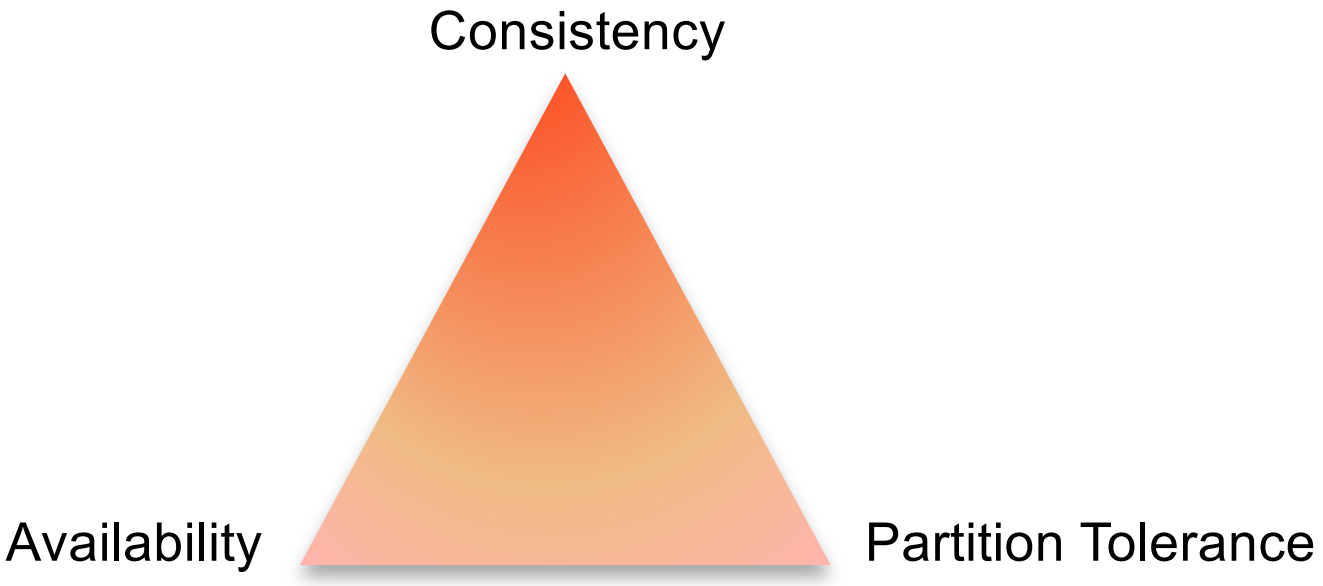
- Strong Consistency
  - Every read receives the most recent write.
  - Ideally, all copies have the same data as if there is only one copy
- High Availability
  - A data system should always be up.
  - The challenge: larger the system, higher the probability of failure
- Partition Tolerance
  - If there is network failure that splits the processing into two subsets that cannot talk to each other, then the goal is to keep going to allow processing to continue in these groups.
  - Ensures the system functions despite communication breakdowns.

Consistency

- Suppose you log in to a social media platform that has replicated servers in *New York* and *London*.
- You update your profile picture (write operation) while connected to New York. The **New York** replica accepts your update but fails to sync immediately with the London replica due to lazy propagation.
- You ***refresh*** your profile page, but due to load balancing, your read request goes to the **London** replica, which has not yet received your update.
- You see your old profile picture instead of the new one you just uploaded.
- This is known as a "Read Your Own Write" consistency violation.
  - Even though you just updated your profile, the system fails to show your own update because you are reading from a replica that hasn't caught up.

CAP Theorem

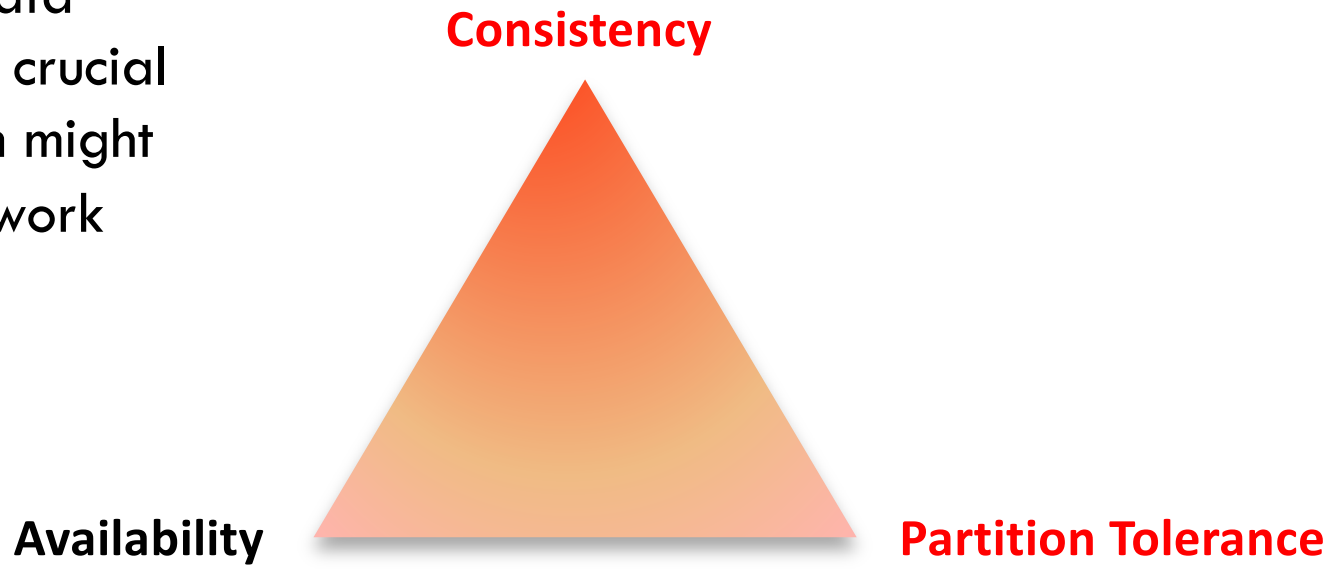
- You can have at most two of the properties for any network shared data system.



Consistency + Partition Tolerance (CP)

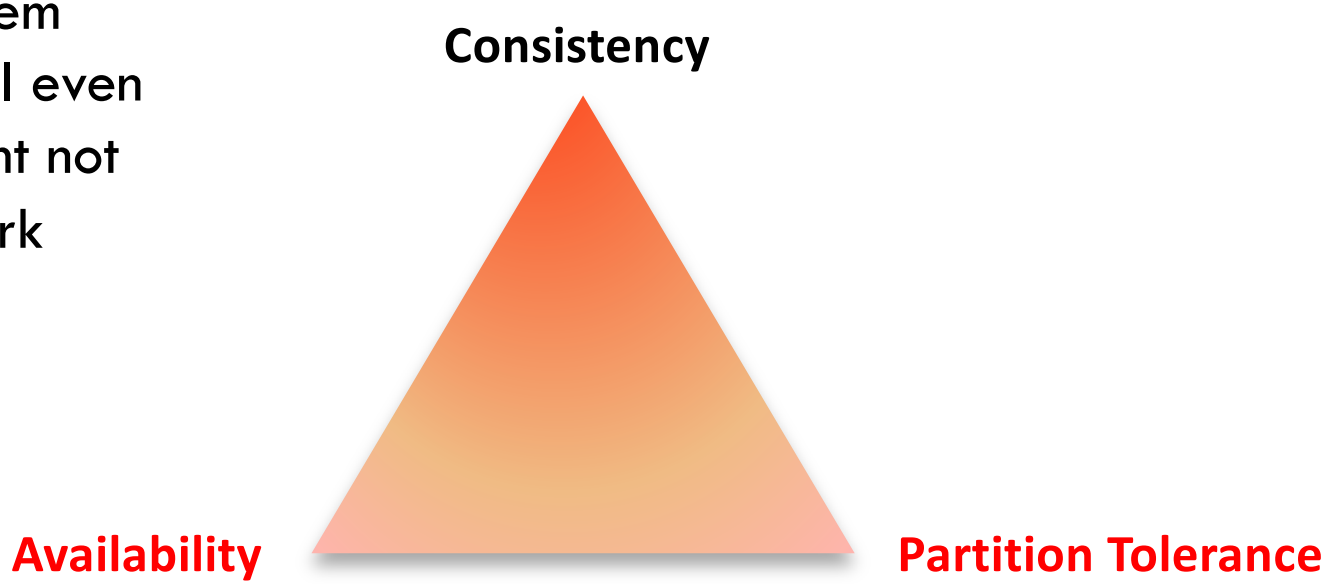
知道什么场景选择什么

Scenario: In a financial application, maintaining data consistency across nodes is crucial even if it means the system might be unavailable during network issues.



Availability + Partition Tolerance (AP)

Scenario: In a social media platform, ensuring the system remains available is crucial even if it means some data might not be consistent during network partitions.



Summary

- Scale-agnostic data management is crucial for supporting scalable data pipeline
  - Data Partitioning / Data Sharding
  - Data Replication
  - In-memory Data Caching

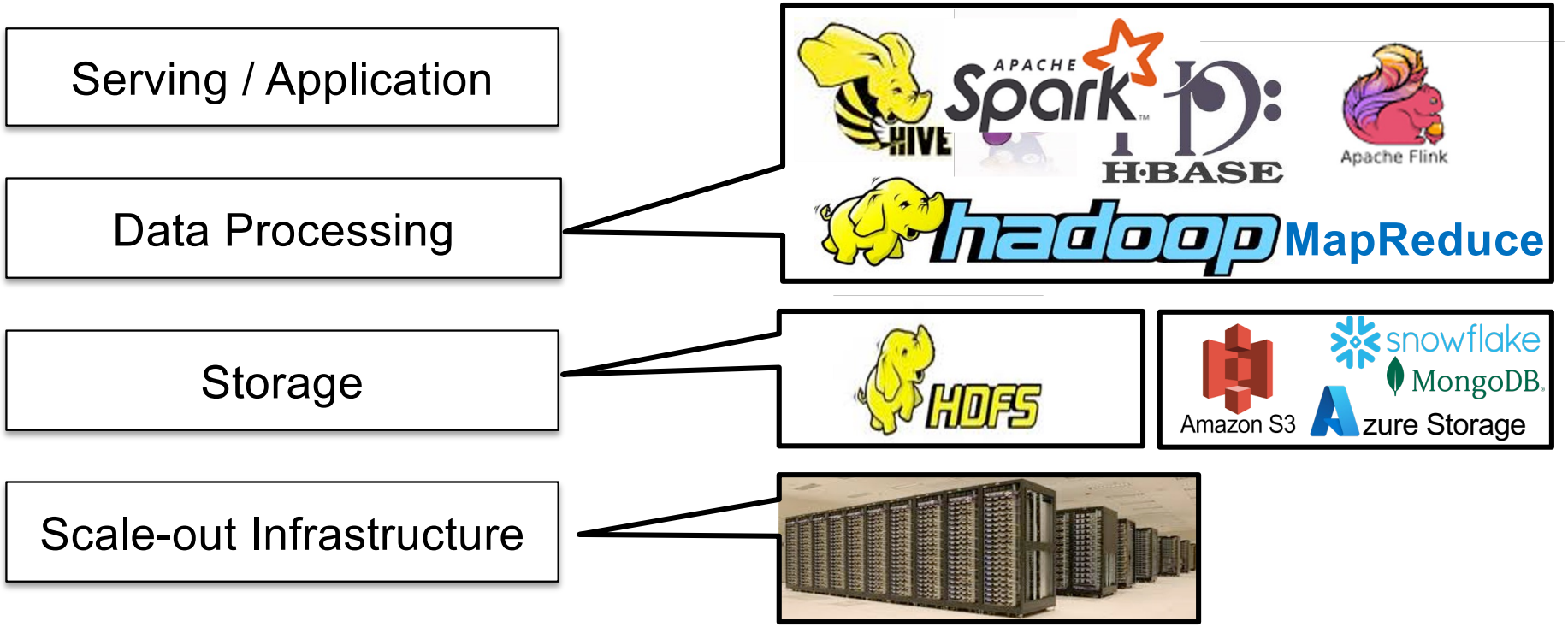
- Optimisations of data organization per node
  - Indexing and Clustering
  - Column Store vs Row Store

Scale-Agnostic Data Processing

Big Data Analytics Stack

记住四层结构

- Layered stack of frameworks for distributed data management and processing
- Many choices of distributed data processing platforms



Big Data Analytics Stack

- Stable storage
  - Distributed file/blob storage such as HDFS or S3
  - Flexible Structure Storage ('NoSQL')
  - Cloud Data Lake / Data Warehouse services

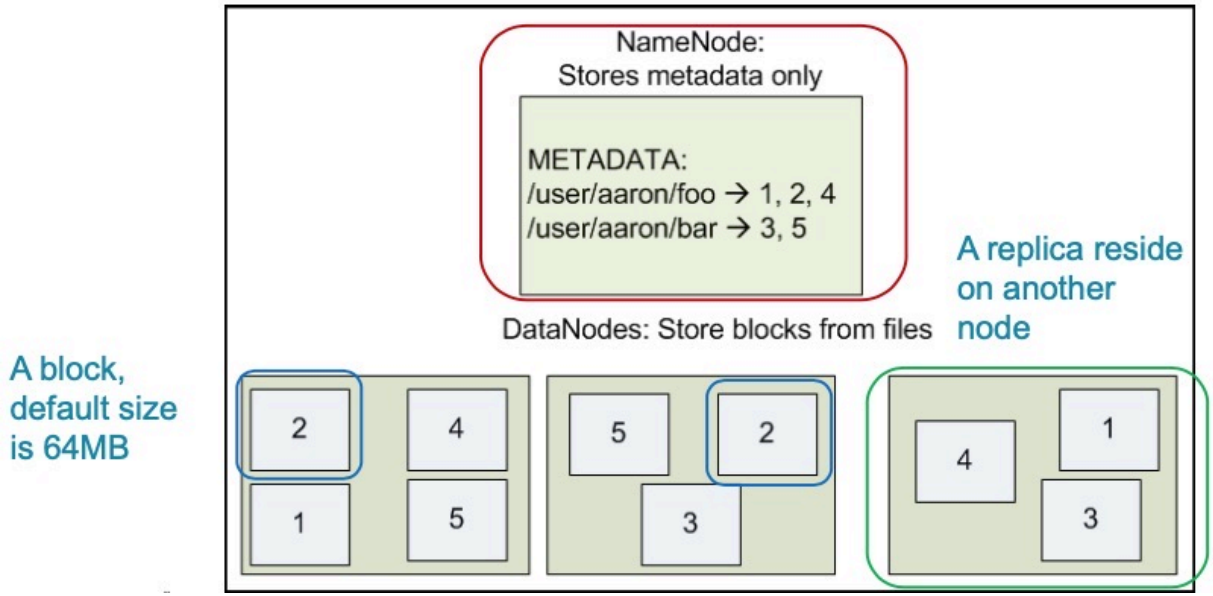
- Higher-level data programming framework
  - MapReduce/Hadoop
    - "A new abstraction … to express computation … but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library"

      Jeffrey Dean and Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*. In OSDI'04.
  - Apache Spark, Apache Flink

HDFS – Hadoop Distributed File System

- A distributed file systems allow clients to access files on remote servers "transparently".
  - It is the open-source implementation of the original Google File System (GFS)
- Core design ideas
  - One **NameNode:** Name space, access control, control of data location and data replication
  - Multiple **DataNodes**: actual data storage
- HDFS files
  - **Partitioned** into large blocks (64 MB by default)
  - Each block is **replicated** across multiple nodes (default: 3)

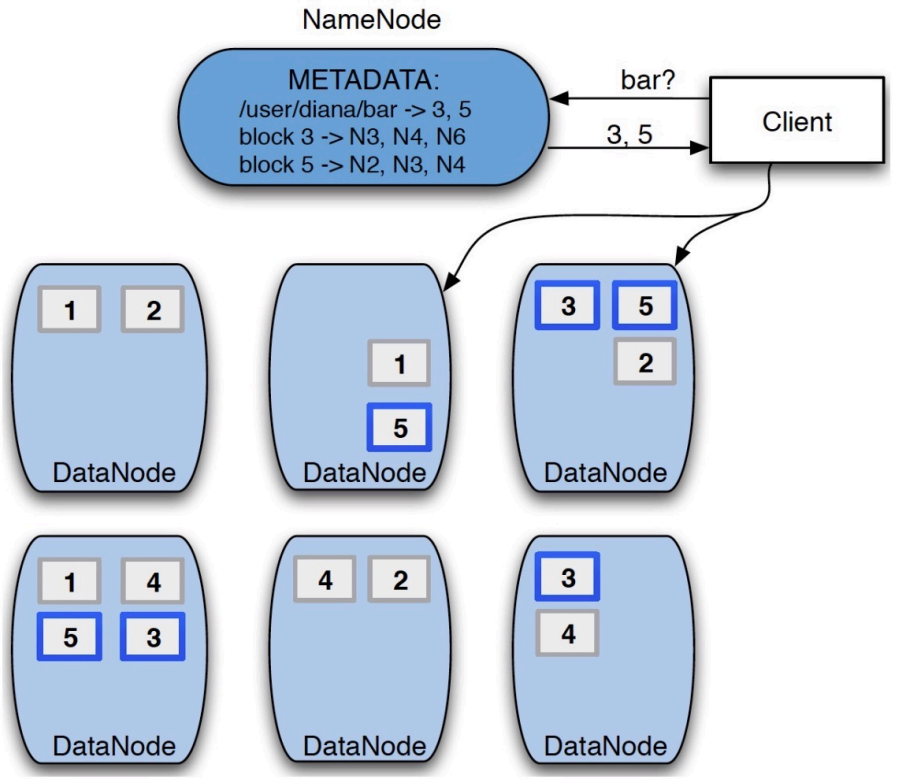
HDFS Architecture



HDFS Read

记住步骤

1. Client sends request to NameNode
2. NameNode replies with block handle and location of replicas.
3. Client caches this information.
4. Client sends request to a nearby replica, specifying block handle.
5. The DataNode sends data blocks to the client



搞清楚每个是干嘛的

Distributed Data Analytics Frameworks

- **Apache Hadoop**
  - Open-source implementation of original MapReduce from Google; Apache top-level project
  - Java framework, but also provides a Python interface nowadays
  - Parts: own distributed file system (HDFS), job scheduler (YARN), MR framework (Hadoop)
- **Apache Spark**
  - Distributed cluster computing framework on top of HDFS/YARN
  - Concentrates on **main-memory** processing and more **high-level data flow control**
- **Apache Flink**
  - Efficient data flow runtime on top of HDFS/YARN
  - Similar to Spark, but more emphasize on **build-in dataflow optimiser** and **pipelined processing**
  - Strong for data stream processing

Distributed Data Analytics Frameworks (continued)

- **Apache Hive**
  - Provides an SQL-like interface on top of Hadoop / HDFS
  - Allows to define a relational schema on top of HDFS files, and to query and analyse data with HiveQL (SQL dialect)
  - Queries automatically translated to MR jobs and executed in parallel in cluster
  - Example: WordCount in HIVE

```
CREATE TABLE docs (line STRING);
LOAD DATA INPATH 'input_file' OVERWRITE INTO TABLE docs;

CREATE TABLE word_counts AS
      SELECT word, count(1) AS count
        FROM (SELECT explode(split(line, '\s')) AS word FROM docs) temp
       GROUP BY word
       ORDER BY word;
```

- **Many more high-level frameworks for advanced data analytics.**

Map Reduce

- MapReduce is a programming model for processing data sets with a parallel, distributed algorithm on a cluster of nodes

- MapReduce program is composed of
  - a **map** procedure, which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name)
  - a **reduce** method, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies).

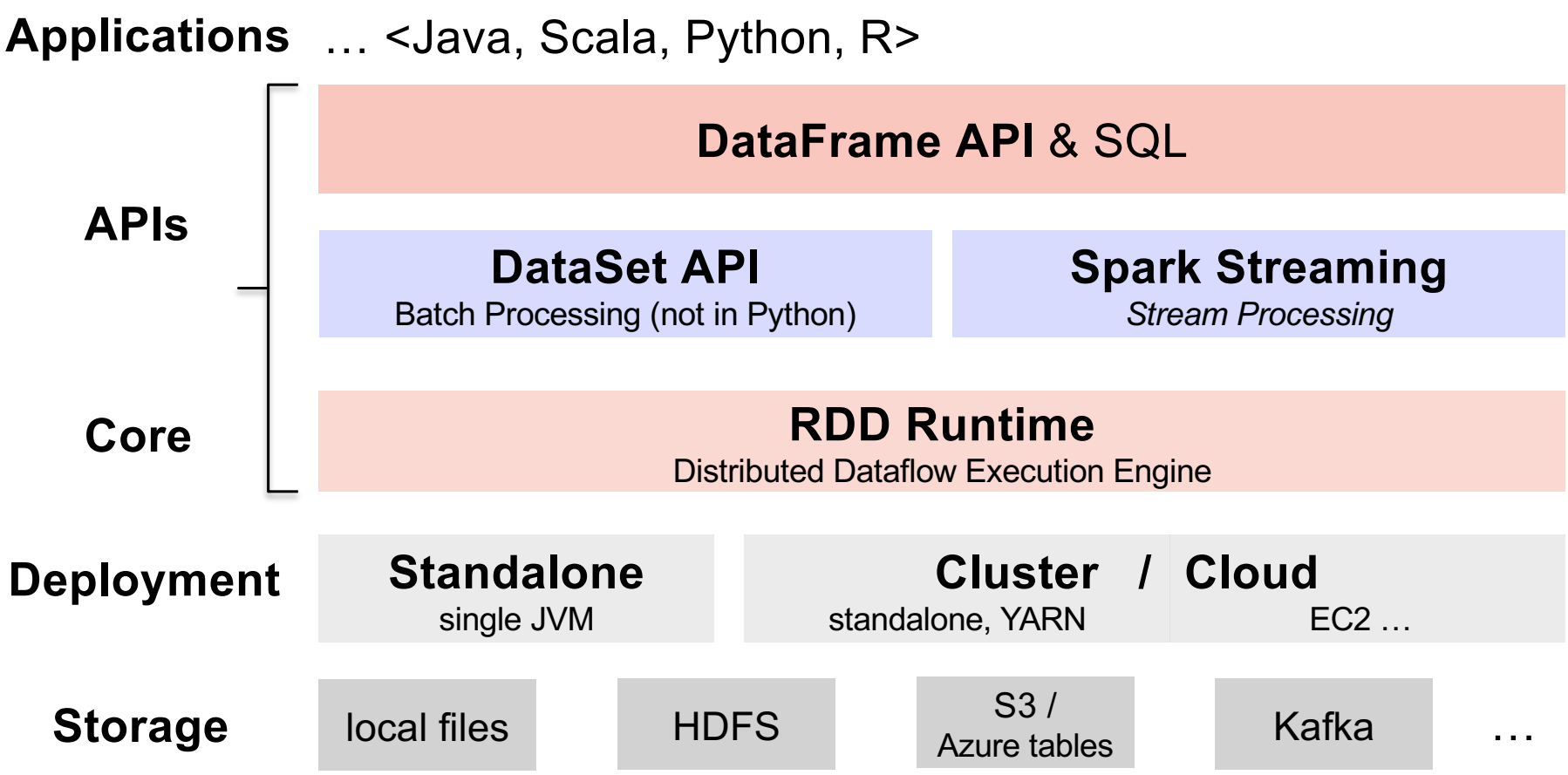
- The success of Google's MapReduce led to the development of open-source implementations, notably Apache **Hadoop**

Apache Spark

(https://spark.apache.org)

- **In-memory** framework for distributed, iterative computation
- Core: augment data flow model with **Resilient Distributed Dataset (RDD)**
  - RDD: fault-tolerance, *in-memory* storage abstraction
  - **Immutable** collections partitioned across cluster that can be rebuilt if a partition is lost
  - Created by transforming data in stable storage using data flow operators (map,filter,group-by…)
  - Can be *cached* across parallel operations
- Parallel operations on RDDs
  - Reduce, collect, count, save, …
- Supports to develop sophisticated (distributed) data analysis programs
  - Multiple language bindings (Java, Python, R, Scala, …)
  - Compatible with existing open source ecosystem (Hadoop/HDFS/YARN)
  - Interoperate with existing storage and input formats (e.g., HDFS, S3, JDBC, ..)
  - Current version (2025): v3.5.x

Apache Spark System Stack



Spark Program Pattern

- Regular programs in Java / Scala / Python / R

1. Initialise the runtime **environment**
2. Load or create **source** data
3. Specify the **data transformations**
   - Different APIs available: RDD API, Dataset API, DataFrame API
   - Might include usage of self-defined UDFs
4. Specify where the **output** should go
5. **Execute**

Spark Program

- A Spark program is just a regular **main** program that creates **`SparkSession`** object, which is used to access Spark (local or in a cluster).
  - In Python, this object is of type **`pyspark.sql.SparkSession`**
- The spark session provides methods for
  - configuration
  - data input
  - data output
  - Different specialised execution contexts (e.g. SQL)
  - ...
- The data processing steps are defined using either DataFrames or RDDs

pySpark Example: WordCount – using DataFrame API

```
from pyspark.sql import SparkSession
import pyspark.sql.functions as f

spark = SparkSession.builder \
                    .appName("WordcountExample") \
                    .getOrCreate()

text_df= spark.read.text("shakespeare-hamlet.txt")
counts = text_df.withColumn('word', f.explode(f.split(f.col('value'), ' '))) \
                .groupBy('word') \
                .count() \
                .sort('count', ascending=False)

counts.write.csv("wordcount")
```

[Cf.: http://spark.apache.org/examples.html]

Spark DataFrame API

- DataFrame: dataset with named columns; strongly typed; built-in functions
  - Main **data transformations**:
    - **select**(attrs) new DataFrame with selected columns
    - **filter**(condition) new DataFrame with rows that fulfil given filter condition
    - **orderBy**(attr) sorting
    - **groupBy**(attr) grouping by attribute(s)
    - Aggregation functions on grouped data such as **count**(), **avg**(), **max**(), **min**(), **sum**()
    - Join operations: **join**(other, condition), **crossJoin**(other); also **union**() and **intersect**()
  - **Actions**:
    - **show**(), **collect**(), **take**(), **save**(), **cache**(), ..
  - Conversion to Pandas Dataframe with **toPandas**() function
- Additional functions that can be used as expressions to operations: **pyspark.sql.functions**
- https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html

Spark DataFrame API Execution

- Spark uses ***lazy evaluation***
  - **Data transformations** are specified without actually being executed until result is required by a Spark action
  - DataFrame **Actions** triggering execution:
    - show(), collect(), take(n), saveAsTextFile(path), reduce(func), count()…
- Internally translated into calls to Spark's core RDD API
  - E.g. see DAG Visualisation in Spark UI on port :4040
  - or: DataFrame.explain() command
- Operations automatically parallelised internally

PySpark SQL

- PySpark SQL is a module in Spark that provides a higher-level abstraction for working with structured data and can be used with SQL queries.

- Running SQL-like queries in PySpark involves the following steps:
  1. **Initialize a SparkSession** – This serves as the entry point to using Spark functionality.
  2. **Load your data into a DataFrame** – For example, read data from CSV files or other supported sources.
  3. **Register the DataFrame as a temporary view** – This allows you to run SQL queries against it.
  4. **Execute SQL queries** – Use the spark.sql() method to run queries on the registered view.
  5. **Process the query results** – You can continue working with the returned DataFrame for further analysis or transformations.

PySpark SQL

- Registering a DataFrame as a temporary view allows you to query the DataFrame using SQL syntax through SparkSession's SQL engine.
  - The temporary view is scoped to the SparkSession in which it was created. It is only available for the duration of that session and does not persist across sessions.

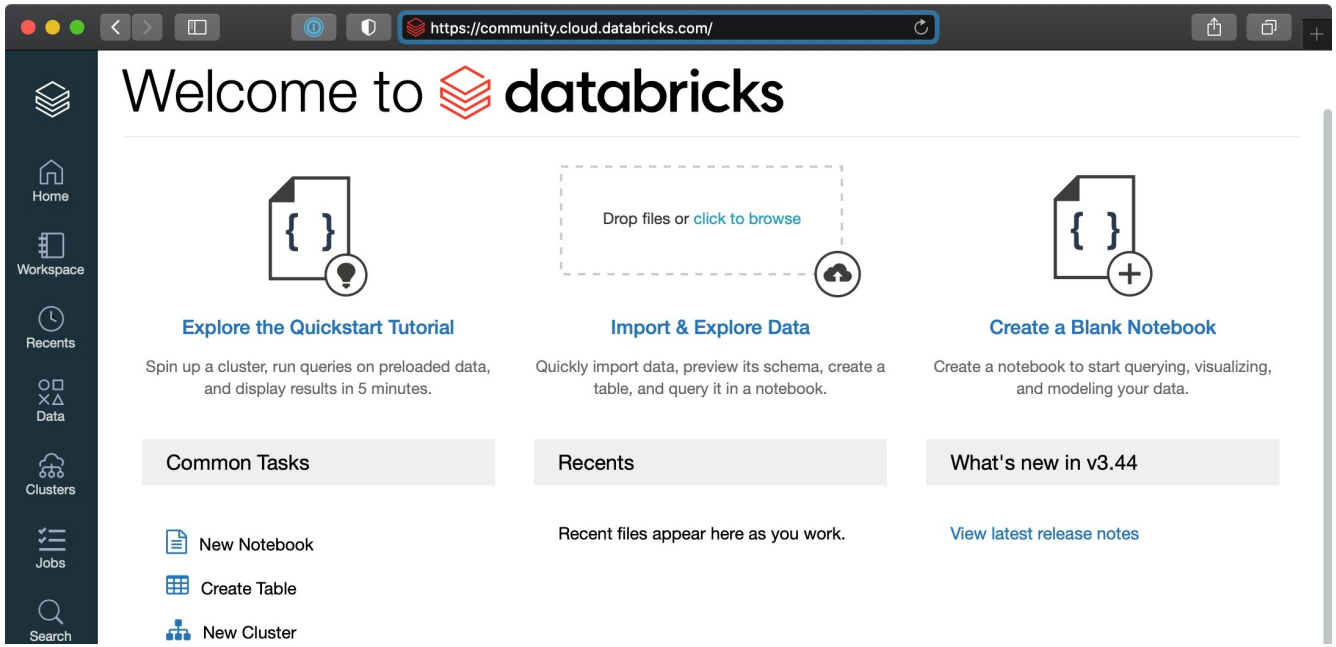
```
# Create temporary table
spark.read.option("header",True).csv("zipcodes.csv") \
     .createOrReplaceTempView("zipcodes")
```

- PySpark SQL to select columns

```
spark.sql("SELECT country, city, zipcode, state FROM ZIPCODES") \
      .show(5)
```

PySpark and Jupyter

- Spark clusters can be accessed from Jupyter notebooks
  - For example: **Sparkmagic kernel extensions** provide a dedicated pySpark kernel mode where each cell by default is executed in cluster mode
- Alternative: Cloud-Hosted Spark environment with notebook interface, such as Databricks



Apache Flink Project

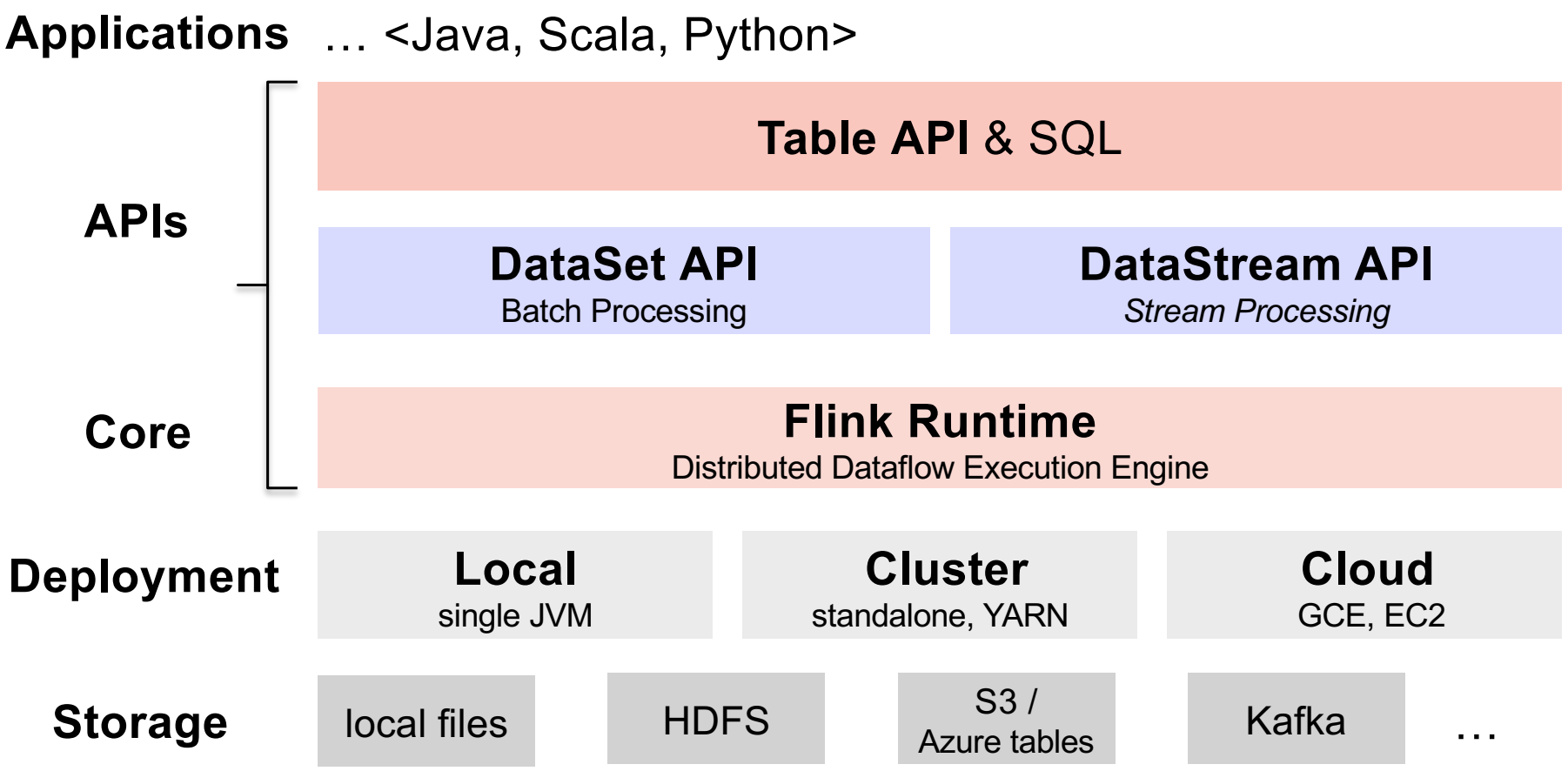
- Started in 2010 as an EU/DFG research project ('Stratosphere') by database research groups at TU Berlin, Humboldt University Berlin and HPI Potsdam
- Since Feb 2015, Apache top-level project (Apache Flink)
  - backed by spin-off company “Data Artisans” -> acquired by Alibaba in 2019

- Efficient data flow runtime on top of Hadoop/HDFS/YARN
  - Similar scalability and fail safety
  - More powerful data flow operators and optimization component
- Seamlessly integrates into existing Hadoop infrastructure

Core Features

- Core APIs: DataSet and DataStream
  - on top of either: **Table API**
- Multiple **data transformations,** supporting UDFs:
  - Join, Cross, Union, Iterate, …
- **In-memory pipelining** between operators
- Support for **Iterative Algorithms**
- **Lazy Evaluation**
- Support for Java, Python, and Scala

Apache Flink System Stack



Flink Program Pattern

- Regular programs in Java / Scala / Python

1. Initialise the runtime **environment**
2. Load or create **source** data
3. Specify the **data transformations**
   - This might include usage of self-defined UDFs
4. Specify where the **output** should go
5. **Execute**

WordCount in Flink (Python)

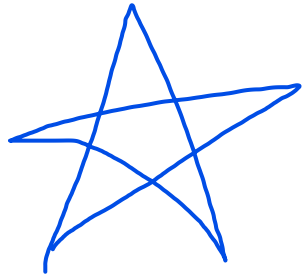
```
from flink.plan.Environment import get_environment
from flink.functions.GroupReduceFunction import GroupReduceFunction

class Sum(GroupReduceFunction):
    def reduce(self, iterator, collector):
        word, count = iterator.next();
        count += sum([x[1] for x in iterator])
        collector.collect((word,count))

env = get_environment()
data= env.read_text("hdfs://...");
data.flat_map(lambda x,c: [(word,1) for word in x.lower().split()]) \
    .group_by(0) \
    .reduce_group(Sum(), combinable=True) \
    .write_csv("hdfs://...")
env.execute(local=True)
```

[Cf.: https://nightlies.apache.org/flink/flink-docs-release-1.4/dev/batch/python.html]

Flink Plan Execution

- Lazy evaluation
  - In Flink program, one specifies an execution plan, e.g. using DataSet API
  - Plan gets actually executed by Flink only when **env.execute**() is called (only this starts executing Flink job; might include writing to disk)
  - Some exceptions in DataSet API which trigger an immediate eager execution:
    - **plan.print**() is called (print on console)
    - **plan.collect**() is called (retrieves results into program variable)
    - **env.getExecutionPlan**() is called

Differences between Spark and Flink



	Apache Spark	Apache Flink
Principle	set-oriented data transformations in stages	transformations by iterating over collections with pipelining
Data Abstraction	RDD	DataSet
Processing Stages	separate stages	overlapping stages
Optimiser	cost-based + adaptive exec	cost-based, integrated into API
Batch Processing	RDD	DataSet
Stream Processing	micro-batching	pipelining; DataStream API

Summary

- Several approaches exist for more sophisticated data analytics on top of a distributed Hadoop/HDFS infrastructure
  - strive to make data processing easier to program (more high-level)
  - support for iterative and streaming data processing
  - while keeping scalability and fault tolerance of MR
- **Apache Spark**
  - distributed, in-memory focused RDD Runtime
  - SparkSQL + DataFrame API abstractions on top which support, e.g. joins
  - Lazy evaluation principle
- **Apache Flink**
  - Powerful data flow language on top of its own pipelined data engine
  - Integrates many ideas from traditional data processing, focus on **pipelined execution** model and automated plan optimization
  - Strong for data stream processing