CS4225: Big Data Systems for Data Science

by: Jiahao

▷ what is data science?

"data science is an interdisciplinary field about processes and systems to extract knowledge or insights from data in various forms"

▷ challenges of big data

↳ volume

scale of data

- challenges of large volume
  - performance
  - reliability
  - cost
  - algorithm design complexity
- more machines ⇒ increased change for machine failure
- increase in volume ⇒ computation becomes a problem

↳ velocity

speed of new data (streaming data)

- challenges of high velocity
  - performance
  - cost
  - reliability
  - algorithm design complexity

↳ variety

format of data

- why it matters?
  - "one size does not fit all"
  - data integration
  - multi-modal learning

↳ veracity

accuracy of data (and importance of data quality)

- why it matters?
  - dirty and noisy data
  - data provenance (origin of data)
  - data uncertainty
- $\sim 3.3\%$ of data in some of the most popular datasets are mislabelled
- one of the biggest challenges in data in companies
- data sources span from log, transactions, events, emails, sensors, external feeds, RFID scans, etc.

▷ virtuous product cycle

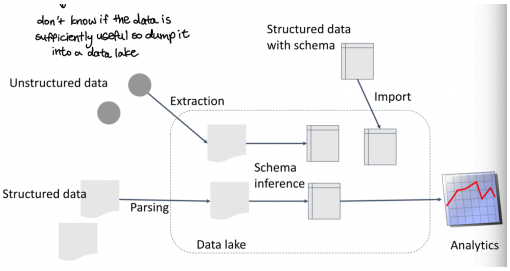
"why do we have big data problems?"

1. a useful service
2. analyze user behavior to extract insights
3. transform insights into action
4. repeat

- general cycle requires data to be captured

↳ data lake

- since the above data collection is very vast, collect everything possible and dump it into a data lake



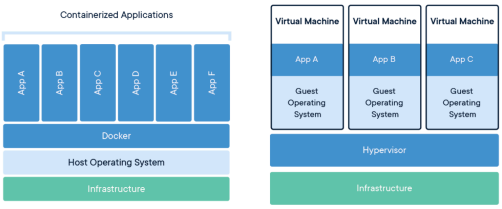
▷ cloud computing

↳ utility computing

- computing resources as a metered service ("pay as you go")
- ability to dynamically provision virtual machines
- used for
  - scalability; "infinite" capacity
  - elasticity: scale up or down on demand

↳ enabling technologies

- virtualization: virtual machines → enable sharing of hardware resources by running each application in an isolated virtual machine
  - high overhead as each VM has its own OS
- containerization: containers → enable lightweight sharing of resources, as applications run in an isolated way, but still share the same base OS
  - container is a lightweight software package that encapsulates an application and its environment



↳ everything as a service

- Infrastructure as a Service (IaaS): utility computing
  - user rents a virtual machine and makes all decisions on what to run on it
  - e.g. AWS EC2, Rackspace, GCE
- Platform as a Service (Paas)
  - provides hosting for web applications and takes cre of the hardware maintenance, upgrades, etc.
  - e.g. Google App engine
- Software as a Service (SaaS)
  - user typically does not write code, and is just using an existing app
  - e.g. Gmail, Dropbox, Zoom

▷ cloud computing and big data systems in AI

- Rockset → real-time search and analytics database
- enhance OpenAI's ability to quickly access and analyze vast amounts of information
- lead to faster and more accurate responses from AI models

▷ data center is the computer

"many machines in a data center as a big "processing unit" being used to solve a problem, rather than just as independent machines"

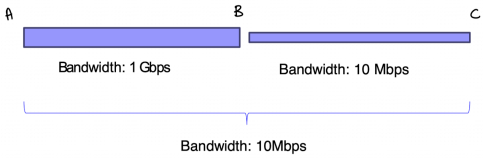
- comprised of server racks and cluster switches
- hides system-level details from the developer
  - no race conditions/lock contention
  - no need for worry about reliability and fault tolerance
- all about the right level of abstraction
  - moving beyond single machine architecture
  - API of the datacenter
    - separating the *what* from the *how*
      - developers specifies the computation to perform
      - execution framework ("runtime") handles actual execution

▷ bandwidth

maximum amount of data that can be transmitted per unit time (e.g. in GB/s)

- when transmitting large amount of data, bandwidth indicates roughly how long the transmission will take
- affect file transfer speed for large files

↳ bandwidth along path



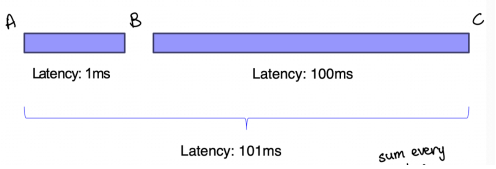
- approximately the minimum bandwidth along the path
- rate of data flow is bottlenecked by the lowest bandwidth segment

▷ latency

time taken for 1 packet to go from source to destination (one-way) or from source to destination back to source (round trip) (e.g. in ms)

- when transmitting very small amount of data, latency indicates how much delay there will be
- affects delay when communicating, e.g. over a Zoom call

↳ latency along path



- combines approximately additively
- other factors include: transport protocols, congestion, queueing, etc.

▷ throughput

rate at which some data was actually transmitted across the network during some period of time

- throughput ≤ bandwidth
- affect file transfer speed for large files

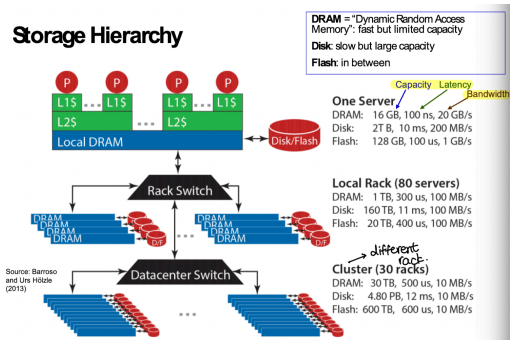
▷ cost of data access in data center

data access may cross several components including hard disk, DRAM, rack switch, and cluster switch

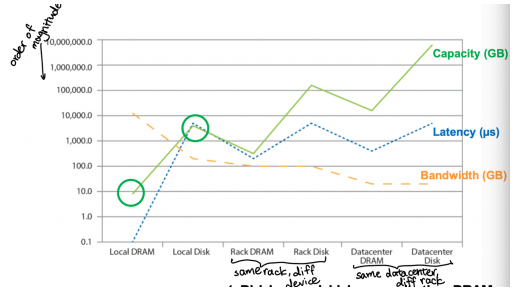
- approximate for the scale or order of magnitude
- assumes (a) one-way communication, (b) no failures, and (c) peak hardware bandwidth

↳ storage hierarchy

- focused on the cost to move data



↳ capacity of storage hierarchy



1. disk has much higher capacity than DRAM (more expensive with more storage)
2. storage hierarchy capacity increases from local server → rack → datacenter
3. disk reads are more expensive than DRAM, in terms of higher latency and lower bandwidth
4. latency increase over storage hierarchy
5. bandwidth decreases over storage hierarchy

↳ data flow paths

⇝ within local DRAM

sending data from server A's DRAM back to its own DRAM

- account for latency and bandwidth of DRAM

⇝ to local disk

sending data from server A's DRAM to its own disk

- local DRAM → local disk
- sum of latency across local DRAM and local disk
- minimum bandwidth between local DRAM and local disk

⇝ to rack DRAM

sending data from server A's DRAM to DRAM of another server B in the same rack

- local DRAM → rack switch → server B's DRAM
- rack switch accounts for some latency and bandwidth
- sum of latency across local DRAM and rack DRAM
- minimum bandwidth between local DRAM and rack DRAM

⇝ to datacenter DRAM

sending data from server A's DRAM to DRAM of another server C in a different rack

- local DRAM → rack switch → datacenter switch → server C rack switch → server C DRAM
- datacenter switch accounts for some latency and bandwidth
- overall bandwidth keeps decreasing since the $\min\{\text{path}\}$ is going to decrease

▷ four Big Ideas

of massive data process in data centers

↳ scale "out", not "up"

- scale "out" → combining many cheaper machines
  - adding more servers to architecture to spread the workload
  - fault tolerant
  - easier to scale out
- scale "up" → increasing the power of each individual machine
  - adding further resources to increase computing capacity of physical servers
- horizontal vs vertical scaling

↳ seamless scalability

- aggregated bandwidth scales with number of machines
- design scales linearly with number of machines → trade more machines with better performance
- if processing a certain dataset takes 100 machine hours, ideal scalability means using a cluster of 10 machines to do it in 10 hours

↳ move processing to the data

- move task to machines where data is stored
- rather than accessing data across the datacenter

↳ process data sequentially, avoid random access

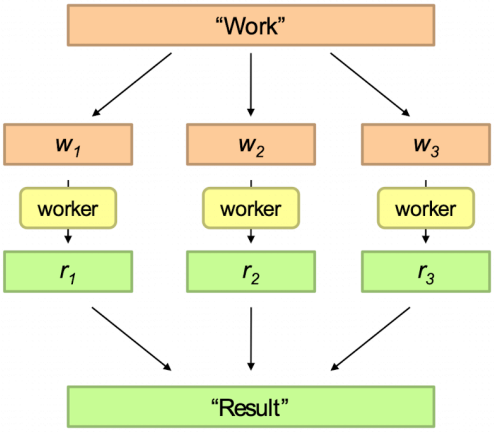
- seeks are expensive, disk throughput is reasonable

▷ abstractions of big data systems

- e.g. Google indexes over 20 billion web pages (more than 400TB)
  - if each computer reads 30-35 MB/sec from disk
  - takes ~ 4 months to read the web and ~ 1000 hard drives to store the web
  - takes even more to do something useful with the data

▷ divide and conquer

- partition the work and combine the results



↳ challenges

1. how do we assign work units to workers?
2. what if we have more work units than workers?
3. what if workers need to share partial results?
4. how do we aggregate partial results?
5. how do we know all the workers have finished?
6. what if workers die/fail?

↳ machine failures

- one server stay up for 10 years
- 3650 servers ⇒ expect to lose 1/day
- out of memory is also very common

↳ synchronization

- unsure of
  - order of workers running
  - worker interruptions
  - worker communication of partial results
  - unknown order of shared data access
- requires use of semaphores, conditional variables, barriers
- other concurrency issues like deadlocks, livelocks, race conditions, etc.

↳ programming difficulty

- concurrency is difficult to reason about
  - at the scale of datacenters and across datacenters
  - in the presence of failures
  - in terms of multiple interacting services
- debugging is challenging
- reality
  - one-off solutions and custom code
  - dedicated libraries to handle
  - burden on programmer to explicitly manage everything

▷ typical big data problem

1. iterate over large number of records
2. extract something of interest from each (**map**)
3. shuffle and sort intermediate results (**shuffle**)
4. aggregate intermediate results (**reduce**)
5. general final output

- key idea → provide a functional abstraction for map and reduce

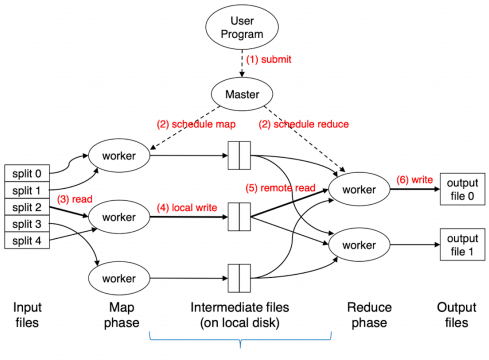
▷ MapReduce

- map phase → 1 map task, multiple calls to map function
- shuffle phase → data is **sorted by key** and distributed across many machines (not seen to users)
- reduce phase → reduce function called once for each intermediate key; 1 reduce task, multiple calls to reduce function
- interfaces for programmers
  - $\text{map}(k_1, v_2) \to \text{List}(k_2, v_2)$
  - $\text{reduce}(k_2, \text{List}(v_2)) \to \text{List}(k_3, v_3)$
    - all values with the same key are sent to the same reduce task
    - receives the values from the shuffle/sort phase

↳ execution framework

- handles scheduling → assigning workers to map and reduce tasks
- handles shuffle phase → data is shuffled and sorted in between map and reduce phases
- handles synchronization → executes tasks in appropriate order
- handles faults → detects worker failures and restarts
- happens on top of Hadoop Distributed File System

↳ implementation



1. submit → user submits MapReduce program (map and reduce function, configuration) to master node
2. schedule → master schedules resources for map and reduce tasks; master does not handle actual data
3. read → input files are separated into "splits" of around 128MB each
   - corresponds to one map task → work that needs to be done to process this split
   - workers execute map tasks 1 at a time
   - chunk size too big ⇒ limited parallelism (too little chunks)
   - chunk size too small ⇒ high overhead on master node to schedule
4. local write → each worker writes the output of the map function to intermediate files on its own local disk
   - files are partitioned into N chunks for N reduce tasks
   - files are sorted by key within each chunk
   - may run on separate machines
5. remote read → each reduce worker is responsible for ≥ 1 keys
   - for each key, it reads the data it needs from the corresponding partition of each mapper's local disk
6. write → output of the reduce function is written to disk
   - usually to HDFS
   - one file per reduce function

- map phase → each worker iterates over each `<key, value>` tuple in its input split, and computes the map function on each tuple
- shuffle phase → comprised of local write and remote read
  - runs on the worker nodes
- reduce phase → reducer receives all its needed key value pairs, with keys arriving in sorted order
  - computes the reduce function on the values of each key
  - processes keys in sorted order
- map must complete before reduce starts to avoid computing wrong results in reduce phase

↳ components

1. **worker** → component of the cluster that performs storage and processing tasks
   - single worker handles multiple map tasks
   - worker is reassigned when completed
2. **task** → basic unit of work (typically chunk size of 128MB)
   - job required to process one split
   - single task involves many calls to function
   - map tasks → number of input splits
   - reduce tasks → specified by user when configuring job
3. **function** → single call to user-defined $\text{map}(k_1, v_2) \to \text{List}(k_2, v_2)$ function
   - map function calls → number of input key-pair values
   - reduce function calls → number of distinct intermediate keys

↳ partitioners

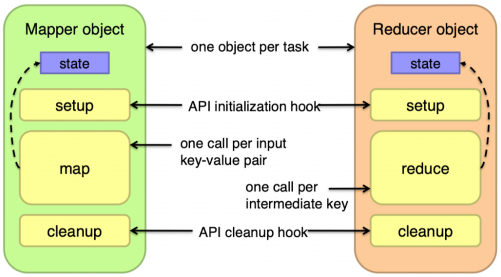
- by default, keys are assigned to reducers via some hash function
- declare custom partitions to better spread out the load among reducers (if some keys have more values than others)
- runs in local write phase

↳ combiners

- locally aggregate output from mappers
- "mini-reducers"
- user's responsibility to ensure combiner does not affect correctness of final output
- *not guaranteed to always run
- combiners should be idempotent
- usually useful for operations that are associative and commutative
- runs after map phase, during local write, but before disk write
- reduces disk writes, but increases demand for memory working set
- rule of thumb: mapper → reducer should be the same as combiner → reducer
  - accounts for cases where the combiner does not run

▷ performance guidelines for basic algorithmic design

1. linear scalability → more modes can do more work at the same time
   - linear on data size
   - linear on computer resources
2. minimize disk and network I/O
   - minimize disk I/O → sequential vs random access
   - minimize network I/O → send data in bulk vs in small chunks
3. reduce memory working set of each task/worker
   - working set → portion of memory actively used during algorithm execution
   - large working set ⇒ high memory requirements ⇒ increased probability out-of=memory errors



▷ HDFS assumptions

- use of commodity hardware instead of "exotic" hardware
  - scale "out", not "up"
- high component failure rate
  - inexpensive commodity components fail all the time
- "modest" number of huge files
- large sequential reads instead of random access
  - better throughput

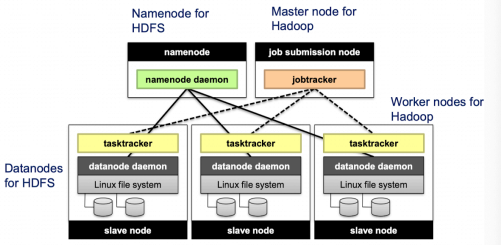
▷ HDFS design decisions

- files stored as chunks (or blocks) → fixed size, defaults to 128MB
- reliability through replication → each chunk is stored as multiple replicas, defaults to 3
- single master to coordinate access, keep metadata → simple centralized management

▷ HDFS interface

- similar to regular file systems (with base commands) with extra commands to transfer files to and from the local filesystem
- designed for **write-once, read-many**
  - does not support modifying files apart from appending
- handles very large files, while providing fault tolerance and replication

▷ HDFS architecture

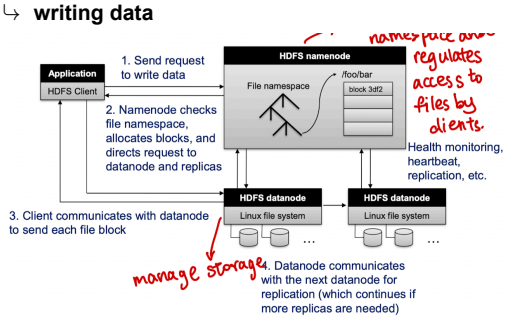


namenode → master server that manages the filesystem namespace

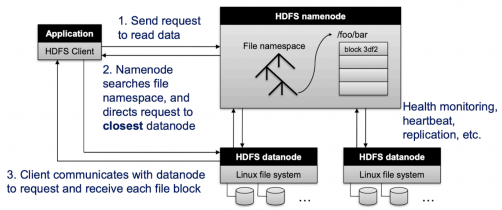
- holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- directs clients to datanodes for reads and writes
- no data is moved through the namenode
- maintains overall health by performing heartbeats and block rebalancing (to ensure data is distributed close to evenly)
- data is lost ⇒ all files on the filesystem cannot be retrieved since there is no way to reconstruct them from raw block data
  - Hadoop provides ways of improving resilience → backups and checkpointing

data locality → Hadoop tries to schedule map tasks to run on machines that already contain the needed data

↳ writing data

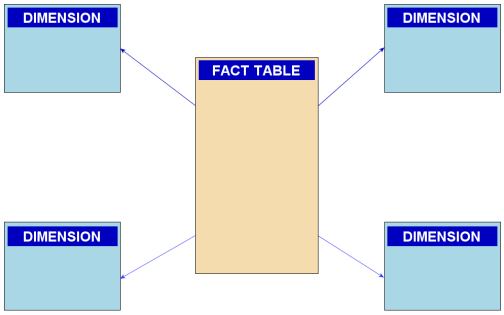


↳ reading data



▷ star schema

central facts table contains foreign keys to all other dimensional tables



▷ projection

selecting a subset of columns from a table

- **map** → take in a tuple (tuple ID as key) and emit new tuples with appropriate attributes
- **reduce** → not needed
  - no shuffle step ⇒ more efficient job

▷ selection

filtering rows by a given predicate

- **map** → take in a tuple (tuple ID as key) and emit only tuples that meet the predicate
- **reduce** → not needed

▷ group by… aggregation

grouping data by columns and calculating aggregate values

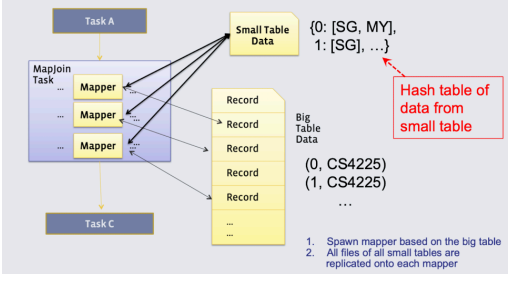
- **map** → over tuples, emit `<product_id, price>`
- Hadoop automatically groups these by key
- **reduce** → compute aggregation
- optimize using combiners

▷ relational joins (inner join)

- types of relationship: many-to-many, one-to-many, and one-to-one

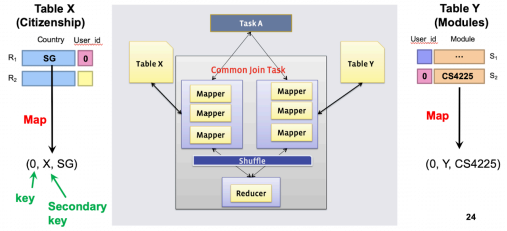
↳ broadcast/map join

- requires one of the tables to fit in memory
- all mappers store a copy of the small table (converted to a hash table with key as keys to join by)
- mappers iterate over big table and join the records with the small table
- performed before map function
- minimizes cost incurred for sorting and merging in the shuffle and reduce stage
- improves performance of task by decreasing time to finish task
- only useful when one table can fit in memory



↳ reduce-side/common join

- does not require a dataset to fit in memory
- slower than broadcast join
- different mappers operate on each table and emit records, with key as the variable to join by
- performed as a reduce step
  - use secondary sort to ensure all keys from X arrive before Y
  - hold keys from X in memory and cross the with records from Y



▷ secondary sort

allows reducer to receive values in sorted order

- keys in each reduce task arrives in sorted order, but values arrive unsorted
- makes computing certain statistics more convenient, particularly for data with timestamps, and for medians and quantiles
  - cheaper to sort through secondary sorting than use sort in reducer
    - uses inbuilt distributed sorting Hadoop performs

1. create a composite key of the `<key, value to sort>` → `(key, value to sort)`
   - the key is the natural key
   - the value to sort is the natural value
2. define custom comparator to compare by `key` then `value to sort`
3. implement custom partitioner to partition by `key` only

▷ similarity

- many problems can be expressed as finding "similar" objects
- e.g.
  - documents with similar words (for duplicate detection or classification or mirror websites or similar news articles)
    - all pair similarity → given a large number N of documents, find all "near duplicate" pairs, e.g. using Jaccard distance below a threshold
    - similarity search → given a query document D, find all documents which are "near duplicates" with D
  - customers who purchased similar products (for recommendation)
  - users who visited similar websites (for grouping users with similar interests)

↳ measures

nearest neighbors → points that are a "small distance" apart

- distance between objects `x` and `y` measured using function `d(x, y)` → distance measure
- similarity is opposite of distance

↝ euclidean distance

$$d(\vec{a},\vec{b}) = \left|\left|\vec{a}-\vec{b}\right|\right|_2 = \sqrt{\sum_{i=1}^{D}(a_i-b_i)^2}$$

↝ manhattan distance

$$d(\vec{a},\vec{b}) = \left|\left|\vec{a}-\vec{b}\right|\right|_1 = \sum_{i=1}^{D}|a_i-b_i|$$

↝ cosine similarity

$$s(\vec{a},\vec{b}) = \cos(\theta) = \frac{\vec{a}\cdot\vec{b}}{||\vec{a}||\cdot\left|\left|\vec{b}\right|\right|}$$

- only considers direction; cosine similarity does not change if we scale $\vec{a}$ or $\vec{b}$

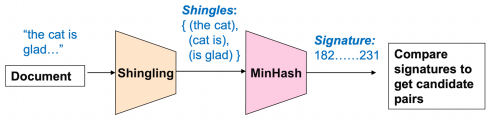
↝ jaccard similarity

$$s_{\text{jaccard}}(A,B) = \frac{|A\cap B|}{|A\cup B|}$$

↝ jaccard distance

$$d_{\text{jaccard}}(A,B) = 1 - s_{\text{jaccard}}(A,B)$$

▷ similar document cleaning



↳ shingling

convert documents to sets of short phrases (shingles)

- k-shingle (or k-gram) → sequence of k tokens that appears in the document
- every document D_i is a set of its k-shingles C_i
- similarity → represent as a matrix with columns as documents, rows as shingles
  - measure similarity via Jaccard similarity → number of times both shingles appear over the number of times either shingle appears

$$\frac{C_1 \cap C_2}{C_1 \cup C_2}$$

↳ **min-hashing**

convert these sets of shingles to short "signatures" of each document, while preserving similarity

- signature → block of data representing the contents of a document in a compressed way
- documents with the same signature are candidate pairs for finding near-duplicates
- pairwise Jaccard similarities for a million pairs is too slow → min-hashing gives a fast approximation
- hash each column C to a small signature $h(C)$ such that
  - $h(C)$ is small enough that the signature fits in RAM
  - highly similar documents have the same signature
  - take the minimum hashed score
  - $h(\cdot)$ should be: $s(C_1, C_2)$ is high $\Rightarrow$ high probability that $h(C_1) = h(C_2)$
- key property → probability that two documents have the same MinHash signature is equal to their Jaccard similarity

↳ **candidate pair**

documents with the same final signature

↳ **MapReduce implementation of similarity cleaning**

- **map** → read over the document and extract shingles, hashing each single and taking the minimum
  - emit `<signature, document_id>`
- **reduce** → receive all documents with a given MinHash signature
  - generate all candidate pairs from these documents
  - (optional) compare each pair to check for actual similarity

▷ **clustering**

separates unlabelled data into groups of similar points

- clusters should have high intra-cluster similarity and low inter-cluster similarity
- applications
  - microbiology → find groups of related genes
  - recommendation & social networks → find groups of similar users
  - search & information retrieval → grouping similar search (or news) results

↳ **k-means algorithm**

1. initialization: pick k random points as centers
2. repeat:
   1. assignment: assign each point to nearest cluster
   2. update: move each cluster center to average of its assigned points
   3. stop: if no assignments change

↝ **MapReduce implementation v1**

```
class Mapper
  method Configure()
    c <- LoadClusters()

  method Map(id i, point p)
    n <- NearestClusterId(clusters c, point p)
    p <- ExtendPoint(point p)
    Emit(clusterid n, point p)

class Reducer
  method Reduce(clusterid n, points [p1,
p2, ...])
    s <- InitPointSum()
    for all point p in points do
      s <- s + p
    m <- ComputeCentroid(point s)
    Emit(clusterid n, centroid m)
```

- `Map` runs for every point (`n` times), for every iteration `m` and there are `d` dimensions
  - total disk I/O: $O(nmd)$

↝ **MapReduce implementation v2**

```
class Mapper
  method Configure()
    c <- LoadClusters()
    H <- InitAssociativeArray()

  method Map(id i, point p)
    n <- NearestClusterId(clusters c, point p)
    p <- ExtendPoint(point p)
    H{n} <- H{n} + p

  method Close()
    for all clusterid n in H do
      Emit(clusterid n, point H{n})

class Reducer
  method Reduce(clusterid n, points [p1,
p2, ...])
    s <- InitPointSum()
    for all point p in points do
      s <- s + p
    m <- ComputeCentroid(point s)
    Emit(clusterid n, centroid m)
```

- in-memory storage is used to avoid sending for every point
  - total disk I/O: $O(kmd)$ where k is the number of clusters, since we emit per cluster

▷ **photo synchronization system**

goal → design a system like iCloud/Google Photos that synchronizes photos across each user's devices

challenges

1. connectivity → devices may lose connection to the internet
   - data can go "out of sync" between devices
   - may result in more conflicting files/edits
2. eventual consistency → files should be consistent assuming devices are connected to the internet for a sufficiently long time

▷ **NoSQL**

non-relational database where data is stored in a format other than relational tables

- types:
  1. key-value
  2. wide column
  3. documents
  4. graph
  5. vector
- properties:
  1. horizontally scalability (volume/velocity)
  2. replicate/distribute data over many servers (volume/velocity)
  3. simple call interface
  4. often weaker concurrency model than RDBMS
  5. efficient use of distributed indexes and RAM
  6. flexible schemas (variety)
- depends on needs of application
  - denormalization is suitable
  - complexity of queries
  - importance of consistency
  - data volume/need for availability
- pros
  1. flexible/dynamic schema → suitable for less well-structured data
  2. horizontal scalability
  3. high performance and availability → due to their relaxed consistency model and fast read/writes
- cons
  1. no declarative query language → query logic (e.g. joins) may have to be handled on the application side
  2. weaker consistency guarantees → application may receive stale data that needs to be handled by application

↳ **key-value stores**

- stores associations between keys and values
- keys are usually primitives and can be queried
- values can be primitive or complex
  - usually cannot be queried
- suitable for
  - small continuous read and writes
  - storing basic information
  - when complex queries are not required/rarely required
- examples:
  - storing user sessions
  - caches
  - user data that is often processed individually

↝ **operations**

- very simple API
  - GET → fetch value associated with key
  - PUT → set value associated with key
- optional operations
  - multi-GET
  - multi-PUT
  - range queries

↝ **implementation**

- non-persistent
  - big in-memory hash table
  - e.g. Memcached, Redis
- persistent
  - data is stored persistently to disk
  - e.g. RocksDB, Dynamo, Riak

↳ **document stores**

- database has many collections (tables)
- collections have multiple documents (rows)
- document is a JSON-like object with fields and values
  - can be nested

↝ **querying**

- allow querying based on content of document
- CRUD
  - better than key-value store because of this

↳ **graph databases**

- represent data as nodes and relationships as edges

↳ **vector databases**

- store vectors → each row represents a point in d dimensions
  - usually dense, numerical, and high-dimensional
- allow fast similarity search → retrieve similar neighbors from the database
  - uses locality-sensitive hashing
- features → scalability, real-time updates, replication
- e.g. Milvus, Redis, Weaviate, MongoDB Atlas
- use in AI/ML
  - LLMs used to convert text into vectors (embeddings)
  - vision models convert images into embeddings
  - used to quickly search for similar embeddings

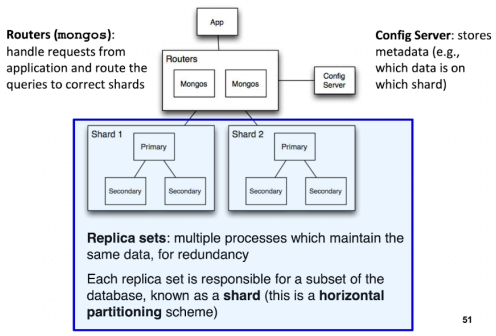
▷ **consistency**

- **consistency in ACID** → database must always obey certain predefined constraints
  - **strong consistency** → any reads immediately after an update give the same results on all observers
- **eventual consistency** → if the system is functioning and we wait long enough, eventually all reads will return the last written value
  - offers better availability at the cost of a much weaker consistency guarantee
  - acceptable for some applications (e.g. statistical queries, tweets, social network feed) but not for others (e.g. financial transactions)

▷ **BASE**

- Basically Available → basic reading and writing operations are available most of the time
- Soft state → without guarantees, some probability of knowing the state at any time (state might change even without input)
- Eventual consistency → contrast to "strong consistency"

▷ **architecture of MongoDB**



1. query issued to a router instance
2. router determines which shards to query using config server
3. query sent to relevant shards (partition pruning)
4. shards run query on their data, and send results back to router
5. router merges the query results and returns the merged results to the application

▷ replication in MongoDB

- standard configuration → 1 primary, 2 secondaries
- **writes**
  - primary receives all write operations
  - records writes onto its operation log
  - secondaries replicate this operation log, applying it to their local copies of the data, and acknowledge the operation
- **reads**
  - user configures "read preference" → decides whether secondaries can be read from (default) or the primary
  - allowing reading from secondaries can decrease latency and distribute load (improving throughput), but allows for reading stale data
- sharding does not exist

▷ joins in NoSQL

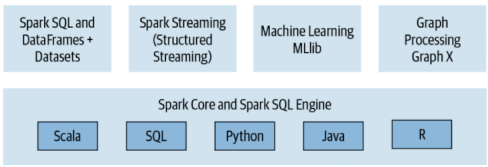
- denormalization/deduplication of data

▷ Hadoop vs Spark

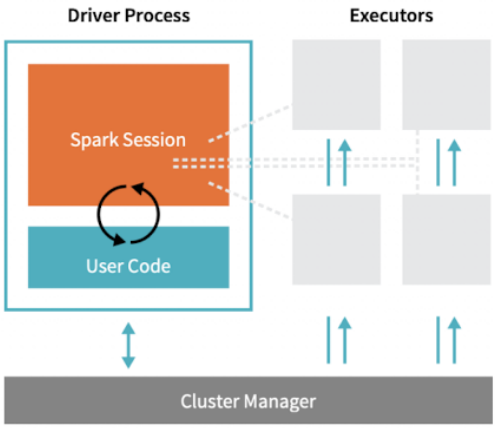
- issues with Hadoop MapReduce
  - network and disk I/O costs → intermediate data has to be written to local disks and shuffled across machines, which is slow
  - not suitable for **iterative** processing → individual step has to be modelled as a MapReduce job
- spark
  - stores most intermediate results in memory, making it much faster, especially for iterative processing
  - spills to disk if memory is insufficient
  - outperforms Hadoop in K-Means clustering and Logistic regression
  - easier to program with

```
val file = sc.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
          .map(word => (word, 1))
          .reduceByKey(_ + _)
counts.save("...")
```

▷ spark



↳ architecture



- driver process → responds to user input, manages the Spark application, and distributes work to Executors, which run the code assigned to them and send the results back to the driver
- cluster manager (Spark's standalone cluster manager, YARN, Mesos, or Kubernetes) → allocates resources when the application requests it
- local mode → all processes run on the same machine

▷ Resilient Distributed Datasets (RDDs)

- resilient → achieve fault tolerance through lineages
- distributed datasets → collection of objects that is distributed over machines

```
dataRDD = sc.parallelize(["Alice", "Bob", "Carol", "Daniel"], 3)
```

- split the above into three partitions
- immutable
- distributed at the start based on partition sizes

$$n = \text{len}(\text{arr})$$
$$\text{partition\_size} = \left\lfloor \frac{n}{k} \right\rfloor$$
$$\text{excess} = n - \left(\left\lfloor \frac{n}{k} \right\rfloor \times k \right)$$

- data read on each worker node in parallel, not on driver node
- instructs Spark how to compute the query
  - intention is completely opaque to Spark
  - Spark does not understand the structure of the data in RDDs or the semantics of user functions

▷ transformations

- way of transforming RDDs into RDDs

```
dataRDD = sc.parallelize(["Alice", "Bob", "Carol", "Daniel"], 3)
nameLen = dataRDD.map(lambda s: len(s))
```

- lazy transformations → transformation will not be executed until an action is called on it
  - advantages: spark can optimize the query plan to improve speed
- e.g. `map`, `order`, `groupBy`, `filter`, `join`, `select`
- executed in parallel

▷ actions

- trigger Spark to compute result from a series of transformations

```
dataRDD = sc.parallelize(["Alice", "Bob", "Carol", "Daniel"], 3)
nameLen = dataRDD.map(lambda s: len(s))
nameLen.collect()

# [5, 3, 5, 6]
```

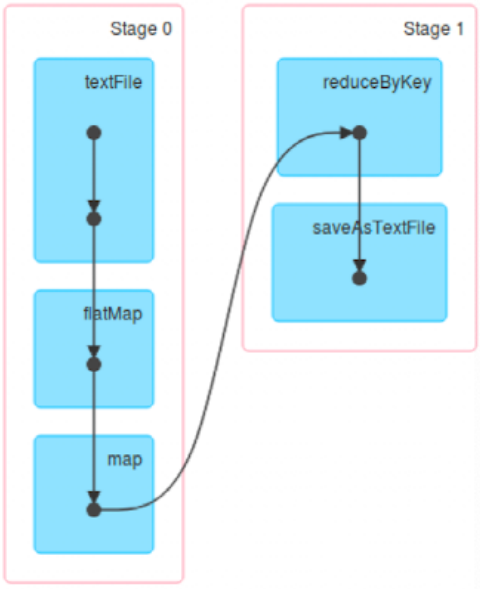
- `collect()` → retrieve all elements of the RDD to the driver node
- e.g. `show`, `count`, `save`, `collect`
- executed in parallel

▷ caching

- `cache()` → saves RDD to memory of each worker node
- `persist(options)` → used to save an RDD to memory, disk, or off-heap memory
- subsequent calls are made, it reads from the cache instead of recomputing
- principle
  - use caching when it is expensive to compute an RDD and it needs to be reused multiple times
  - worker nodes have not enough memory ⇒ evict the "least recently used" RDDs

▷ RDD transformation representation

- internally, Spark creates a DAG
- transformations construct the graph
- actions trigger computations on it



↳ dependencies

- **narrow dependencies** → each partition of the parent RDD is used by at most 1 partition of the child RDD
  - e.g. `map`, `flatMap`, `filter`, `contains`
- **wide dependencies** → each partition of the parent RDD is used by multiple partitions of the child RDD
  - e.g. `reduceByKey`, `groupBy`, `orderBy`
- consecutive narrow dependencies are grouped into stages
  - within stages → Spark performs consecutive transformations on the same machines
  - across stages → data needs to be exchanged across partitions (shuffled), writing intermediate results to disk
  - minimize shuffling is a good practice for improving performance

▷ fault tolerance

- Spark does not use replication to allow fault tolerance
- duplicating everything in memory is expensive
- lineage approach → if a worker node goes down, we replace it by a new worker node, and use the DAG to recompute the data in the lost partition
  - only need to recompute the RDDs from the lost partition

▷ DataFrames

- table of data, similar to tables in SQL
- higher level interface with more transformations that resemble SQL operations
- recommended interface for working with Spark
- all DataFrame operations are compiled down to RDD operations by Spark

```
flightData2015 = spark \
  .read \
  .option("inferSchema", "true") \
  .option("header", "true") \
  .csv("/...")

flightData2015.sort("count").take(3)
```

- tell Spark what to do, instead of how to do
- code is far simpler and more expressive
  - using a DSL similar to pandas
  - high-level DSL operators to compose the query
- Spark can inspect or parse the query and understand the intention → optimize or arrange the operations for efficient execution

↳ transformations

- use SQL queries

```
flightData2015.createOrReplaceTempView("flight_data_
maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")
maxSql.collect()
```

- equivalent in transformations
  - take in strings or column objects that represent the columns

```
from pyspark.sql.functions import desc

flightData2015 \
  .groupBy("DEST_COUNTRY_NAME") \
  .sum("count") \
  .withColumnRenamed("sum(count)", "destination_total") \
  .sort(desc("destination_total")) \
  .limit(5) \
  .collect()
```

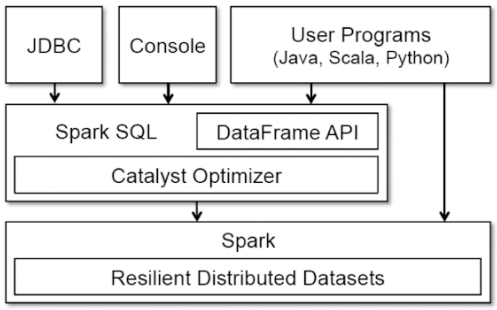
▷ Datasets

- similar to DataFrames but are type safe
  - in Spark Scala, DataFrame is an alias for `DataSet[Row]`
- not available in Python and R

```
case class Flight(DEST_COUNTRY_NAME: String, ORIGIN_COUNTRY_NAME: String, count: BigInt)

val flightsDF = spark.read.parquet("...")
val flights = flightsDF.as[Flight]
flights.collect()
```

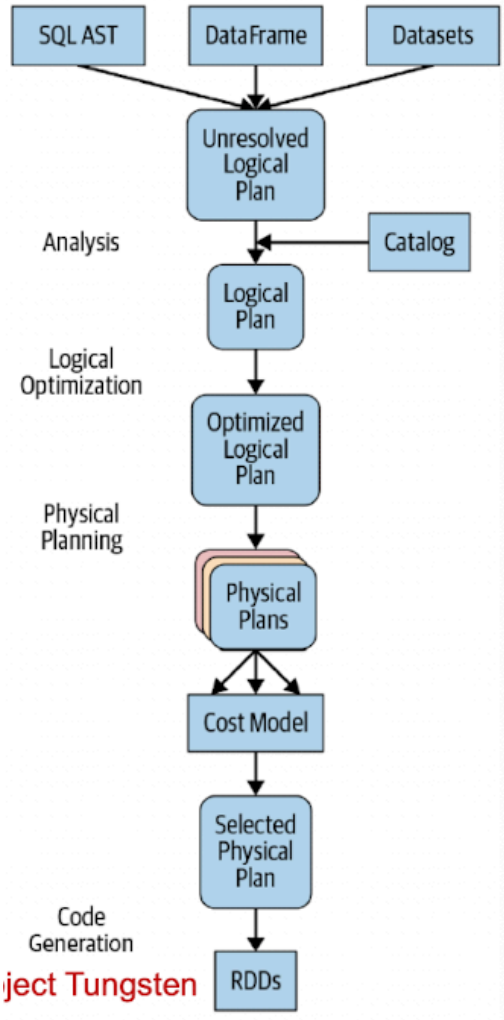
▷ Spark SQL

- unifies Spark components and permits abstraction to DataFrames/Datasets in Java, Scala, Python, and R
- keep track of schema and support optimized relational operations
- act as a common middleware for different data access layers
- before turning to Spark SQL
  - cluster resource provisioning appropriately to run 24/7
  - number of partitions for shuffles to be set much lower than batch queries
  - setting source rate limits for stability
  - multiple streaming queries in the same Spark application



↳ **catalyst optimizer**

- takes computational query and converts it into an execution plan through four transformational phases:
  1. analysis
  2. logical optimization → logical plan
     - benefit: performance parity across languages (same performance)
  3. physical planning → physical plan
  4. code generation (Project Tungsten)



↳ **Project Tungsten**

- substantially improve the memory and CPU efficiency of Spark applications
- push performance closer to the limits of modern hardware
- implemented using
  - memory management and binary processing
  - cache-aware computation
  - code generation
- enables unified API, one engine, automatically optimized

▷ **machine learning**

- classification → categorize samples into classes, given training data
- regression → predict numeric labels, given training data
- pipeline: preprocessing → training → testing → evaluation

↳ **preprocessing**

- **data quality** → poor data quality is an unfolding disaster
  - poor data quality costs the typical company at least 10% to 20% of revenue
- undervalued part of data science
  - avoids "garbage in, garbage out"
- **missing data**
  - causes → not collected, missing at random
  - handling → eliminate rows with missing values; fill in missing values (imputation)
    - based on mean/median
    - using regression model to predict the attribute given other attributes
    - dummy values

```
from pyspark.ml.feature import Imputer

imputer = Imputer(inputCols=['a', 'b'],
outputCols=['out_a', 'out_b'])
model = imputer.fit(df)
model.transform(df).show()
```

- **categorical encoding** → converting categorical features to numerical features
  - numerical values assigned in a way that represents the ordinal relationship or inherent order among the categories
  - e.g. [Low, Medium, High] → [0, 1, 2]
  - can apply algorithms that handle numerical features like linear regression
- **one hot encoding** → convert discrete features to a series of binary features
  - each group is broken into `0` or `1` depending on what value the old column was
  - useful when there is no ordinal relationship among categories
  - ensures that categorical variables do not imply any numerical relationship
- **normalization**
  - clipping: $\text{clip}(x)$
  - log transform: $\log(1+x)$
  - standard scaler: $\frac{x-\text{mean}(x)}{\text{std}(x)}$
  - max min normalization: $\frac{x-\min(x)}{\max(x)-\min(x)}$

↳ **training & testing**

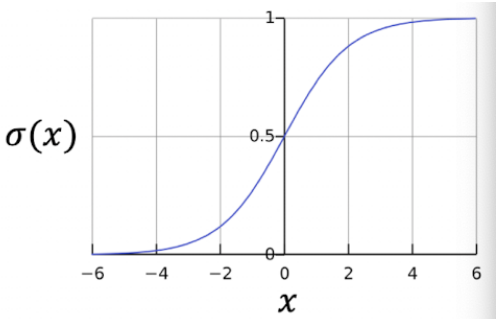
- **logistic regression** → sigmoid function $\sigma(x)$ maps real numbers to a range $(0, 1)$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\lim_{x\to-\infty} \sigma(x) \to \frac{1}{1+e^{\infty}} = 0$$

$$\lim_{x\to\infty} \sigma(x) \to \frac{1}{1+e^{-\infty}} = 1$$

  - prediction: $\hat{y} = \sigma\left(\vec{x}\cdot\vec{w}+\vec{b}\right)$
    - $\vec{x}$ → input
    - $\vec{w}$ → weight
    - $\vec{b}$ → bias



- **training** → fitting parameters of a model by minimizing a loss/cost function
  - cost function J is Cross Entropy Loss
  - uses gradient descent to minimize $J\left(\vec{w},\vec{b}\right)$
    - start at an arbitrary point and move following the steepest downward slope (gradient)
    - continue until convergence
    - stop when improvement in J is below a fixed threshold
    - given n training samples with d features
      - update rule: $\vec{w}_{i+1} = \vec{w}_i - \alpha\Delta J(\vec{w}_i)$
      - weights vector update: $\vec{w}_{i+1} = \vec{w}_i - \alpha\sum_{j=1}^{n}\left[\sigma\left(\vec{x}^{(j)}\cdot\vec{w}_i\right) - y^{(j)}\right]\vec{x}^{(j)}$

↳ **evaluation**

- **accuracy** → (TP + TN) / (TP + FP + FN + TN)
- **precision** → TP / (TP + FP)
- **recall** → TP / (TP + FN)
- **f1 score** → 2 x (Precision x Recall) / (Precision + Recall)
  - harmonic mean of precision and recall
- **errors** → lower the better
  - mean squared error (MSE) → $\sum\frac{(y_i-y_i')^2}{n}$
  - root mean squared error (RMSE) → $\sqrt{\sum\frac{(y_i-y_i')^2}{n}}$
  - mean absolute error (MAE) → $\sum\frac{|y_i-y_i'|}{n}$
- **r squared value** (0 to 1) → higher the better
  - % of variance from the samples

↳ **pipelines**

building complex pipelines out of simple building blocks

- promotes better code reuse
- makes it easier to perform cross validation and hyperparameter tuning
- e.g. tokenizer → hashing transformer → logistic regression
- chains together multiple transformers and estimators to form an ML workflow
  - pipeline is an estimator
- **transformer** → mapping DataFrames to DataFrames
  - e.g. one-hot encoding, tokenization
  - uses `transform()` method
  - outputs new DataFrame which appends the results to the original DataFrame
- **estimator** → algorithm which takes in data, and outputs a fitted model
  - `fit()` returns a transformer

```
from pyspark.ml.classification import
LogisticRegression

training =
spark.read.format("libsvm").load("...")

lr = LogisticRegression(maxIter=10)

lrModel = lr.fit(training)

print("Coefficients: " +
str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
```

▷ **streams**

↳ **motivation**

- data arriving over time, not all at once
- streaming (or online) approaches are designed to process input as it is received
- in contrast to offline or batch approaches that operate on full datasets

↳ **streaming data**

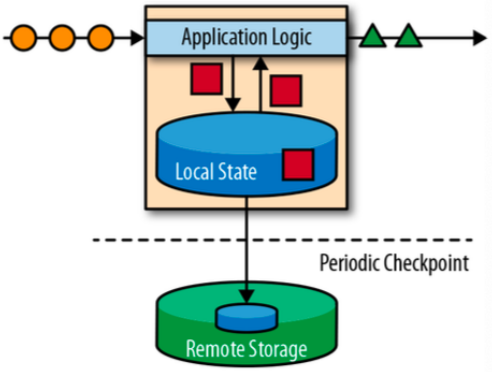
- input elements enter at a rapid rate from input ports (e.g. receiving data from a sensor, from a TCP connection, from a file stream, or from a message queue)
- elements of the stream are referred to as "tuples"
- stream is potentially infinite → system cannot store the entire stream accessibly due to limited memory

↳ **examples**

- sensor data
- online user activity data
- financial transactions
- stock trades data

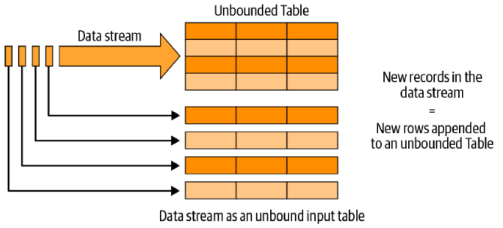
↳ **stateful stream processing**

- ability to record-at-a-time transformations
- ability to store and access intermediate data
- state can be stored and accessed in many different places including program variables, local files, or embedded or external databases

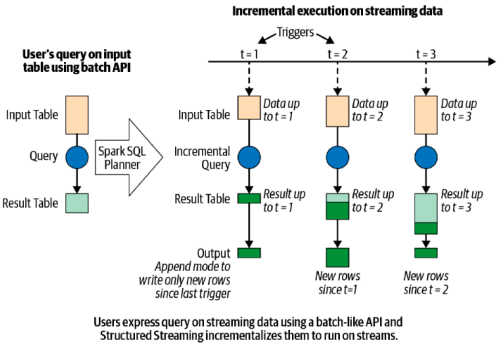


↳ **structured streaming**

- writing stream processing pipelines should be as easy as writing batch pipelines
  - single unified programming model and interface for both types of processing
- treat data as unbounded table



- processing model → incremental execution on streaming data



↳ defining a streaming query

1. define input sources
2. transform data
3. define output sink and output mode
   - output writing details
   - where and how to write the output
4. specify processing details
   - triggering details → when to trigger the results and processing of newly available streaming data
   - checkpoint location → store the streaming query process info for failure recovery
5. start the query

↳ micro-batch stream processing

- Spark Structured Streaming uses micro-batch processing model
  - ‣ divides data from the input stream into micro batches
  - ‣ each batch is processed in the Spark cluster in a distributed manner
  - ‣ small deterministic tasks generate the output in micro-batches
- advantages
  1. quickly and efficiently recover from failures
  2. deterministic nature ensures end-to-end exactly-once processing guarantees
- disadvantages
  1. latencies of a few seconds

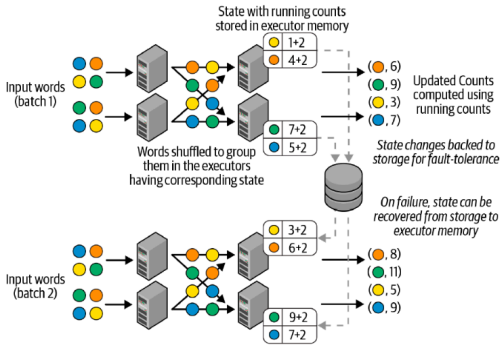
↳ Spark Streaming

```
spark.conf.set("spark.sql.shuffle.partitions", 5)

static = spark.read.json("...")
dataSchema = static.schema

streaming = spark.readStream.schema(dataSchema) \
  .option("maxFilesPerTrigger", 1)\
  .json("...")
```

↳ data transformation

- **stateless transformation** → process each row individually without needing any information from previous rows
  - ‣ projection operations → select(), explode(), map(), flatMap()
  - ‣ selection operations → filter(), where()
- **stateful transformation** → in every micro-batch, the incremental plan adds to the previous value generated from the previous micro-batch
  - ‣ partial state communicated between plans is the state
  - ‣ state maintained in memory of Spark executors and is check pointed to the configured location to tolerate failures
  - ‣ e.g. `DataFrame.groupBy().count()`

↳ stateful streaming



distributed state management in structured streaming

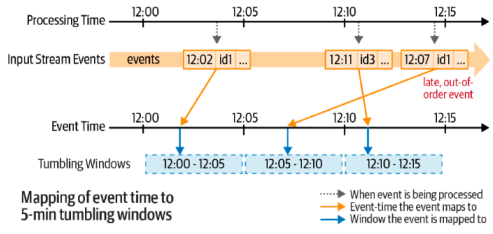
↳ time semantics

- processing time → time of stream processing machine
- event time → time an event actually happened
  - ‣ decouples processing speed from results
  - ‣ operations based on event time are predictable and their results are deterministic
  - ‣ event time window computation will yield the same result no matter how fast the stream is processed or when the events arrive at the operator

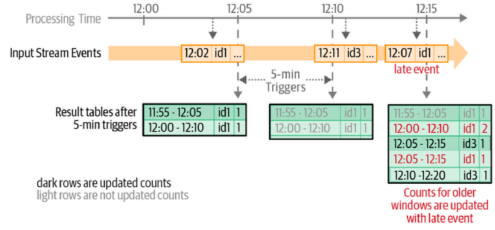
↝ aggregations

- not based on time
- global aggregations: `runningCount = sensorReadings.groupBy().count()`
- grouped aggregations: `baselineValues = sensorReadings.groupBy("sensorId").mean("value")`
- all built-in aggregation functions in DataFrames are supported
- supports aggregations over event time windows

```
sensorReadings.groupBy("sensorId",
window("eventTime", "5 minute")) \
  .count()
```



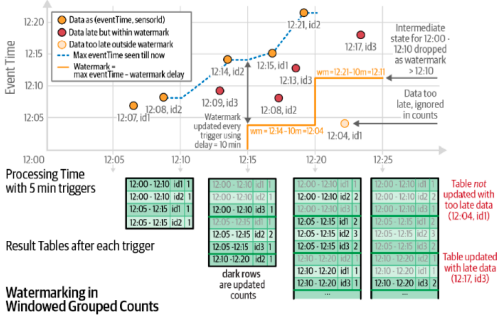
- supports updates in intervals of processing time



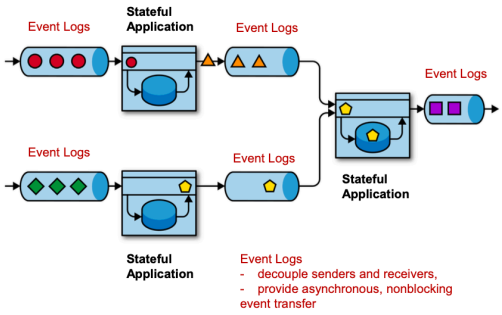
- use watermarks to handle late data

```
sensorReadings \
  .withWatermark("eventTime", "10 minutes") \
  .groupBy("sensorId", window("eventTime", "10
minutes", "5 minutes")) \
  .count()
```

  - ‣ watermarks help Spark understand the processing progress based on event time, when to produce windowed aggregates, and when to trim the aggregations state
  - ‣ watermarks define a threshold for how late data is allowed to arrive, and trims old state associated with data older than the watermark
  - ‣ suppose an event comes in very late, it should not be relevant to the existing data, so it should be ignored

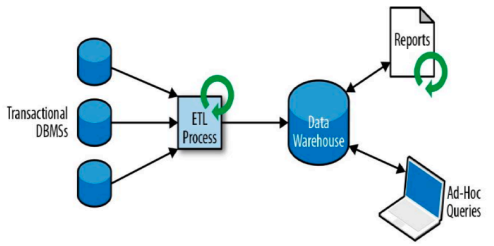


▷ event-driven streaming application



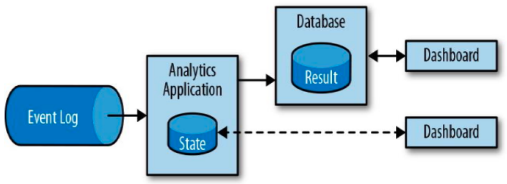
- **event logs** (or message queues) → decouple senders and receivers and provide asynchronous, nonblocking event transfer
  - ‣ most popular solution → Kafka uses message queues to orchestrate producer and consumer action
- events can contain some offset/sequence number where the information must be processed in some order
- store streaming data in a sequence and identify different messages using sequence numbers
  - ‣ consumers read messages one at a time
- events are not purged so they can re-read
- stateful applications can be independent of inputs and outputs

▷ traditional data warehouse architecture for data analytics



- consist of several individual components such as an ETL process and a storage system

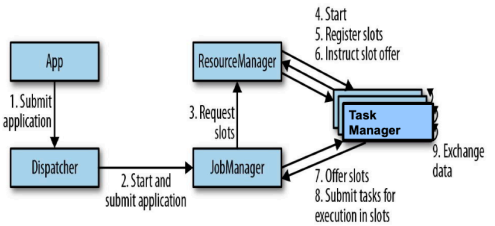
▷ streaming analytics application



- stream processor handles all processing steps
  - ‣ event ingestion
  - ‣ continuous computation
    - – state maintenance
    - – updating results
- sorter time needed for an event to be incorporated into an analytic result
- Flink is the analytics application

▷ Flink

distributed system for stateful parallel data stream processing



- JobManager is similar to Master node in Spark

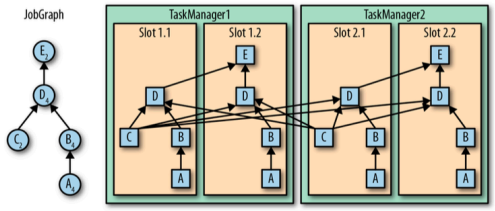
↳ dataflow model

used by Flink

- **dataflow graph** → logical flow of data through a process or a system
  - ‣ nodes → operators
  - ‣ edges → data dependencies
  - ‣ converted to physical dataflow plan that uses the clusters of machines
- similar to lineage in Spark, except data is streaming

↳ task execution

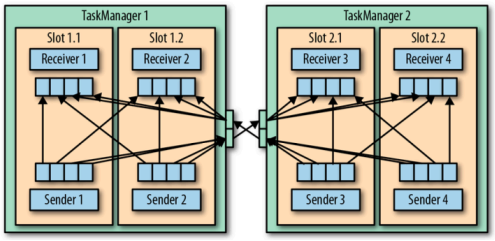
- each machine has a task manager
- task manager executes several tasks at the same time
  - ‣ data parallelism → tasks of the same operator
  - ‣ task parallelism → tasks of different operators
  - ‣ job parallelism → tasks from a different application



- task manager → offers a certain number of processing slots to control the number of tasks it is able to concurrently execute
- processing slot → execute one slice of an application (one parallel task of each operator of the application)

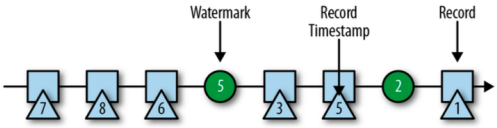
↝ data transfer

- there is network shuffling involved
- tasks of a running application are continuously exchanging data
- task manager takes care of shipping data from sending tasks to receiving tasks
- network component of the task manager collects records in buffers before shipping them

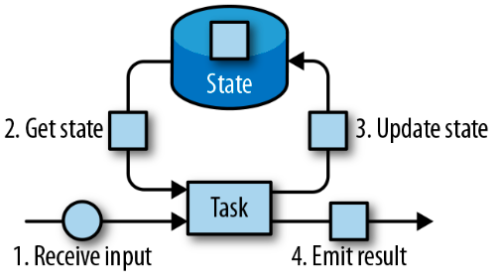


↳ event-time processing

- timestamps → every record must be accompanied by an event timestamp
- watermarks → Flink event-time application must provide watermarks
  - ‣ used to derive the current event time at each task in an event-time application
  - ‣ implemented as special records holding a timestamp as a Long value
  - ‣ flow in a stream of regular records with annotated timestamps
  - ‣ different from that of Spark
  - ‣ user best effort
  - ‣ indicates that all events BEFORE watermark has already arrived
    - – anything that arrives AFTER watermark but has a timestamp before watermark is late
  - ‣ indicates that given a window, the computation can happen already
- lateness horizon → determines if a message is processed if it arrives later
- early triggering is allowed as well



↳ state management



- maintaining information of a task in memory
- state uploaded to remote place for fault tolerance
  - ‣ difficult to implement in Flink

↝ local state management

- task of a stateful operator reads and updates its state for each incoming record
- each parallel task locally maintains its state in memory to ensure fast state accesses

↝ checkpointing

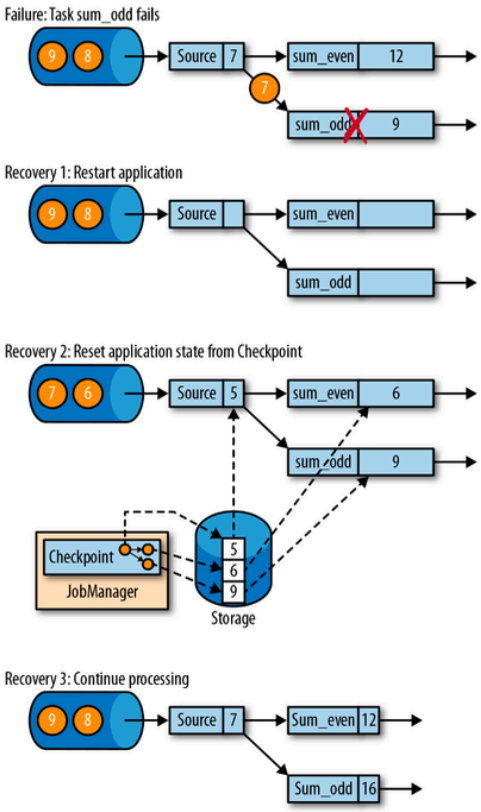
- motivation → task manager process may fail at any point ⇒ storage must be considered volatile
- checkpointing state of a task to a remote and persistent storage
- remote storage → distributed filesystem or database system

consistent checkpoints

- “stop-the-world”
- similar to Spark micro-batch checkpoints

1. pause the ingestion of all input streams
2. wait for all in-flight data to be completely processed ⇒ all tasks have processed all their input data
3. take a checkpoint by copying the state of each task to a remote (persistent storage)
   - checkpoint complete when all tasks have finished their copies
4. resume the ingestion of all streams

- **failure recovery**
  1. restart the whole application
  2. reset the states of all stateful tasks to the latest checkpoint
  3. resume the processing of all tasks



Flink’s checkpointing algorithm

- based on the Chandy-Lamport algorithm for distributed snapshots
- does not pause the complete application but decouples checkpointing from processing
- some tasks continue processing while others persist their state
- uses a special type of record → **checkpoint barrier**
  - ‣ injected by source operators into regular stream of records and cannot overtake or be passed by other records
    - – i.e. if a task is holding a checkpoint barrier, any new incoming inputs will be buffered since those inputs will have to be a part of the checkpoint
    - – if we allow the task to be processed, then there might be a gap between what was checkpointed and what is present
  - ‣ carries a checkpoint ID to identify the checkpoint it belongs to and logically splits a stream into two parts
  - ‣ all state modifications due to records that precede a barrier are included in the barrier’s checkpoint
  - ‣ all modifications due to records that follow the barrier are included in a later checkpoint

1. job manager initiates a checkpoint by sending a message to all sources
2. sources checkpoint their state and emit a checkpoint barrier to all receiving tasks
   - they also acknowledge the completion of the checkpointing to the job manager
3. tasks wait to receive a barrier on each input partition → if 2 input partitions, wait for both to send their barrier
   - records from input streams for which a barrier already arrived are buffered
   - all other records are regularly processed (i.e. those AFTER the current task or those already checkpointed)
4. tasks checkpoint their state once all barriers have been received and they forward the barrier
5. tasks continue regular processing after the checkpoint barrier is forwarded
6. sinks acknowledge the reception of a checkpoint barrier to the job manager
   - a checkpoint is complete when all tasks have acknowledged the successful checkpointing for their state

▷ spark vs flink

- spark
  - ‣ microbatching streaming processing (latency of a few seconds)
  - ‣ checkpoints are done for each microbatch in a synchronous manner (stop the world)
  - ‣ watermark → configuration to determine when to drop the late events
- flink
  - ‣ real-time streaming processing (latency of milliseconds)
  - ‣ checkpoints are done distributedly in an asynchronous manner
    - – more efficient ⇒ lower latency
  - ‣ watermark → special record to determine when to trigger the event-time related results
    - – Flink uses late handling functions to determine when to drop late events

▷ graph data

- modelling relationship between nodes (objects)
- main character → several in degrees

▷ web as a graph

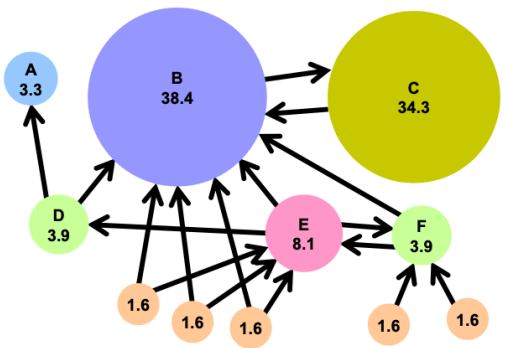
- web as a directed graph
- nodes → webpages
- edges → hyperlinks

▷ PageRank

- ranking pages on the web
- not all web pages are equally “important”
  - ‣ importance → citation by other pages
- measures the importance of pages
- necessary for many web-related tasks (e.g. search, recommendation)
- also used in other applications like bioinformatics

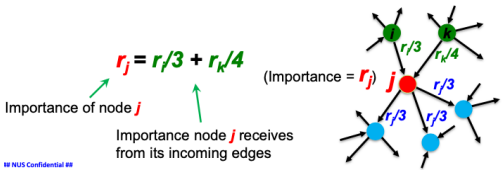
↳ links as votes

- more in-links ⇒ page is more important
  - ‣ assuming incoming links are harder to manipulate
- naive solution → rank each page based on its number of in-links
  - ‣ problem → malicious user can create a huge number of ‘dummy’ web pages to link to their one page to drive up its rank
- solution → make the number of ‘votes’ a page has proportional to its own importance
  - ‣ dummy pages have low importance ⇒ contribute little votes
  - ‣ links from important pages count more



↳ voting formulation

- each link's vote is proportional to its importance of its source page
- for page j, define its importance (or rank) as r_j
- if page j with importance r_j has n out-links, each link gets $\frac{r_j}{n}$ votes
- page j's own importance is the sum of the votes on its in-links
  - each page receives a certain amount of candies from its incoming neighbors
  - distributes these candies evenly to its outgoing neighbors



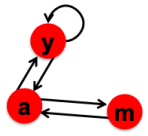
⇝ flow formulation

$$r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

- where d_i is the number of out-links of node i
- solving
  - no unique solution
  - all solutions are rescalings of each other
- introduce additional constraint to force uniqueness → all ranks must equal to 1
  - solved using substitution

matrix representation

- stochastic adjacency matrix M
- let page i have d_i out-links
- $i \to j \Rightarrow M_{ji} = \frac{1}{d_i}$
  - else $M_{ji} = 0$
- M is a column stochastic matrix where columns sum to 1
- rank vector $r \to$ vector with an entry per page
  - $r_i \to$ importance score of page i
  - $\sum_i r_i = 1$
- rewrite flow equation to $r = M \cdot r$



$$r_y = \frac{r_y}{2} + \frac{r_a}{2}$$
$$r_a = \frac{r_y}{2} + r_m$$
$$r_m = \frac{r_a}{2}$$
$$M = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{pmatrix}$$
$$r = \begin{pmatrix} r_y \\ r_a \\ r_m \end{pmatrix}$$

- solved using power iteration

power iteration

- given a web graph of n nodes, where nodes and edges are hyperlinks
- each node starts with equal importance

1. initialize: $r^{(0)} = \left[\frac{1}{n}, ..., \frac{1}{n}\right]^T$
2. iterate: $r^{(t+1)} = M \cdot r^{(t)}$
3. stop when $\left|r^{(t+1)} - r^{(t)}\right|_1 < \varepsilon$ (using L1 normalization)
   - converges
   - ideally, the rank should not change

⇝ random walk formulation

- imagine a random web surfer who at every time step, randomly visits an outlink of current page i with uniform randomness
- let $p(t)$ be the vector whose ith coordinate is the probability that the surface is at page i at time t
  - probability distribution over pages

$$p(0) = \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix}$$
$$p(1) = \begin{pmatrix} \frac{1}{3} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{2} \\ \frac{1}{3} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{2} \\ \frac{1}{3} \times \frac{1}{2} \end{pmatrix}$$

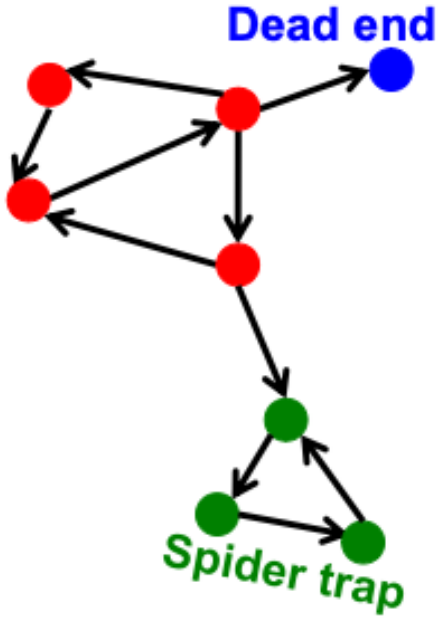
- **stationary distribution** → as $t \to \infty$, probability distribution approaches a steady state representing the long term probability that a random walker is at each node
- $p(t+1) = M \cdot p(t)$
- same as flow formulation

↳ does it always converge?

- not always since they might alternate between each other

↳ does it converge to what we want?

- not always
- problems:
  1. dead ends → no out links
     - random walk has no where to go
     - causes importance to leak out (converge to 0)
  2. spider traps → all out-links within the group
     - random walk gets stuck in a trap
     - cycle
     - eventually trap absorbs all importance (converge to 1)



⇝ teleports

- solution for dead ends and spider traps
- at each time step, the surfer can
  - with probability β, follow a link at random
  - with probability $1-\beta$, jump to some random page (with equal probability for ALL pages)
  - common values of $\beta \in [0.8, 0.9]$
- if at a dead end, always randomly teleport
  - make the dead end m connected to every node (including itself)
- updated page rank algorithm with teleports:

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$$
$$A = \beta M + (1-\beta)\left[\frac{1}{N}\right]_{N \times N}$$

▷ topic specific PageRank

- instead of generic popularity, can we measure popularity within a topic?
- evaluate web pages not just according to their popularity, but by how close they are to a particular topic, e.g. "sports" or "history"
- allows search queries to be answered based on interests of the user
- random walker has a small probability of teleporting at any steps
  - standard PageRank → any page with equal probability
    - avoids dead ends and spider traps
  - topic specific PageRank → topic-specific set of "relevant" pages (teleport set)
- introduce bias in the random walk
  - when the random walker teleports, pick a page from a set S
  - S contains only pages that are relevant to the topic
  - for each teleport set S, get a different vector r_S

$$A_{ij} = \begin{cases} \beta M_{ij} + \frac{1-\beta}{|S|} \text{ if } i \in S \\ \beta M_{ij} + 0 \text{ otherwise} \end{cases}$$

- all pages in S weighted equally
  - assign different weights to pages
- multiply by M and add a vector to maintain sparseness
- number of in-links still dominate the importance rank
- as β decreases, the importance of pages in the topic specific set increases
- as S grows, the importance of each page within the topic specific set decrease (other pages can be teleported to)

↳ discovering topic vector S

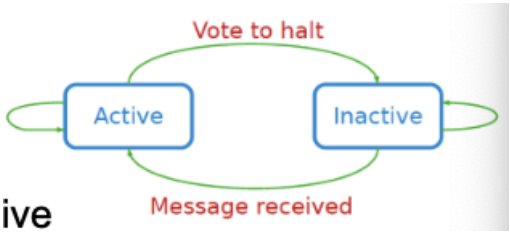
- create different PageRanks for different topics
  - 16 DMOZ top-level categories: arts, business, sports, …
- which topic ranking to use?
  - user can pick from a menu
  - classify query into a topic
  - use the context of the query
    - e.g. query is launched from a web page talking about a known topic
    - history of queries
  - user context, e.g. user's bookmarks

▷ PageRank implementation

- common features of graph algorithms
  - local computations at each vertex
  - pass messages to other vertices
- think like a vertex → algorithms are implemented from the view of a single vertex, performing one iteration based on messages from its neighbors
  - user only implements `compute()` that describes the behavior at one vertex

↳ Pregel

- computational model
- computation consists of a series of supersteps
- closed source by Google, but Meta created open sourced version (Giraph)
- in each superstep, the framework invokes a user-defined function `compute()` for each vertex (in parallel)
  - read message sent to v in superstep $s-1$
  - send messages to other vertices that will be read in superstep $s+1$
  - read or write the value of v and the value of its outgoing edges (or even add or remove edges)
- termination:
  - a vertex can choose to deactivate itself
  - is "woken up" if new messages received
  - computation halts when all vertices are inactive



- e.g. computing maximum over the values in the vertices in this graph
  - each vertex repeatedly sends it current value to its neighbors
  - each vertex updates its value to the maximum over its own value and all messages it receives
  - repeat till all vertex values stop changing
- keep communication within 1-hop neighbors

```
def compute(v, messages):
  changed = False
  for m in messages:
    if v.getValue() < m:
      v.setValue(m)
      changed = True
  if changed:
    for neighbor in v.neighbors:
      sendMessage(w, v.getValue())
  else:
    voteToHalt()
```

↳ architecture

- master & workers architecture
- vertices are hash partitioned (by default) and assigned to workers (edge cut)

- each worker maintains the state of its own portion of the graph in memory
- computations happen in memory
  - happens at each worker machine and on each node
- in each superstep, each worker loops through its vertices and executes `compute()`
- messages from vertices are sent, either to vertices on the same worker or to vertices on different workers (buffered locally and sent as a batch to reduce network traffic)
  - done through network shuffling
  - some vertices need to send information to other worker
  - expensive to send information

↳ **implementation**

- fault tolerance
  - checkpointing to persistent storage – after each superstep
  - failure detection through heartbeats
  - corrupt workers are reassigned and reloaded from checkpoints

↳ **PageRank**

1. invoke `compute()`
2. each vertex, aggregate messages from neighbors to compute PageRank update
3. send message to outgoing neighbors

```
class PageRankVertex : public Vertex<double,
void, double> {
public:
  virtual void Compute(MessageIterator* messages)
{
    if (superstep() >= 1) {
      double sum = 0;
      for (; !messages->done(); messages->next())
{
        sum += messages->value();
      }
      *MutableValue() = 0.15 / NumVertices() +
0.85 * sum;
    }

    if (superstep() < 30) {
      const int64 n =
GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    } else {
      VoteToHalt();
    }
  }
}
```

▷ **PageRank with Spark**

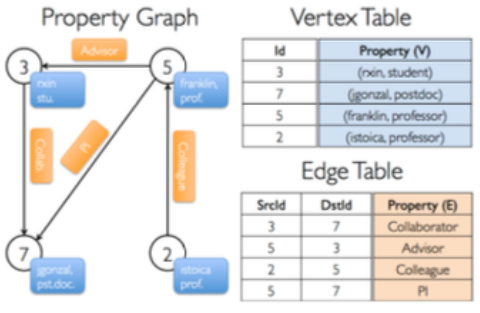
- Spark is more efficient in iterative computation than Hadoop
- integration of record-oriented and graph-oriented processing
- GraphX → extends RDDs to Resilient Distributed Property Graphs
- property graphs
  - present different views of the graph (vertices, edges, triplets)
  - support map-like operations
  - support distributed Pregel-like aggregations
- Graphframe → extends GraphX to provide a DataFrame API and support for Spark's different language bindings
- given the page rank algorithm above

1. join vertices and edges table to form triplets

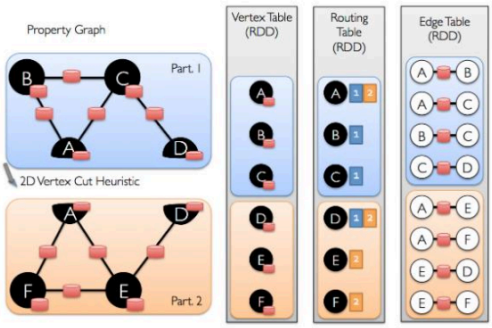
```
CREATE VIEW triplet AS
SELECT s.Id, d.Id, s.P, e.P, d.P
FROM edges as e
JOIN vertices AS s JOIN vertices AS d
ON e.srcId = s.Id AND e.dstId = d.Id
```

2. group by neighbors and perform computation

```
SELECT t.dstId, 0.15 + 0.85 * SUM(t.srcP *
t.eP)
FROM triplets AS t GROUP BY t.dstId
```



- performs vertex cutting to partition the graph → nodes are duplicated across partitions



▷ **comparing graph processing projects**

- Pregel → Google
- Giraph → Facebook open-source implementation of Pregel
- Spark GraphX/GraphFrame
  - achieve performance parity compared with specialized graph systems
  - outperformed by Giraph and Pregel
- Neo4j → graph database + graph processing