Divide:

Sharding: split documents into different servers.

Conquer: Each server process a "soln"

Combine: merge all "solns"

Issues

* fit in main memory
* How to split
* Server crash (re-do all work)

MapReduce

map function: like "counting words"

reduce function: combining local into a global

MapReduce invented by Google.

↳ Simplified programming framework for big data analytics,
users must only specify the computing logic.
(map(), reduce(), framework handles the rest)

Defn: MapReduce: is a data-parallel programming framework (or model) for clusters of commodity machines.

Goals: Scalability of large volumes, cost-efficiency.

Map() → Transform input data into some intermediate (local) results. (word Count)

Reduce → Aggregate or combine the local results into global results.

Parameters must be flexible and powerful

MapReduce: uses lists & key-value pairs as its main data primitives.

- Map $(k_1, v_1) \rightarrow$ List (k_2, v_2)
- Reduce $(k_2,$ list$(v_2)) \rightarrow (k_3, v_3)$

Input has to be a list of key-value pair.

↳ list of documents. (can be multiple pairs for many docs)

< doc 1, "Hello world ..." >

Input list of key-value pairs is partitioned into different servers and each key value, $<k_1, v_1>$ is processed by calling the map function → convert to list $<k_2, v_2>$

i.e → <Hello,1>, <world,1>, ...

All pairs that share the same key will be grouped together

<Hello,1>, <Hello,1> → <Hello, list(1,1)> } <k2, list(v2)>
singles → <world, list(1)> }

Reduce function converts each <k2, list(v2)> → <k3, v3>

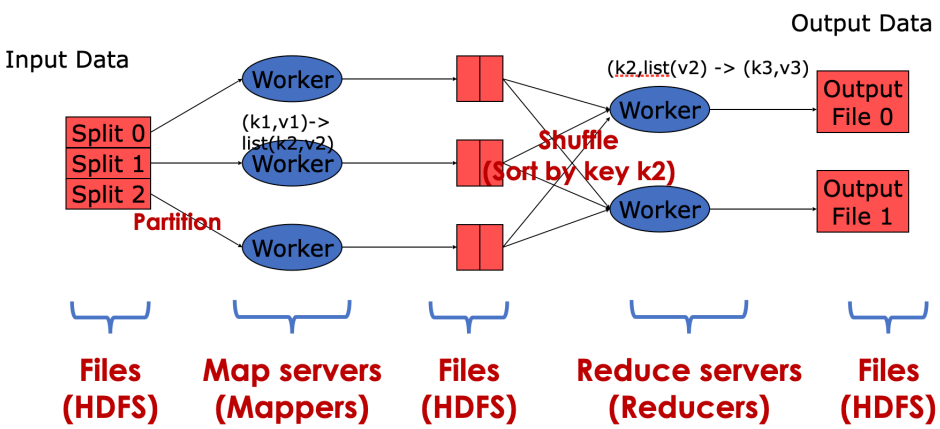
<Hello, list(1,1)> → <Hello, 2> } <k3, v3>
<world, 1>

In general:

Map: Extract something you care about from each record.

Reduce: aggregate, summarize, filter or transform.

Architecture



Details

* Data represented in key value pairs
* Data is chunked (64 MB) based on an input split
* Multiple servers to run Map and Reduce
* Mappers read a chunk of data
* Mappers emit (write out) a set of data
* Intermediate data: (output of mappers) is sorted by key and split to a number of reducers.
* Reducers receive each key of data, along with All values associated with it. (each key must be sent to same reducer)
* Reducers emit a set of data. (reduced from its input, written to disk)

Worker Failure:

Detect failure via periodic heartbeats from master node.

Re-execute in-progress map/reduce tasks

Master failure

- Single-point of failure; Resume from Execution log.

If a node fails:

- query is restarted
- Bad for long jobs (10hrs...)

Recovery in the face of partial failure is an important contribution of MapReduce.

Slow-workers lengthen completion time.
(other jobs, bad disks...)

Soln: spawn backup copies of tasks, whichever one finishes first "wins"

MR vs DB

Failure → restart required

High scalability

No schemas

No query language

flexible UDFs

Optimizations, but limited options

MR yields new data

Can lose machines and keep going

MR → good for large-scale data
hard to write efficient code.

MR mindset: Every task has to be represented using map() and reduce()

Soln: Expose SQL interface and hide MR programming

SQL Queries → Hive → Hadoop

Hive: SQL DB on top of Hadoop.
users only write SQL queries → automatically converted to map reduce.

Hive Terminology

DBs: contains tables, views, partitions
Each table has corresponding directory in HDFS

Tables: - Data: Records (tuples) w/ same schema

Prim Types: TinyInt (1 byte), SMALLINT (2 bytes), INT (4 bytes
BigInt (8 bytes)

Complex types: map<key-type, value-type>

List<element-type>

struct<file-name, field-type, ...>

Hive Query Language

HQL ⊆ SQL + some extensions

↳ Support basic SQL statements

select, project, join, group by, aggregation, create table

Hive Join: Only support equality join

```
SELECT pageid, age, count(*)
FROM pv_users
GROUP BY pageid, age;
```

pv_users

page id	age
1	25
2	25
1	32
2	25

pageid_age_sum

page id	age	Count
1	25	1
2	25	2
1	32	1

```
SELECT pageid, age, count(*)
FROM pv_users
GROUP BY pageid, age;
```

pv_users

page id	age
1	25
2	25
1	32
2	25

pageid_age_sum

page id	age	Count
1	25	1
2	25	2
1	32	1

Hive cannot insert data is designed for data analytics.

Data is generated from outside of Hive

Pros: better concurrency control

Cons: hard for optimize, cannot ensure best storage layout

In DB's they figure out the best storage layout for fast query.

Hive Architecture

Metastore: component that store the system catalog and meta data about tables, cols, parts, ... stored on RDBMS.

Driver: component that manages the lifecycle of a HiveSQL statement as it moves through the hive. Also, maintains a session handle and any session statistics.

Query compiler: component that compiles HiveSQL into a directed acyclic graph of map/reduce tasks.

Optimizer: chain of transformations such that the operator DAG resulting from one trans. is passed as input to the next trans.
(col. pruning, part. pruning, repartitioning of data)

Execution Engine: component that executes the tasks produced by the compiler in proper dependency order. The execution engine interacts w/ the underlying Hadoop instance.

Thrift server: component that provides a thrift interface and a JDBC/ODBC server and provides a way of integrating Hive with other apps.

Client components: command line, Interface (CLI)
web UI, JDBC/ODBC driver

software that let programs talk to the DB.

Some Techniques We Know and Love Are not Directly Applicable

- Indexing
- Zone-maps
- Co-located joins
- Query rewrites
- Cost-based optimization

- Databases own their storage SQL-on-Hadoop systems do not
  - Metadata management is tricky
  - Data inserted/loaded without SQL system knowledge
  - No co-location of related tables
  - HDFS is for most practical purposes, read-only

Hadoop Distributed File system (HDFS)

* storage for Hadoop MR

Server failures, huge files, append-only workload.

Apps

Web crawls: "find all pages that link to given page"

spam pages for training.

File System Interface

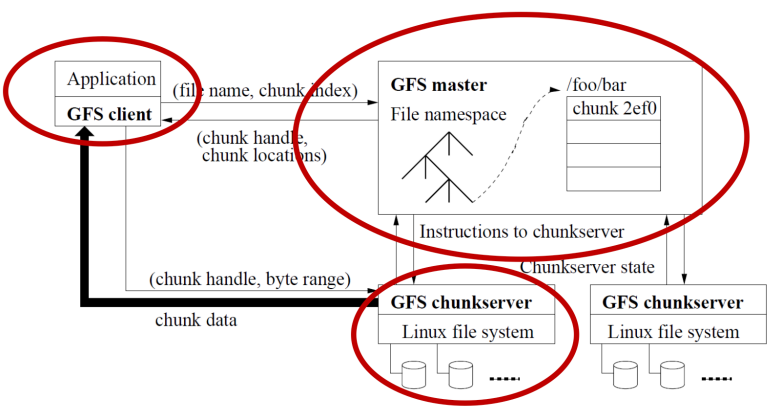
Standard file operations:

* create/delete
* open/close
* read/write

Additional

* snapshot: create a copy of a file or a directory tree at a low cost, by copy and write.
* record-append: append data to the end of the file.

Architecture



GFS Cluster

* single master and multiple chunk servers
* accessed by multiple clients
  - (different from users/applications)
  - users access clients to perform file system operation

GFS Client

* server that interacts w/ user apps.
* provide read/write APIs
* access file → go through GFS client
* Interact with GFS master for metadata and then access chunkservers to get actual data. (does not cache data)

GFS Master

* store the metadata and control the entire GFS cluster.
* only 1 master server in the cluster

monitoring: periodic heartbeat messages to each chunkserver.

* Clients communicate with master for metadata
* data request go to chunkservers.
* Perform: garbage collection, orphaned chunks, chunk migration

GFS Chunkserver

- files stored on chunkservers
- files divided into fixed-size chunks.
- identified by unique chunk handle (assigned by master)
- Read/write based on handle and byte range.
- Chunks replicated for reliability (common factor is 3)

Chunk Size

- parameter, set to a large value (64 MB)
- Chunks stored as plain Linux files.

Pros:

- reduce interaction between client/master
- reduce network overhead by using TCP connection to do many operations in one chunk.
- reduce size of Metadata on master.

Cons:

- small files consist of one or a few chunks (slow read)
- risk of hot spots (popular executable file)
  - ↳ Soln: increase replication factor for small files to distribute the load to more servers.

File Read

Read example:

- GFS client translates file & byte offset → chunk index (fixed chunk size)
- GFS master responds w/ chunk handle & locations
- GFS client sends read requests to chunk servers. (closest)

Must separate data flow vs control flow

- best utilize network bandwith
- o.w, master bottleneck if all data access through GFS master.

Fault Tolerance

- Chunkserver
  - Every chunk is replicated in 3 chunkservers
- Master
  - Single point of failure
  - Write-ahead logging to disk (called operation logs)
  - Can also replicate operation logs to multiple servers
- Data integrity: what if a chunk is corrupted?
  - Use checksums
  - Chunkservers use checksums to detect data corruption

Goals for chunk placement policy

- max data reliability and availability
- max network bandwith utilization

- Chunk Creation: Selecting a chunkserver
  - place chunks on servers with below-average disk space utilization (goal is to equalize disk utilization)
  - place chunks on servers with low number of recent creations (creation → heavy write traffic in near future)
  - spread chunks across racks

- Place chunks on servers with below-average disk space utilization because our goal is to equalize disk utilization
- place chunks on servers with low number of recent creations, prevents heavy write traffic in near future
- spread chunks across racks

GARBAGE COLLECTION

- GFS does not immediately reclaim physical storage after a file is deleted
- "Lazy" garbage collection mechanism
  - master logs are changed to a "hidden file name"
  - master removes hidden files during regular file system scan (3 day window to undelete)

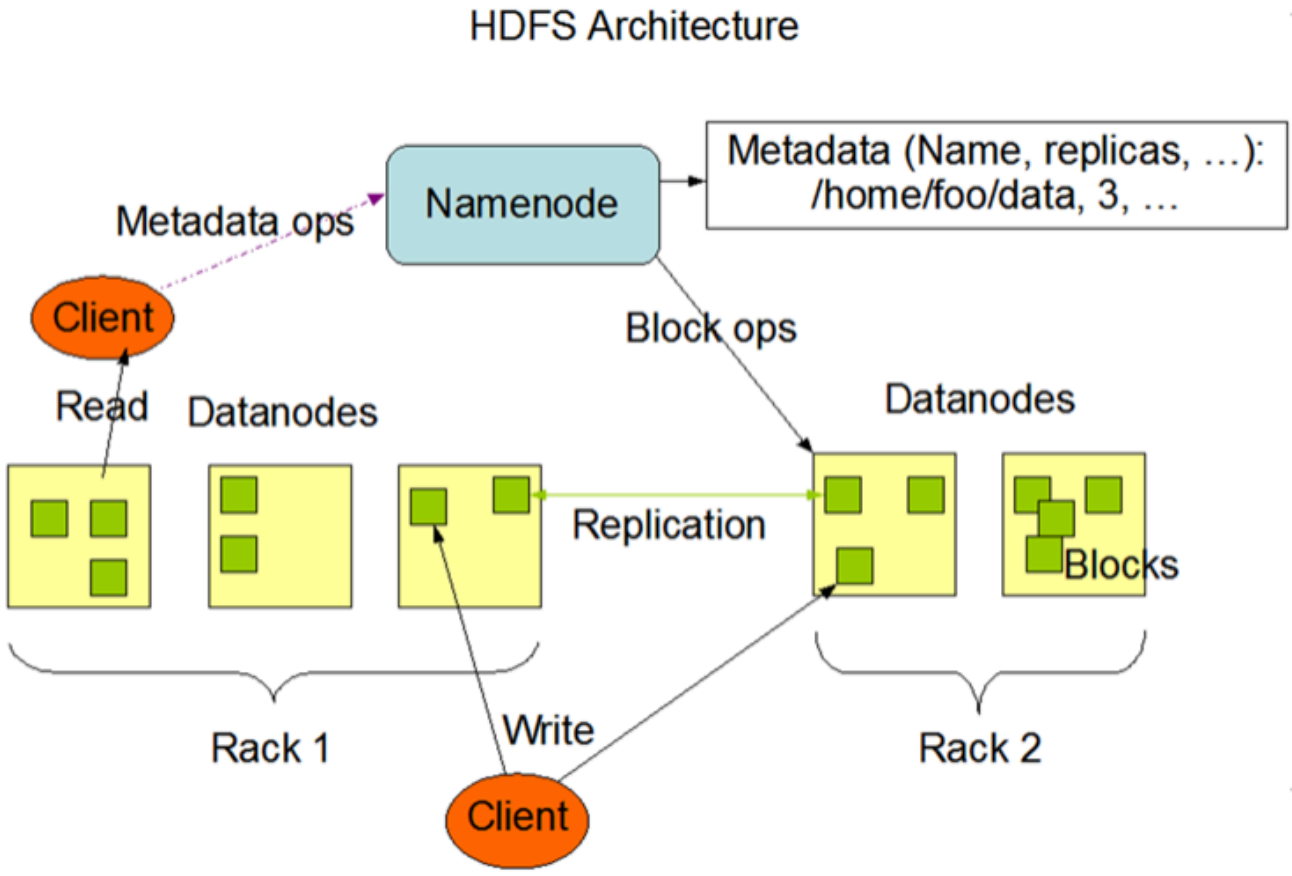
HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

- Open source clone of GFS
  - similar assumptions, design, architecture
    Differences
- no support for random writes, append only
- platform independence (implemented in java)

Terminology

- HDFS -> GFS
- namenode -> master
- datanode -> chunkserver
- block -> chunk

- edit log -> operation log



Bigtable: a distributed storage system for structured data
(HBase is an open-sourced Bigtable)

Issues with HDFS

- inefficient for random accesses (optimized for large-file sequential accesses)
- inefficient fro writers (optimized for read)
- inefficient for supporting structured/semi-structured data, in particular big tables (AKA sparse tables)

Usually when tables have lot of columns there are several empty entries.

Traditional DBs must store empty entries, which is a waste of space.

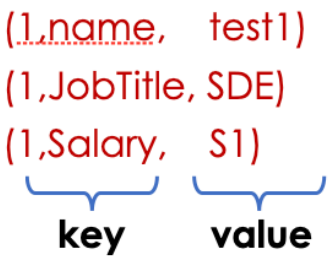
Sparse table is a semi-structured data model where tuples do not exactly follow a fixed schema

Formally, a Bigtable is a sparse, distributed, persistent, multidimensional sorted map

The map is indexed by a row and column key, and a timestamp.
Each value in the map is an uninterpreted array of bytes.

UserID	Name	Age	City	JobTitle	Company	Salary	Address
1	test1			SDE		S1	A1
2		20		Sales	C2		
3			Chicago		C3	S3	A3
4	test4		New York	Consultant		S4	
5	test5	25					A5

- We can use key-value store to represent this table:
  - (Row ID + Column Name ➔ Value)
  - Key is a compound key
  - This can reduce space (only store non-empty cells)



)UE

For new updates we do not update immediately (lazy update)
must append a new tuple of the same key but with a different timestamp for every cell.

- every cell is a collection of pairs with (value, timestamp)
- represents the value at that time
- {(SDE,0),(Senior SDE, 1)}

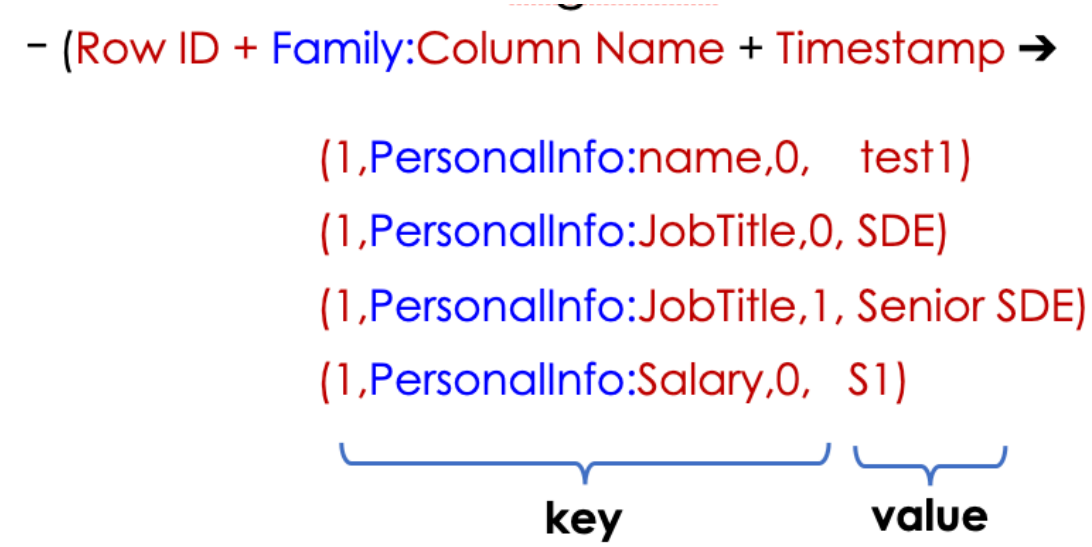
Support timestamp in the key-value model:

- (Row ID + Column Name + Timestamp ➔ Value)
- This can support updates efficiently

- Bigtable organizes the columns into *column families*
  - Easier to manage because there are many columns
  - every column is referenced by family

	PersonalInfo			JobInfo			
UserID	**Name**	**Age**	**City**	**JobTitle**	**Company**	**Salary**	**Address**
1	test1			SDE		S1	A1
2		20		Sales	C2		
3			Chicago		C3	S3	A3
4	test4		New York	Consultant		S4	
5	test5	25					A5

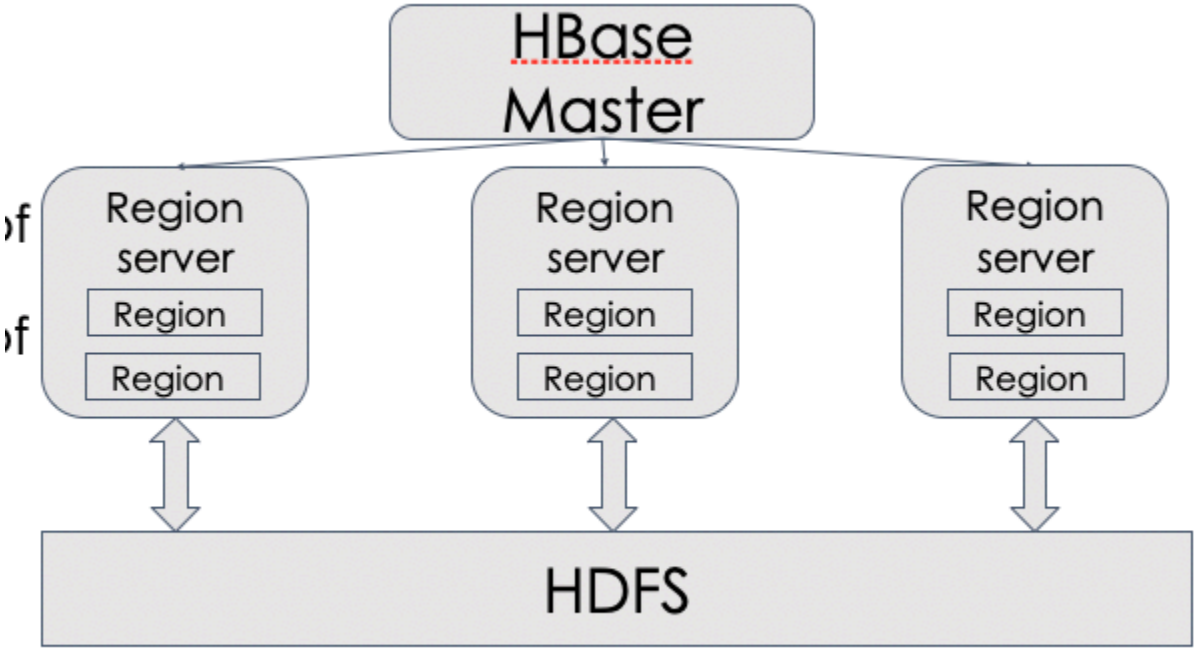
Naturally we need to include column families in the key value and thus:



HBASE STORAGE

- on top of HDFS
- HBase tables are divided horizontally by row key range into regions
- regions are the basic building elements of HBase cluster that consists of the distribution of tables and are comprised of column families
- region server run on HDFS datanode which is present in hadoop cluster

- storage inside each region is based on log-structured merge tree



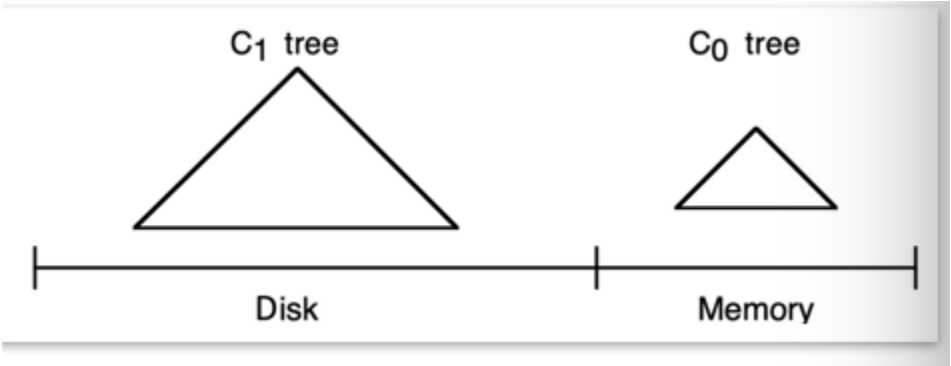
LOG STRUCTURED MERGE TREE (LSM)

- used widely in modern DBs systems
- more than a data structure or storage engine, it is a design principle (okay)
- B-tree insertion incurs many random writes -> LSM converts random writes to sequential writes
- B tree insertion incurs high costs due to in place update -> LSM uses out of place update
  Any static or hard to update structure (vector index) can use LSM

Basic idea of LSM

- if you have existing structure like index table, and new updates come in
- do not update existing structure directly
- instead store new updates in separate structure
- merge the two structures later on
- this is know as out of place update
- it is an immutable index (while B tree is mutable index)

- it has two parts, main memory component (mutable), disk component (immutable)



- Initially C0 and C1 are empty
- when data comes go to C0
- when C0's size exceed a threshold, flush to disk becoming C1
- new data comes again go to C0
- when C0's size exceed a threshold merge with C1
- no random writes to the disk tree C1
- only sequential accesses to C1 with a big chunk
- background compaction
- improve performance for write intensive workloads
- LSM can contain multiple levels
- advantage of LSM: no random writes, no in place update

Limitations of Hadoop MapReduce

- good for one-shot queries when analyzing data (word count, table join, log search) convert to one MR job
- inefficient for iterative queries
  - must have multiple map reduce
  - shows up in many ml task (gradient descent)
  - applications that reuse intermediate results across multiple computations

An *iterative query* includes multiple mr jobs

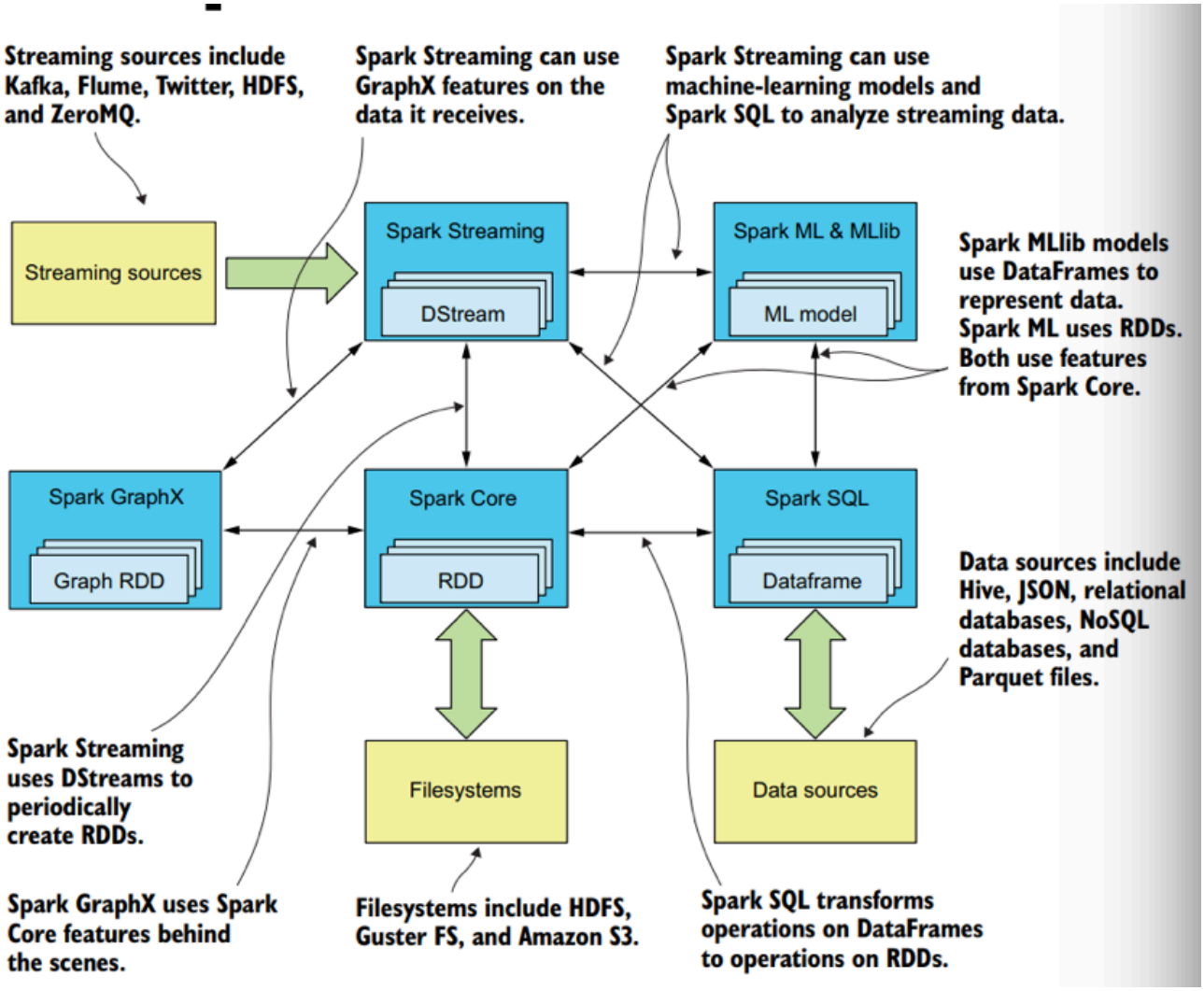
- output of the 1st mr job is the output of the 2nd mr job .....
  Each phase outputs intermediate results in HDFS(on disk), very slow

SPARK

- improved over hadoop
- in memory computing, whenever possible store everything (including intermediate results) in memory instead of disk
- much faster, up to 10 times faster on some iterative workloads, (we care about performance in big data systems)
- has more function, more than simple mr jobs, easier for big data analytics
- written in Scala but can use Java or Python

Spark components

- core
- sql
- graph
- streaming
- ml



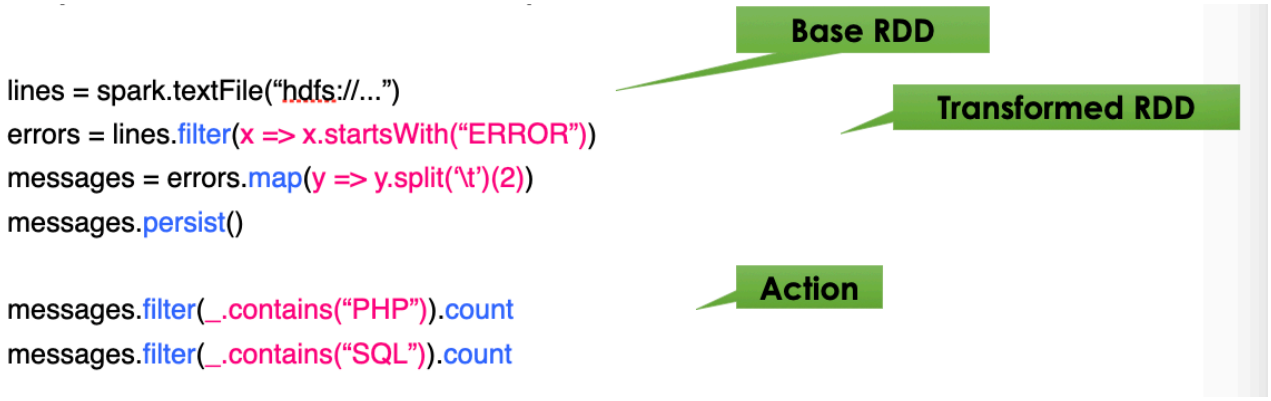
SPARK CORE

- to keep track of different computation stages, spark defines a new concept called Resilient Distributed Datasets (RDD)
- RDD abstracts the data (or objects) transmitted among different computation stages
- RDD is the basic unit of computation and transformation
- RDD is read-only (immutable), partitioned collection of records (think of it like an array or a list but it is a collection of items , a set??)
- RDD can be created from:
  - data in memory or on storage (base RDD)
  - other RDDs (transformed RDD)
    CREATE RDD

```
# SparkContext sc: Spark environment that stores the configuration
# solution 1: from HDFS
# textFile is a build-in method for parsing various types of data files
> val rdd1 = sc.textFile("hdfs://file-path")
# solution 2: from a local file
> val rdd2 = sc.textFile("input/input1.txt")
# solution 3: convert an in-memory array to an RDD (with 3 partitions)
# by default, it's the num of cores in your server
> val rdd3 = sc.parallelize([1,2,3,4,5],3)
> println(rdd3.getNumPartitions)
# solution 4: from another RDD
> val rdd4=rdd3.map(x=>x+10)
```

- All RDDs of a task can form a graph, called *lineage graph*
  - one RDD can be derived from one or more RDDs
  - overall data flow is a graph
- Fault tolerant: can be reconstructed on failure using lineage graph or checkpointed
  - no need for replication
- RDDs are stored in memory can also persist on disk
  - when possible RDDs are stored in memory for fast performance
  - can be reused for multiple computations efficiently (without disk access)
  - can also persist on disk when necessary (insufficient memory)

  Text Search example
- Load error messages from a log into memory, then interactively search for various patterns.



RDD OPERATIONS

- Transformation: transform one RDD to another one
- Action: take some actions on a particular RDD, like count(),...

RDDs provide more functionalities than Hadoop MR
RDD operations are coarse-grained, applied to all items on RDD

Transformation RDD

- map(f: T➔U)
  - RDD[T] ➔ RDD[U]
  - Convert an old RDD to a new RDD by applying the function f to each item in the old RDD

```
data = [1,2,3,4,5]
rdd = sc.parallelize(data).map(x=>x+1)
rdd.foreach(println) # output is [2,3,4,5,6]
```

- flatMap(f: T➔seq[U])
  - RDD[T] ➔ RDD[U]
  - Similar to map(), but it'll flatten the output

```
data = [2,3,4]
rdd1 = sc.parallelize(data)

# range(1,x) will print out values from 1..x-1
# output: [[1], [1, 2], [1, 2, 3]]
rdd1.map(x => range(1,x))

# output: [1, 1, 2, 1, 2, 3]
rdd1.flatMap(x => range(1,x))
```

RDUE

- filter(f: T ➔ Bool)
  - RDD[T] ➔ RDD[T]
  - Convert an old RDD to a new RDD by applying the function f to each item in the old RDD and only showing the qualified items
  - You can think f as a filter

```
data = [1,2,3,4,5]
rdd = sc.parallelize(data).filter(x=>x%2==0)
rdd.foreach(println) # output is [2,4]
```

- reduceByKey(f: (V,V) ➔ V)
  - RDD[(K,V)] ➔ RDD[(K,V)]
  - You can also define a function for more complicated computations

```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
rdd.reduceByKey(add) # output is [('a', 2), ('b', 1)]
rdd.reduceByKey((x,y) => x+y) # output is [('a', 2), ('b', 1)]
```

- join()
  - (RDD[(K, V)],RDD[(K, W)]) => RDD[(K, (V, W))]
  - It merges two RDDs based on the same key

```
rdd1 = sc.parallelize([("a", 1), ("b", 4)])
rdd2 = sc.parallelize([("a", 2), ("a", 3)])
rdd1.join(rdd2) # output is [('a', (1, 2)), ('a', (1, 3))]
```

- union()
  - It merges two RDDs (keeps duplicates if any)

```
rdd1 = sc.parallelize([("a", 1), ("b", 4)])
rdd2 = sc.parallelize([("a", 2), ("a", 3)])
rdd1.union(rdd2) # output is [("a", 1), ("b", 4), ("a", 2), ("a", 3)]
```

Transformations	*map*(f : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]

ACTION RDDS

- action RDD performs actual computation on the input RDD
  - Count(): RDD[T] => Long
  - Collect(): RDD[T] => Seq[T]
  - Reduce(): RDD[T] => T
  - Save(): Outputs RDD to a storage system, e.g., HDFS

- Count()
  - Return the num of items in an RDD

```
sc.parallelize([1,2,3,4,5]).count() # output is 5
```

- Collect()
  - Return the items in an RDD

```
sc.parallelize([1,2,3,4,5]).collect() # output is [1,2,3,4,5]
```

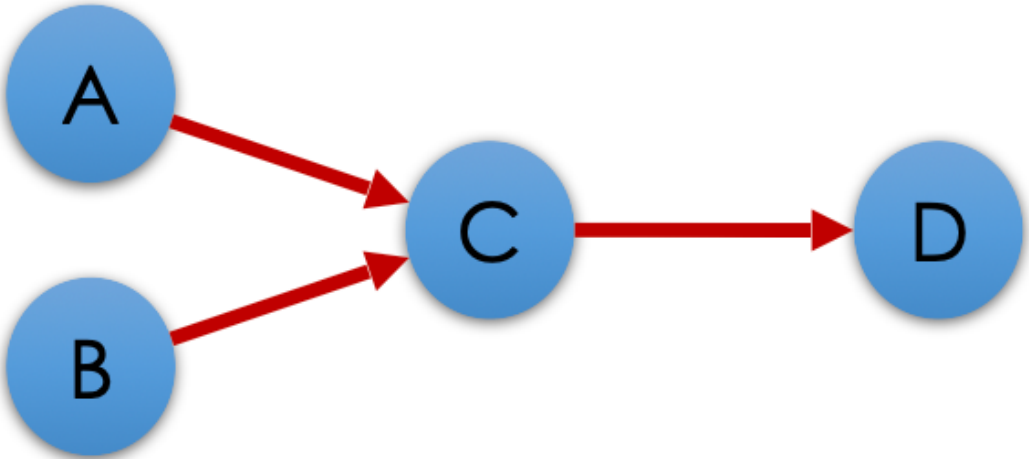
- Reduce(f: (T,T) => T)
  - RDD[T] ➔ T
  - Reduce the items of the input RDD using the function specified
  - Use the function to compute the first two items and produce a new item. Then use the function to compute the new item and the 3rd item and produce another new item…

```
sc.parallelize([1,2,3]).reduce((a,b) => a + b) # output is 6
sc.parallelize([1,2,3]).reduce((a,b) => a min b) # output is 1
sc.parallelize([1,2,3]).reduce((a,b) => a max b) # output is 3
```

SPARK DAG (directed acyclic graph)

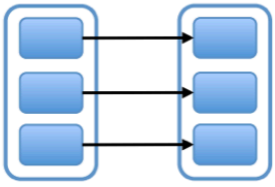
- workflow is represented as a DAG
- DAG tracks dependencies (lineage)
  - nodes are RDDs

- arrows are transformations

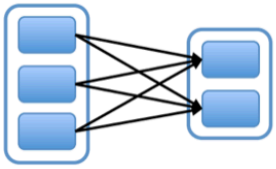


SPARK DEPENDENCY

- Narrow dependency: Parent partition is used by only one child partition
  - Examples: map, filter

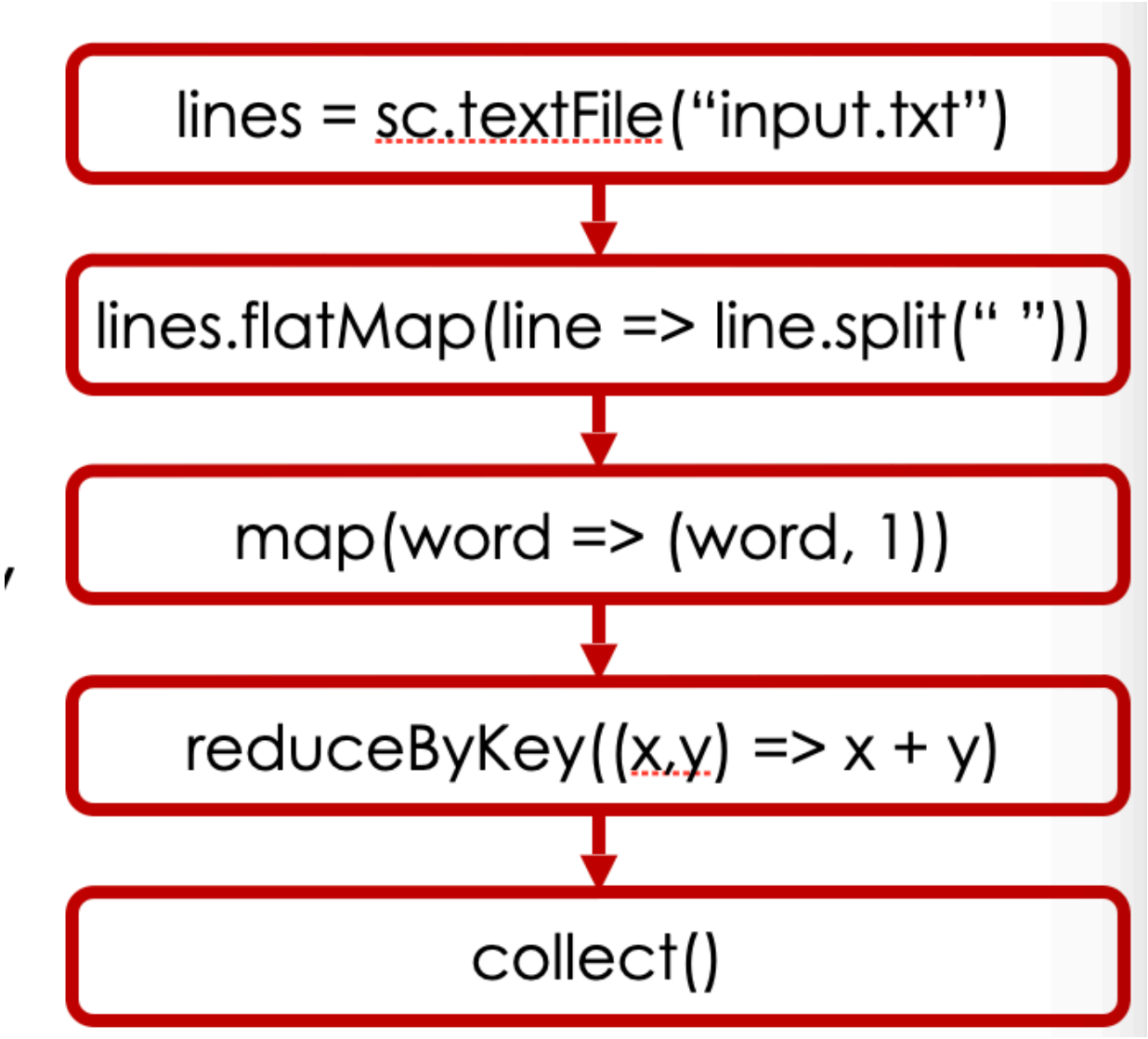


- Wide dependency: Parent partition is used by many child partitions
  - Example: reduceBy



SPARK EXECUTION

- Lazy evaluation
  - data in RDDs is not processed until an action is performed
  - do actual evaluation only when we see action RDDs (only in collect() will trigger actual evolution & computation)



FAULT TOLERANCE

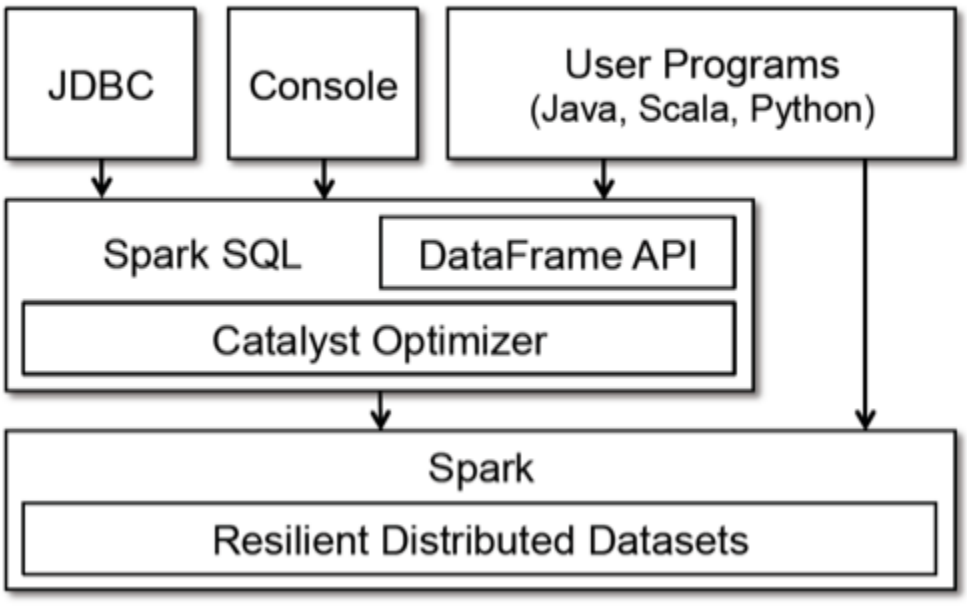
- if a server executing RDD is crashed, we simply reconstruct the RDD from the lineage graph
- For fast recovery, you can persist some intermediate RDDs so that you don't have to rebuild from beginning (checkpointing)

- SparkSQL is used to query data on spark easily
- especially for structured data with schemas
- No need for tedious spark RDDs
- simple SQL queries automatically translated to spark RDDs

GOALS:

- support relational processing both within spark programs and external data sources using friendly API
- high performance using established DBMS techniques
- support new data sources, including semi-structured data and external databases amenable to query federation
- enable graph processing (advanced analytics) and external databases
- enable the use of advanced analytics algorithms, like graph processing and ml

SPARK SQL ARCHITECTURE



DataFrame:

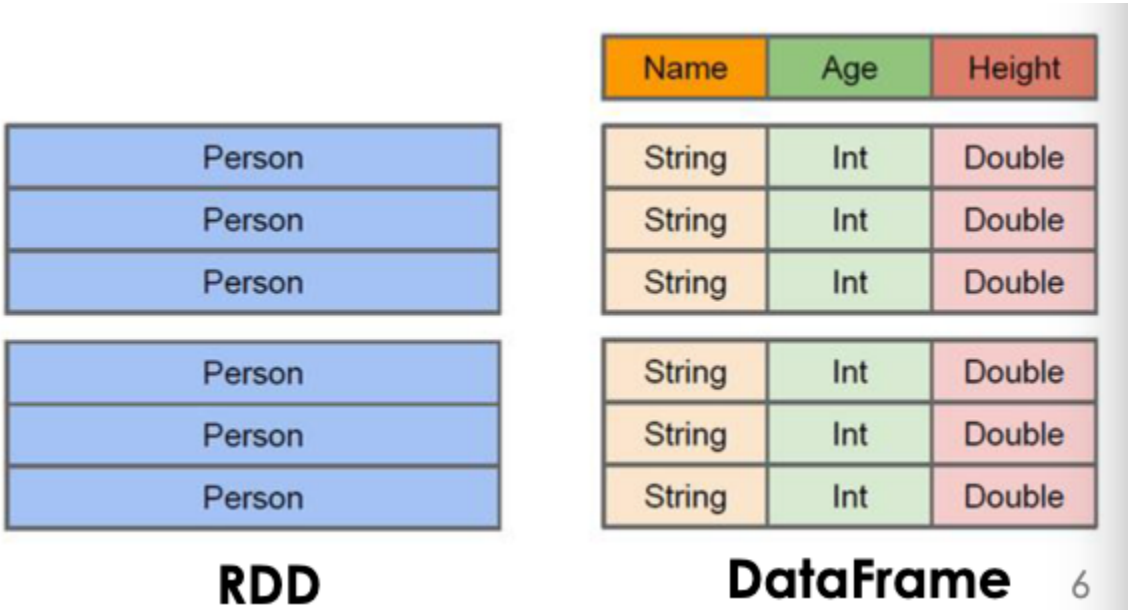
- new concept to abstract RDDs for structured data (like a table)

Types of interfaces:

- DataFrame API
- SQL over DataFrame

DataFrame is designed for handling structured, distributed data in a table-like representation with named columns and declared column types. It is a higher-level abstraction than RDD.

- has schema(types), allows more meaningful operations/queries over columns and rows.
- For RDD we only know that there is a collection of items (not knowing data type of each fiels)



We can create DFs from a csv, json

- Schema will be automatically inferred
- Can specify options("inferSchema","true")

We can print out the dataframe `dfs.printSchema()` note nullable means the column can be null.

Can also create data frame from RDD with/without schema

Two approaches for query DataFrame:

- DataFrame operations
- SQL queries over DataFrame

Operations:

- select one or more columns
- limit(k) to print out first k results
- filter to add some condition like age > 23
- GroupBy and use max(),min(),avg()
- sort(), orderby(), (by default we have ascending order)

- join two dataframes based on common attributes

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

SQL Queries over DataFrame

- DFs can be registered as temporary tables in the system catalog and queried using SQL

```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young'
```

Data Streaming

- Many applications must process large streams of live data and provide results in near real time.
  - IoT data with sensors
  - Social Network trends
  - website statistics
  - monitoring
- We do not know the entire data set in advance
  - Data is generated and ingested continuously
- Think of data as infinite and non-stationary(the distribution changes over time)

Applications

- Mining query streams (which queries are more frequent today than yesterdays)
- Mining click streams (which of its pages are getting unusual number of hits in the past hour)
- Mining social network new feeds (look for trending topics)
- IP packets monitored at a switch (information for optimal routing, detect denial of service attacks)

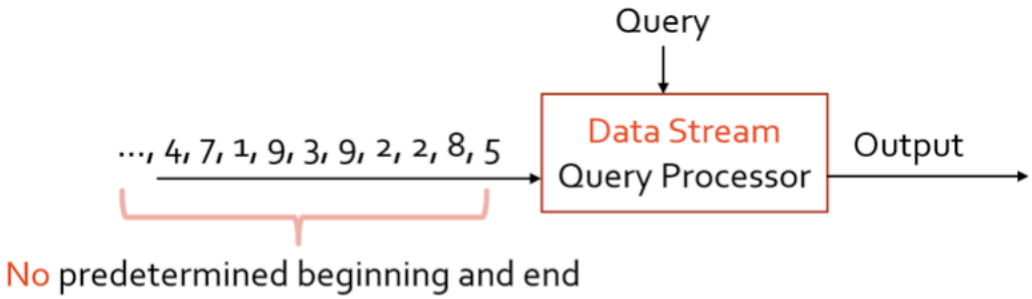
STREAM vs BATCH

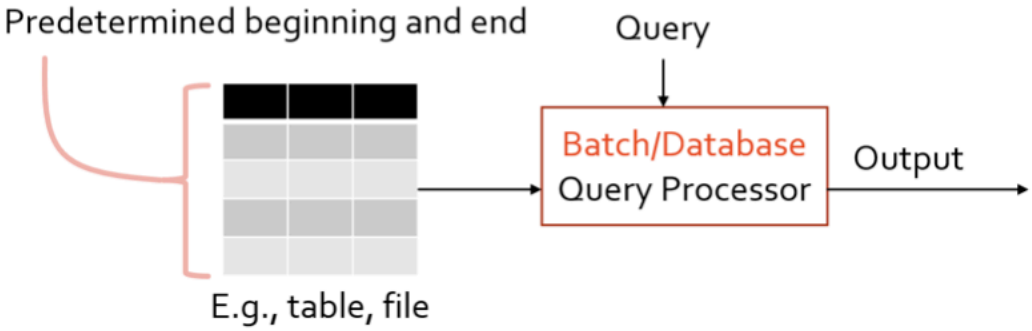
Batch processing:

- see the entire data in advance
- able to store all data

Stream processing:

- don't see the entire data in advance
- can't store all the data





Example streaming K-largest elements:

- Suppose we have a stream (infinite) of integers, how do we find the largest k integers so far?

Constraints:

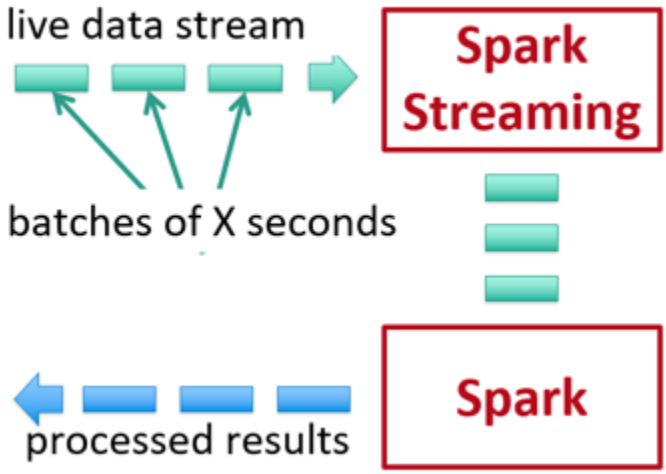
- cannot store all the elements
- can only look once
- use O(k) space

Main idea:

- use a min heap of size k
- top element is the smallest and it represents the kth largest element seen so far
- the heap stores the answer
- scan every element in the stream
- if its smaller than the top, we discard it
- otherwise
  - push e to the heap, will automatically do heap adjustment
  - delete the top, bingo! heap is updated
- time complexity is $nlog(k)$

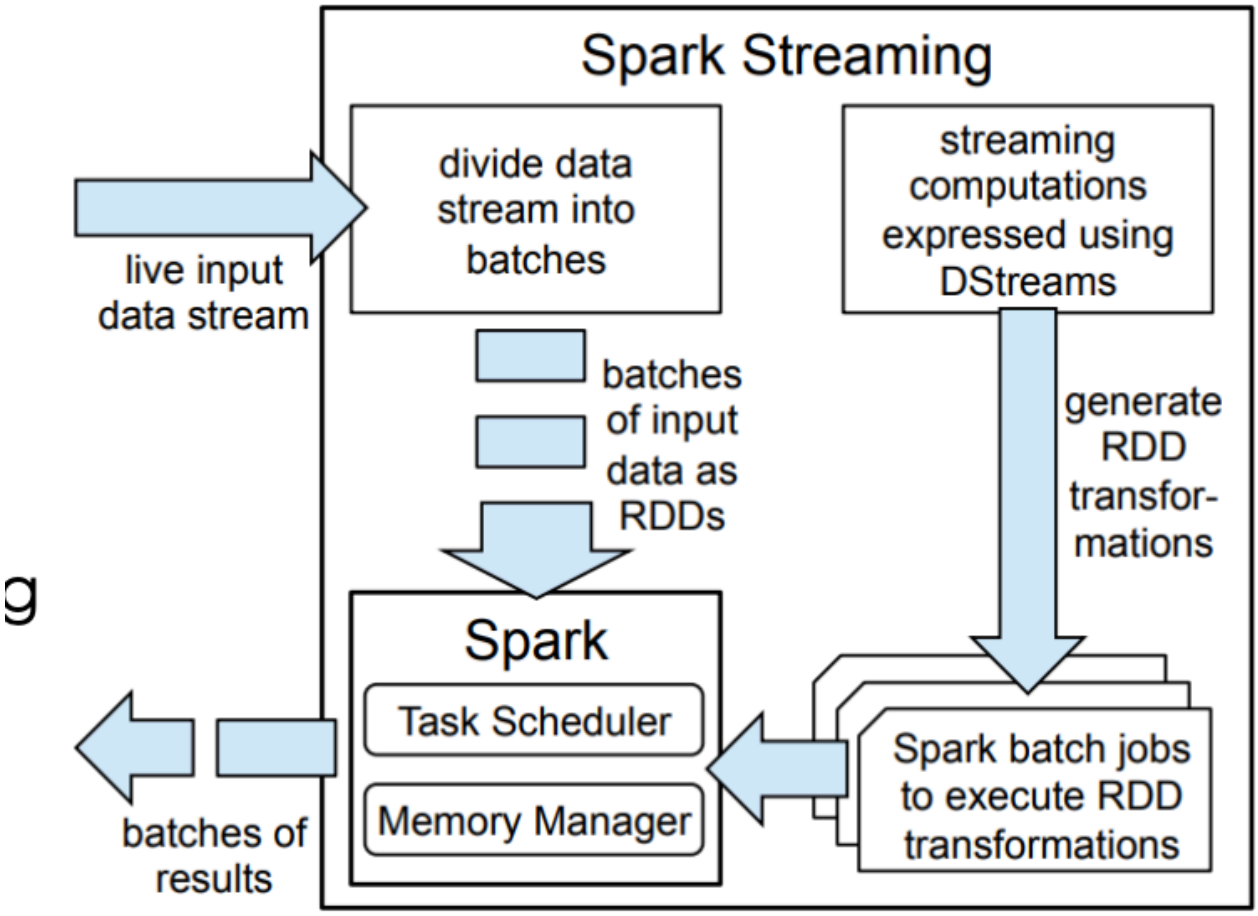
Spark streaming

- spark is a batch processing system
- main idea: discretized stream
  - run a streaming computation as a series of very small stateless deterministic batch jobs
  - chop the incoming data into intervals of X seconds (1 second)
  - run spark within each interval -> batch processing within each interval (using spark RDD)
  - final result: can be returned across different intervals



- Spark streaming divides input data streams into batches and stores them in Spark's memory

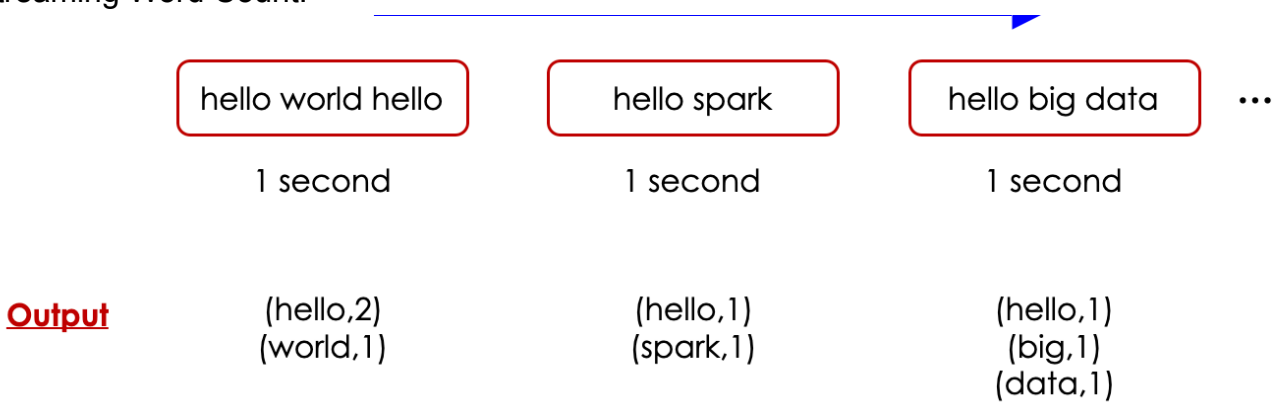
- It then executes a streaming application by generating Spark jobs to process the batches



DStream (discretized streams)

- Sequence of RDDs representing a stream of data
- DStream is a sequence of immutable, partitioned datasets (RDDs) that can be acted on by deterministic transformations.
- These transformations yield new D-Streams and might create an intermediate state in the form of RDDs
  Note there are stateless RDDs (default) and stateful RDDs

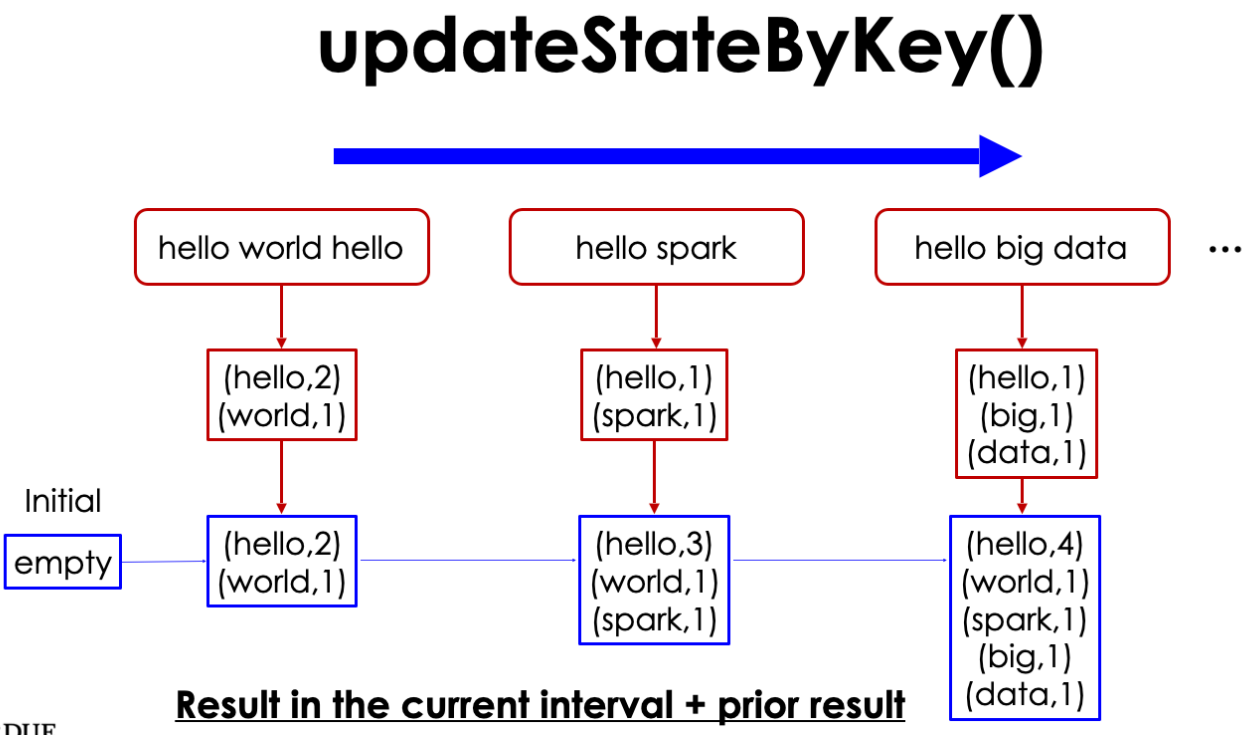
Streaming Word Count:



DStream Operations

- Stateless transformation
  - Only show the results within current time interval
  - word count of the current interval
  - API's that we've learned in spark core: map(), flatMap(), filter(), etc
- Stateful transformation
  - also consider the results of prior intervals
  - word count of the words received so far
  - word count in the last 10 seconds

  Stateful Operations:
- update StateByKey()
  - compute the result based on all the history data received so far
  - need to specify an update function of how to change the status
- window operation
  - compute the results in a specified moving window (last 10 seconds)

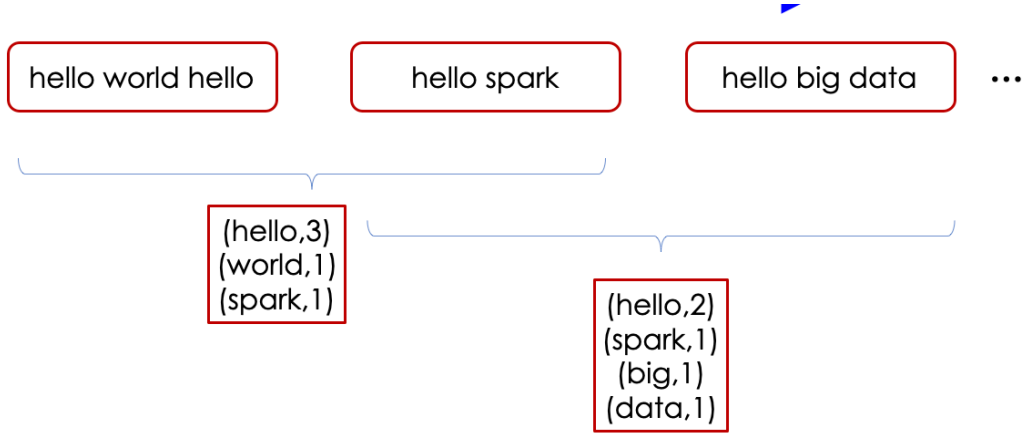


PURDUE

Window Operations:

- compute the result of the last time window
  - window length: multiple units of a time interval
    - if the time interval is 3 seconds, then window size can be 3sec,6sec,9sec, etc
  - sliding interval: time to trigger computation: multiple units of a time interval:

- specify when to compute the results
- it's set to be 3 seconds, then it'll show the results at the end of each time interval



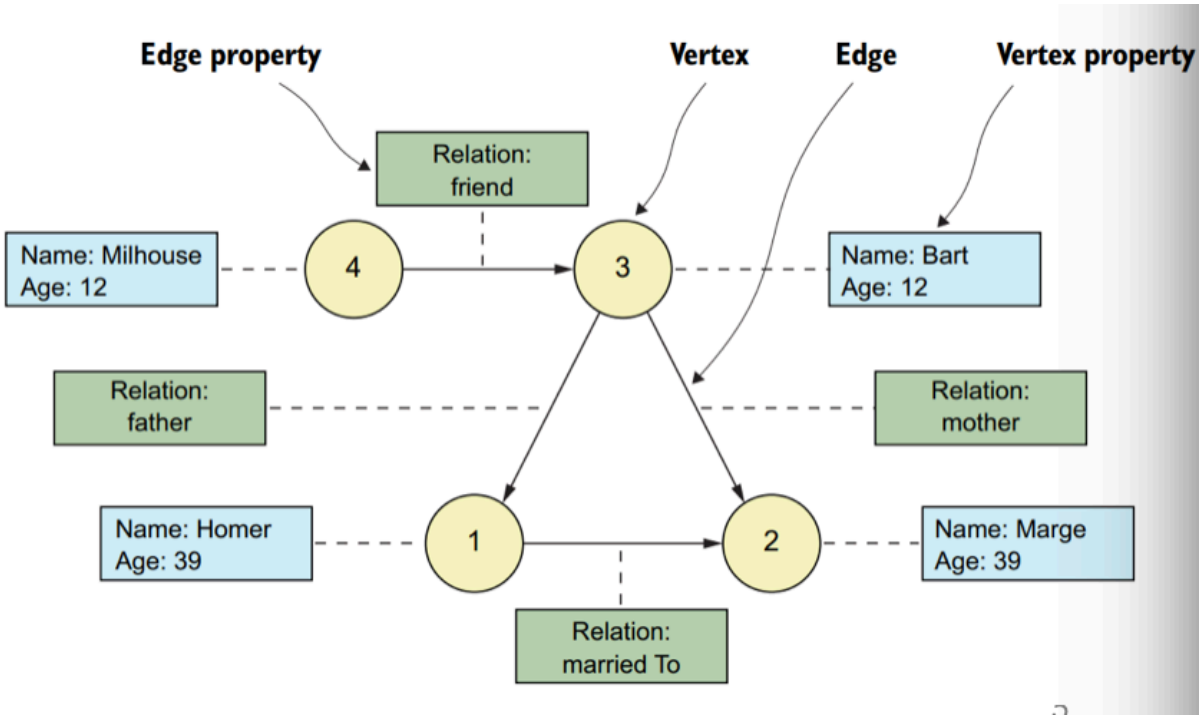
reduceByKeyAndWindow():

- computes the result in the current window instead of the current interval
-

- Graph
  - a collection of vertices and connected edges.
  - storage: adjacency list or adjacency matrix
  - directed and undirected graph
    - directed graph: the order of the two vertices in an edge matters
    - undirected graph: the order doesn't matter

Property Graph

- it is a directed multigraph with user defined objects (or properties) attached to each vertex and edge.
  - note: it's possible to have multiple edges between the same two vertices, because two vertices may have different relationships, friends and coworkers, or multiple flights between two vertices



Graph Computation Model: BSP

BSP: Bulk Synchronous Processing
- a programming model and computation framework for parallel computing
- multiple computing processors, servers of cores
- computation is divided into sequences of supersteps
- each superstep, a set of processors, running the same code, executes concurrently and creates messages that are sent to other processes.
- superstep ends when all the computation in the superstep is complete and all messages have been sent

- a barrier synchronization at the end of the superstep
- the next superstep begins
- until the program terminates(reach mad num of iterations or converges)

- Many graph algorithms are performed in different iterations (pagerank or shortest paths)
- in a superstep (iteration):
  - every node will perform a compute() function based on the neighbor info received (update some info for pagerank or shortest path)
  - the node will sent the new message to its neighbors
- after every node finishes the compute(), the next superstep starts

Computing the max:

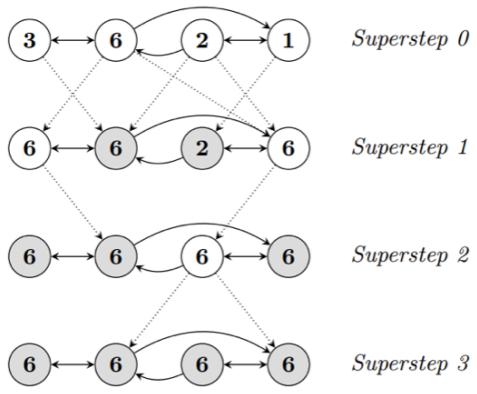
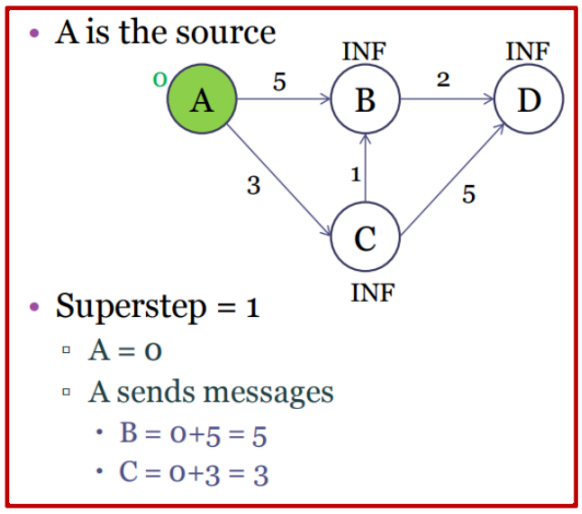


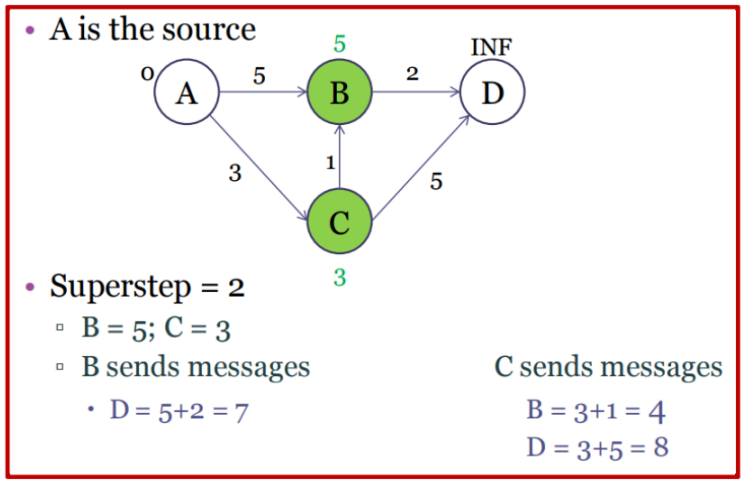
Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.

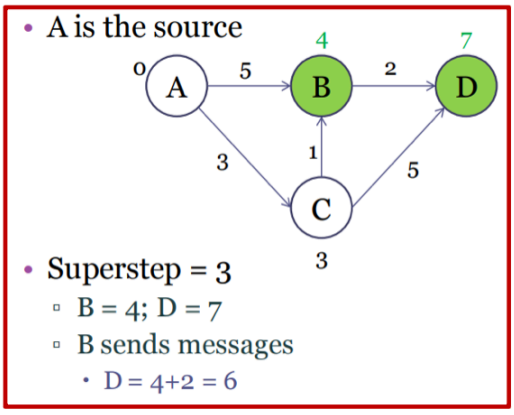
- Superstep 0: initialization
  - Every node has its initial value
  - It'll send its current value to all the neighbors
- Superstep 1
  - Every node will compare its current value with the values received from its neighbors
  - Node A receives 6 from its neighbor B and compares 6 with its old value 3, and then changes to 6
  - If a node doesn't change the value (or status), it'll become inactive and will not send new messages to neighbors
- Superstep 2
  - It a node doesn't receive new messages, it'll become inactive
- The process continues until every node is inactive (or reaches the max iterations)

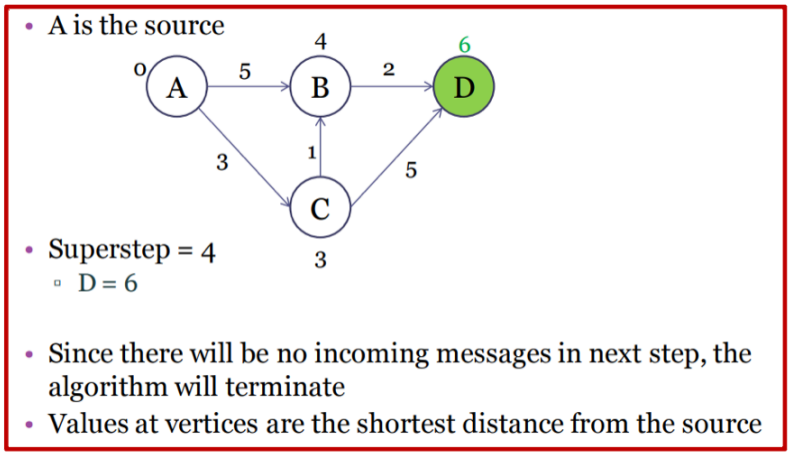
Single Source Shortest Paths:

- Finding shortest path between a single source vertex and every other vertex in the graph.
- Each vertex stores a value denoting the distance from source vertex to this vertex
- Value at each vertex is initialized to INF
- In each superstep
  - receives messages from its neighbors with updated potential minimum distances from source vertex
  - if minimum of these updated values is less than the current minimum distance of the vertex, value is updated and potential updates are sent to the neighbors (current value + outgoing edge weight)
- In first superstep, only source vertex will update its value to zero and send update messages to its neighbors
- algorithm terminates when no more updates









Spark GraphX

- GraphX is a graph processing system built inside Spark
- It relies on RDDs as the building blocks and implements many graphs algorithms for large scale data based on the BSP model

provides APIs

- graph construction
- graph transformation
- graph algorithms

Graph Construction:

- Based on vertexRDD and EdgeRDDs

VertexRDDs: contain tuples, which consist of two elements; a vertex ID of type Long and a property object of an arbitrary type

EdgeRDDs: contain edge objects, which consist of source and destination vertex ID (sourceID and destinationId) and a property object of an arbitrary type(attr, field)

- You can create a Spark graph using VertexRDD and Edge RDD (there are other construction methods)

Shortest Path Algorithm

- Spark implements the shortest-path algorithm with the ShortestPaths object.
- It has only one method, called run, which takes a graph and a sequence of landmark vertex IDs
- The returned graph's vertices contain a map with the shortest path to each of the landmarks, where the landmark vertex ID is the key and the shortest-path length is the value.

Linear regression:

- using a linear function to model the relationship between two variables by fitting a linear equation to observed data.

Notation:
n = number of features
x^(i) = inpute features of ith training example
(x^(i))j = value of features j in ith training example
m = number of examples

- choose the parameters we want to estimate so that the function/model we learn is close to y on all the training examples
- cost function (or loss function)
  - quantify the difference between the estimate value of model and true value

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

easy for derivatives averaged square of differences

Goal: choose θ_0 and θ_1 to minimize the cost function

RDUE

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

easy for derivatives averaged square of differences

Goal: choose θ_0 and θ_1 to minimize the cost function

RDUE

Outline:

- start with (random parameters)
- keep changing parameter to reduce cost function until we end up at a minimum

Gradient descent Algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad (\text{for } j = 0 \text{ and } j = 1)$$

}

learning rate — partial derivative of θ_j → move to the steepest direction

Linear regression with one variable

Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$(\text{for } j = 1 \text{ and } j = 0)$$

}

Linear Regression Model

$$h_\theta(x) = \theta_0 + \theta_1 x$$

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

Hypothesis: $h_\theta(x) = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$ ($x_0 = 1$)

Parameters: $\theta_0, \theta_1, \ldots, \theta_n$

Cost function:

$$J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Gradient descent:

Repeat {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \ldots, \theta_n)$$

}

(simultaneously update for every $j = 0, \ldots, n$)

Large Scale Machine Learning

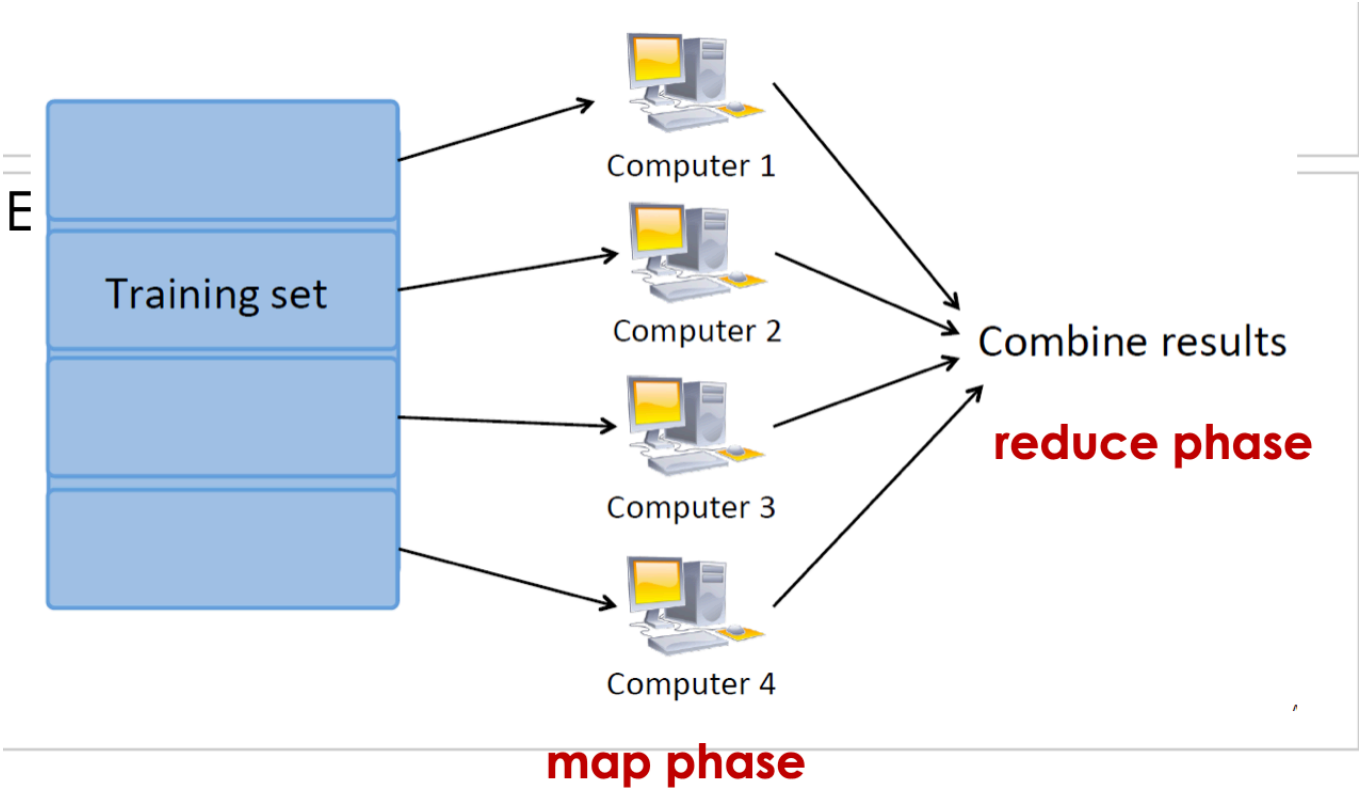
- If data is huge with billions or trillions of training examples, each iteration is very slow

Distributed ML

- Use many machines
- Each machine processes a partition of data
- Each machine computes the gradient on that machine
- Finally, combine all the gradients

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

parallelize this part



Distributed ML with Map-Reduce

- Map
  - workers do computation locally
  - map the training examples to temp
- Reduce
  - compute the sum of all temp

- Server side
  - compute the derivatives
  - Broadcast parameters to all workers
  - continue the next iteration

ML in Spark

- all the advantages of Spark extend to machine learning
  - spark's distributed nature: leverage Spark's RDDs to scale to large-scale data
  - Spark's unifying nature: offer a platform for performing most tasks in man applications
    - can collect, prepare, and analyze the data (various types)
      MLlib (spark, mllib)
- based on Mlbase project in Berkeley
- more mature
- based on RDDs

ML (spark.ml)

- new ml package, still developing
- end-to-end pipeline
- based on dataframe

Data Types in MLlib

- Two building blocks
  - vectors, matrices

For each building block there are:

- local version vs distributed version:
  - local: stored on a single node
  - distributed: based on RDD, stored on multiple nodes
- Dense version vs sparse version
  - sparse: contains a lot of 0's
  - dense: not that many 0's
- Underlying linear algebra operations are provided by Breeze and jblas

Dense vs Sparse Vectors

- Dense vector
  - it can take all elements as inline arguments or
  - it can take an array of elements

- Sparse vector
  - Need to specify a vector size, an array with indices (of non-zero values), and an array with values

Matrix

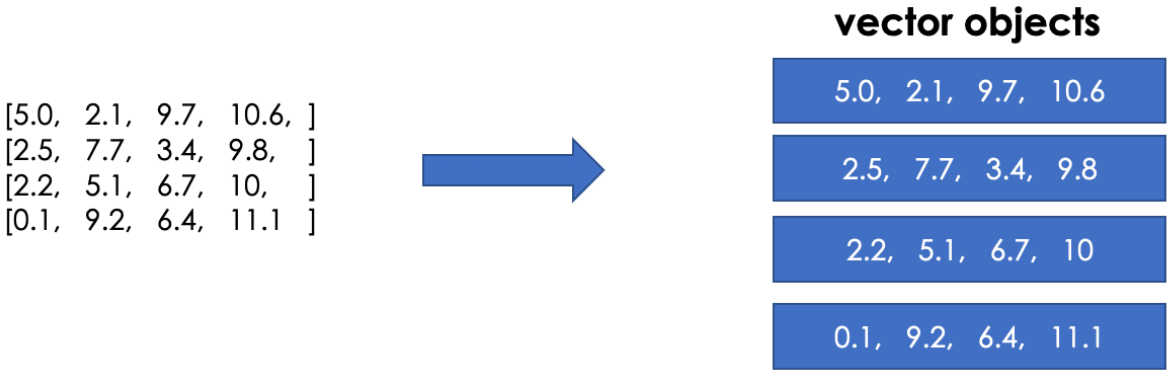
- a local matrix has integer-typed row and column indices and double-typed values, stored on a single machine.
  - dense matrix: entry values are stored in a single double array in column major
  - sparse matrix: compressed sparse column format

Distributed Matrix

- Distributed matrices are necessary when you're using ml algorithms on huge datasets
- they're stored across many machines, and they can have a large number of rows and columns
- Different forms
  - RowMatrix: used widely and we'll focus on it
  - IndexedRowMatrix
  - CoordinateMatrix
  - BlockMatrix

Row Matrix:

- Stores each row as a vector object



LabeledPoint

- LabeledPoint is another important data type
- it's a specialized vector that includes label and a feature vector

Training and Validation Data

- split the data into training and validation sets

- training set is used to train the model and a validation set is used to see how well the model performs on data that wasn't used to train it
- the usual split ration is 80% for the training set and 20% for the validation set

Predication

- you can now use the trained model to predict target values of vectors in the validation set by running predict on every element
- you can see how well the model is doing on the validation set by examining the contents of validPredicts:

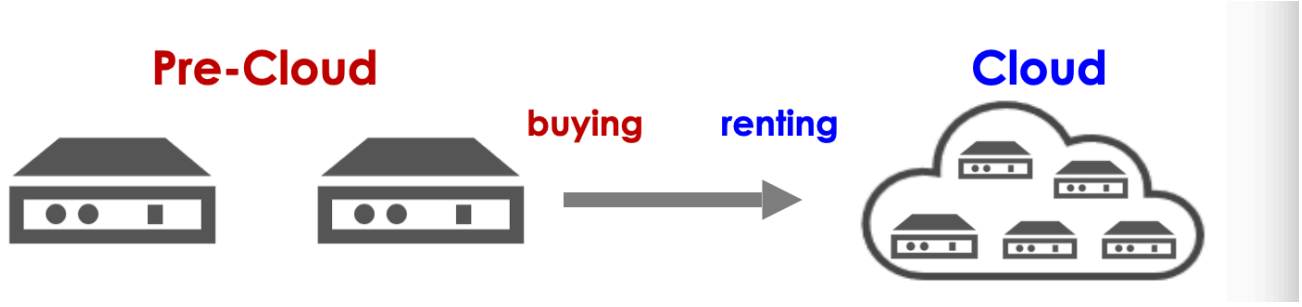
Model Evaluataion

- Some predictions are close to original labels, and some are further off. To quantify the success of your model, calculate the root mean squared error (root of the cost function defined previously)

How to build your IT infrastructure?

- Buy servers
  - how many? how much memory, disk, CPU, GPU?
  - how many users (particularly for special occasions)
  - Simple solution: provision for peak load, but underutilize in most times
- Build a data center
  - where to physically put the servers?
  - what if the machines are crashed?
  - what if we need a software upgrade
  - cooling techniques
  - skilled engineers
    Soln: All is possible without clod but at what cost?

Cloud Computing



- cloud computing is about buying vs renting
- instead of buying hardware, just renting an instance from a cloud provider (Amazon AWS, Google Cloud)

> *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*

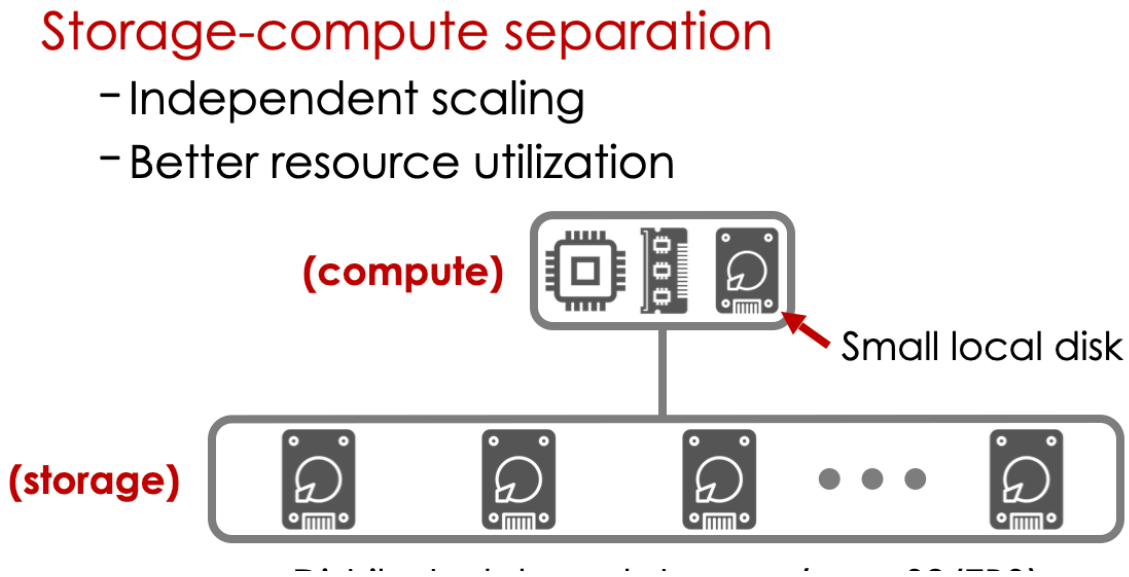
Key Characteristics

- Pay as you go
  - only pay for what you use with fine-grained metering, no up-front commitment (CPU per hour, memory per GB)
  - Elasticity

        - users can release resources if not needed or request additional resources if needed, ideally automatically (severless computing)
    - Virtualization
        - all the resources (cpu, memory, disk, network) are virtualized, no software is bounded to hardware
        - resources are pooled to serve multiple users (multi-tenancy)
    - Automation
        - No human interaction (start/stop a machine, crash, backup)
        - everything is managed by cloud providers

Cloud Native Databases

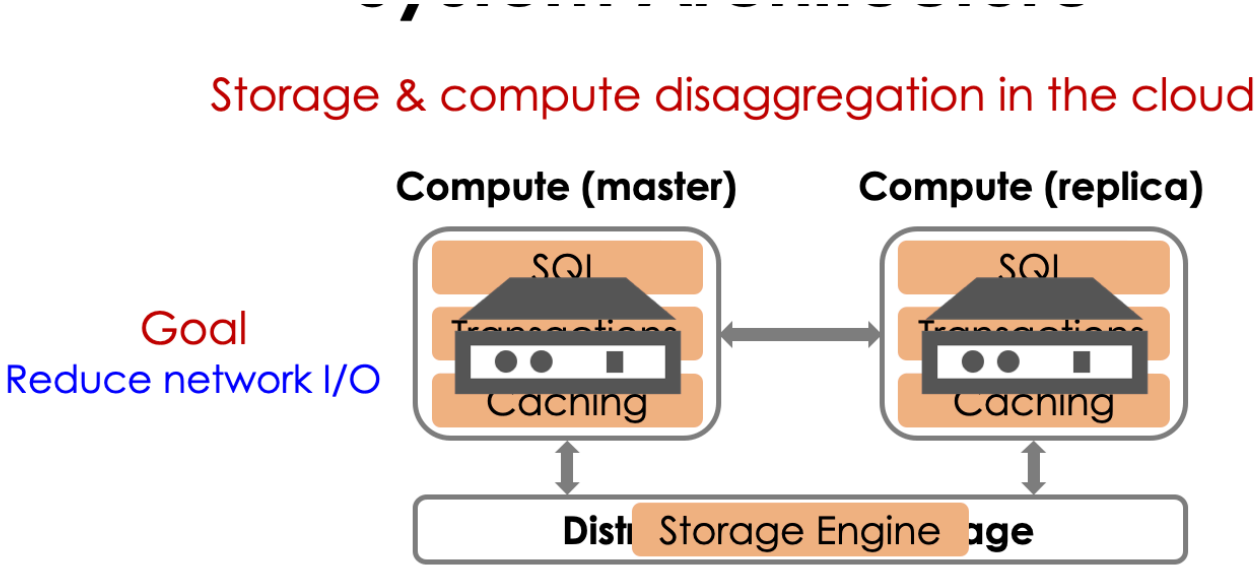
- Treat each cloud machine the same as in-house machine
- run existing database systems directly
- Cloud-native dbs are re-architectured to fully leverage the cloud infrastructure
    - resource/storage disaggregation
    - resource/storage pooling



Implications

- DB software level needs to be aware of the underlying hardware-level resource disaggregation
    - software level disaggregation
    - in order to enable more optimizations
- Distributed database architecture needs to be changed
    - from shared nothing to shared storage

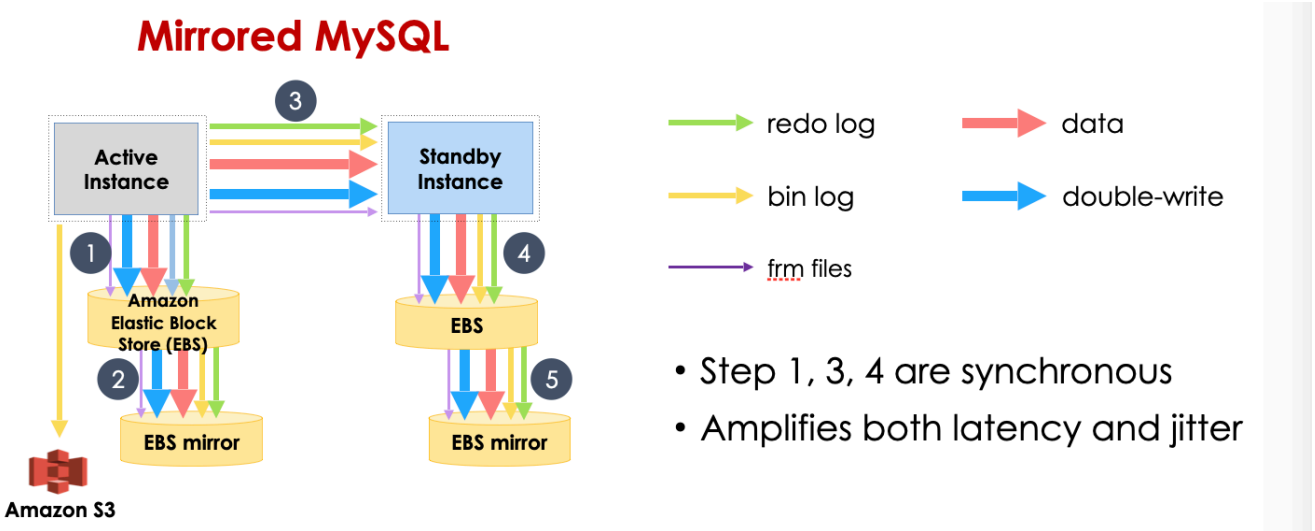
System Architecture



Main idea:

- Log is the database
  - only write redo logs on network
  - push log applicator to storage tier
- Asynchronous processing
  - materialize pages in background
- Buffer cache
  - to avoid network I/O
  - can rad pages upon cache miss

I/O traffic in traditional DB



I/O traffic in Aurora

- Only write redo log records

- all steps asynchronous
- 4/6 quorum storage
- 7.7X less data movement

Writes

- writes (trxns) send logs to storage asynchronously
- durability: each log is durable (ack) with 4/6 quorums
- Volume Durable LSN (VDL)
  - Log records can be lost, out of order
  - VDL: the largest one with all prior LSNs are durable

Transaction Commits

- transaction commits asynchronously
- when a transaction commits, mark its commit LSN
- commit only if VDL >= commit LSN

Replication: Scalability

- 1 writer and up to 15 reader instance
- To keep data consistent between writer and readers
  - writer sends logs to readers at the same time
  - once the reader receives logs, it will check if the page is in the cache
  - replication lag: 20ms

Reads (Caching)

- Each reader instance has a buffer (cache)
- upon read, check cache first
  - the cache is supposed to contain the latest data pages
  - except replication lags
    What if the cache is full?
- always evict a clean page: a page that's durable (pageLSN <= VDL)
- Why? O.w need to write dirty pages to storage, which increases network overhead

Crash Recovery

- if writer (master) is crashed, detected by HM (health monitor), promote a reader to writer first, and perform recovery
  - what if the failed writer comes back? -> contact HM
  - sometimes two master -> many unexpected issues

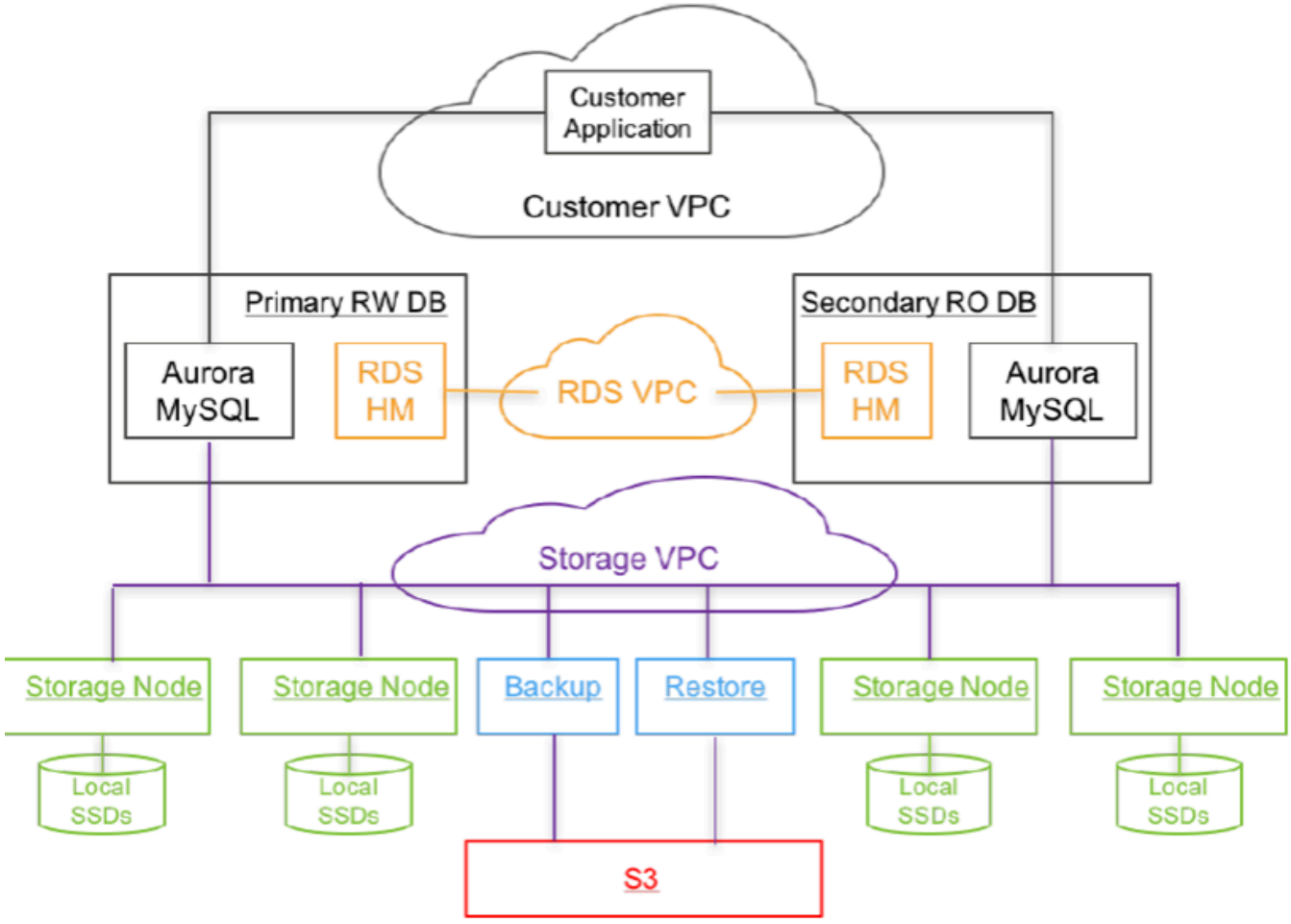
Crash Recovery

Traditional Databases

- Have to replay logs since the last checkpoint
- Typically 5 minutes between checkpoints
- Single-threaded in MySQL; requires a large number of disk accesses

Aurora

- No need to generate pages during recovery (very fast)
- Just need to re-establish VDL ➔ make sure storage is consistent
- Generate pages asynchronously, in parallel
- DB engine undo partial transactions
- Typically a few seconds



- Up to 5x faster than Cloud MySQL, but how about MySQL with local SSDs?

Comments on Aurora

- **New way of building cloud DB systems**
  - Monolithic (since 1970s) ➔ disaggregation
  - Hardware & software
- **Widely adopted in industry**
  - Microsoft Socrates DB
  - Alibaba PolarDB
  - Tencent CynosDB
  - Huawei TaurusDB
  - …

Hadoop is an open-source software framework that is used for storing and processing large amounts of data in a distributed computing environment. It is designed to handle big data and is based on the MapReduce programming model, which allows for the parallel processing of large datasets. Its framework is based on Java programming with some native code in C and shell scripts.

Hadoop is designed to process large volumes of data (Big Data) across many machines without relying on a single machine. It is built to be scalable, fault-tolerant and cost-effective. Instead of relying on expensive high-end hardware, Hadoop works by connecting many inexpensive computers (called nodes) in a cluster.

Hadoop Architecture

Hadoop has two main components:

- ****Hadoop Distributed File System (HDFS):**** HDFS breaks big files into blocks and spreads them across a cluster of machines. This ensures data is replicated, fault-tolerant and easily accessible even if some machines fail.
- ****MapReduce:**** MapReduce is the computing engine that processes data in a distributed manner. It splits large tasks into smaller chunks (map) and then merges the results (reduce), allowing Hadoop to quickly process massive datasets.

Hadoop Distributed File System (HDFS)

HDFS is the storage layer of Hadoop. It breaks large files into smaller blocks (usually 128 MB or 256 MB) and stores them across multiple DataNodes. Each block is replicated (usually 3 times) to ensure fault tolerance so even if a node fails, the data remains available.

****Key features of HDFS:****

- ****Scalability:**** Easily add more nodes as data grows.
- ****Reliability:**** Data is replicated to avoid loss.
- ****High Throughput:**** Designed for fast data access and transfer.

MapReduce

MapReduce is the computation layer in Hadoop. It works in two main phases:

1. ****Map Phase:**** Input data is divided into chunks and processed in parallel. Each mapper processes a chunk and produces key-value pairs.

2. ****Reduce Phase:**** These key-value pairs are then grouped and combined to generate final results.

This model is simple yet powerful, enabling massive parallelism and efficiency.

****Advantages:****

- ****Scalability:**** Easily scale to thousands of machines.
- ****Cost-effective:**** Uses low-cost hardware to process big data.
- ****Fault Tolerance:**** Automatic recovery from node failures.
- ****High Availability:**** Data replication ensures no loss even if nodes fail.
- ****Flexibility:**** Can handle structured, semi-structured and unstructured data.
- ****Open-source and Community-driven:**** Constant updates and wide support.

****Disadvantages:****

- Not ideal for real-time processing (better suited for batch processing).
- Complexity in programming with MapReduce.
- High latency for certain types of queries.
- Requires skilled professionals to manage and develop.

Problem 1 [20 points]

Hadoop MapReduce is a popular framework for big data analytics. In the lecture, we covered a simple example of *WordCount* that counts the number of times each word occurs in the input set of files. In this problem, we extend WordCount to include document IDs.

In particular, assume that you are given a collection of input documents and there are no duplicated words in the same file, but the same word might appear in different files. The problem is to compute for each word the documents that contain the word. You only need to show the document IDs where the corresponding documents contain the word.

Example Input. Assume that there are three documents:

- Doc 0: "Hello World"
- Doc 1: "Hello Data"
- Doc 2: "Big Data"

Example Output. The expected output (without particular ordering) is:

- Hello: 0, 1
- World: 0
- Data: 1, 2
- Big: 2

That's because the word "Hello" appears in both document 0 and 1. "Data" appears in both document 1 and 2. "World" appears in document 0 and "Big" appears in document 2.

a. **[8 points]** Briefly explain your idea to solve the problem using Hadoop MapReduce.

Answer:

Map: The Map phase reads each line of the document and emits tuples in the form (word, ID), where word is each word appearing in that line.

Reduce: The Reduce phase takes these tuples, groups them by word, and then writes the results as (word, sorted set of IDs).

b. **[12 points]** Write down the Map() function and Reduce() function to perform the above task using Hadoop MapReduce. You can write pseudocode.

Answer:

```
def map(ID, lines):
    for w in lines.split():
        print((w, ID), stdout)

def reduce(word, IDs):
    sorted_IDs = sorted(list(set(IDs)))
    print(f"{word}: {*sorted_IDs}")
```

Problem 2 [20 points]

Crash recovery is important in big data analytics. In this course, we covered failure handing in databases, Hadoop, and Spark. But they have different ways to handle failures and we have to understand why.

a. **[10 points]** If you compare databases with Hadoop, what's the difference of handing crash recovery? and why? You may explain the pros and cons of the approaches they used.

Answer:

Databases

• Pros: Using transactions and logs for crash recovery is more reliable and maintains data consistency.

• Cons: Without replicas, restarting a failed operation can be costly and slow down the system.

Hadoop

• Pros: Due to the presence of replicas, Hadoop can recover from failures by switching to a replica without terminating the current process.

• Cons: This approach requires additional storage to maintain the extra replicas.

b. **[10 points]** If you compare Hadoop with Spark, what's the difference of handing crash recovery? and why? You may explain the pros and cons of the approaches they used.

Answer:

Hadoop

• Pros: By using replicas, Hadoop can recover from failures without restarting the entire process, simply switching to a replica.

• Cons: Maintaining these replicas requires additional storage.

Spark

• Pros: Spark's use of cached intermediate RDDs allows for rapid recovery after a failure.

• Cons: Tracking lineage information for each operation can increase the overall time complexity.

Problem 3 [20 points]

In this problem, you will answer questions related to HBase.

a. **[10 points]** Conceptually, HBase and databases are two approaches to store tables with columns and rows. What're the advantages of HBase over databases in storing tables? In other words, when to use HBase over traditional database systems?

Answer:

HBase is especially useful when dealing with extremely large tables in terms of both rows and columns, and when these tables contain many empty (sparse) cells. It can also handle structured, semi structured, and unstructured data efficiently, providing high performance random access in such scenarios.

b. **[10 points]** If you compare HBase with HDFS, what're the pros and cons of the two approaches to store large-scale data?

Answer:

HBase

• Pros: Provides fast random access to specific pieces of data.

• Cons: Less efficient for purely sequential reads and writes, and requires more complex setup and maintenance compared to HDFS.

HDFS

• Pros: Offers very fast sequential data access, making it efficient for bulk reads and writes.

• Cons: Provides slow random access due to its design for batch-oriented processing rather than point queries.

Problem 4 [20 points]

In this question, you will answer questions related to Spark.

a. **[5 points]** RDDs are fundamental to Spark. Why are RDDs important to Spark?

Answer:

RDDs are fundamental to Spark because their in-memory computation reduces disk I/O, speeding up processing. They also provide built-in fault tolerance, ensuring data recovery in the event of failures. By supporting parallel processing through partitioning, RDDs optimize resource usage. Additionally, they minimize redundant computations and help maintain data consistency.

b. **[5 points]** Spark uses lazy evaluation to execute the jobs. Explain why.

Answer:

Spark employs lazy evaluation in executing jobs, which enables it to optimize processes and sidestep superfluous computations. This approach ensures that Spark caches only the results essential for the final computation. Such a strategy leads to a more efficient utilization of resources.

c. **[10 points]** Spark lineage graph includes two types of dependencies: narrow dependency and wide dependency. For the distributed join operation, which dependency does it belong to? Choose **only one answer** from the following list.

(1) narrow dependency;

(2) wide dependency;

(3) Both narrow and wide dependency.

Explain why.

Answer: (3) Both narrow and wide dependency.

If the data partitioning and join keys are such that each partition of the parent RDD is used by at most one partition of the child RDD, then the distributed join operation in Spark has a narrow dependency. Otherwise, it has a wide dependency. It is also possible for the same operation to exhibit both narrow and wide dependencies.

Narrow Dependency: If the join keys align with the partitioning keys, the join can be performed entirely within each partition, without accessing data from other partitions or nodes. In this scenario, each partition of the parent RDD corresponds to at most one partition of the child RDD, resulting in a narrow dependency.

Wide Dependency: If the join keys do not align with the partitioning keys, Spark must reshuffle the data across different partitions or nodes to gather all the relevant keys for the join. This scenario creates a wide dependency, as it involves combining data from multiple partitions and often requires significant data movement across the cluster.

Problem 5 [20 points]

a. **[10 points]** Spark GraphX uses the BSP (Bulk Synchronous Processing) model as the underlying computation model. Explain the pros and cons of BSP.

Answer:

Pros:

• Simplicity: The structure is easy to understand.

• Fault Tolerance: Improved detection and recovery from failures.

• Synchronization Barriers: Makes failure detection and recovery more manageable.

• Scalability: Efficiently handles large-scale workloads.

Cons:

• Synchronization Overhead: May cause inefficiency due to waiting at synchronization points.

• Workload Distribution Issues: Efficiency drops if tasks complete at significantly different times.

b. **[10 points]** Cloud-native databases are designed to separate the storage engine from the compute engine. Explain the pros and cons of this architecture.

Answer:

• Pros: Storage and compute resources can be customized separately, allowing for more precise resource management.

• Cons: Increased reliance on network performance and stability, which can affect overall system efficiency.

Problem 3 [20 points]

Consider relations $R(a, b)$ and $S(a, c, d)$ to be joined on the common attribute a. Assume that there are no indexes available on the tables to speed up the join algorithms.

- There are $B = 36$ pages in the buffer
- Table R spans $M = 1800$ pages with 100 tuples per page
- Table S spans $N = 600$ pages with 60 tuples per page

Answer the following questions on computing the I/O costs (in terms of number of pages) for the joins. You can assume the simplest cost model where pages are read and written one at a time. You may ignore the cost of the writing of the final results.

Some numbers Here are some numbers that may be useful in this problem:

- $\frac{1800}{36} = 50$; $\frac{1800}{35} = 51.4$; $\frac{1800}{34} = 52.9$; $\frac{1800}{33} = 54.5$; $\frac{1800}{32} = 56.3$
- $\frac{600}{36} = 16.7$; $\frac{600}{35} = 17.1$; $\frac{600}{34} = 17.6$; $\frac{600}{33} = 18.2$; $\frac{600}{32} = 18.8$
- $\log_{35} 50 = 1.1$; $\log_{35} 51 = 1.1$; $\log_{35} 52 = 1.1$; $\log_{35} 53 = 1.1$; $\log_{35} 54 = 1.1$; $\log_{35} 55 = 1.1$; $\log_{35} 56 = 1.1$; $\log_{35} 57 = 1.1$
- $\log_{34} 50 = 1.1$; $\log_{34} 51 = 1.1$; $\log_{34} 52 = 1.1$; $\log_{34} 53 = 1.1$; $\log_{34} 54 = 1.1$; $\log_{34} 55 = 1.1$; $\log_{34} 56 = 1.1$; $\log_{34} 57 = 1.1$
- $\log_{35} 16 = 0.8$; $\log_{35} 17 = 0.8$; $\log_{35} 18 = 0.8$; $\log_{35} 19 = 0.8$
- $\log_{34} 16 = 0.8$; $\log_{34} 17 = 0.8$; $\log_{34} 18 = 0.8$; $\log_{34} 19 = 0.8$

Questions

a. **[6 points]** For the Grace hash join algorithm (with S as the outer relation and R as the inner relation), what is the I/O cost of partition phase? What is the I/O cost of the probe phase?

Solution:

Partition Cost: $2 * (M + N) = 2 * (1800 + 600) = 4800$

Probe Cost: $M + N = 1800 + 600 = 2400$

b. **[5 points]** For the block nested loop join with R as the outer relation and S as the inner relation, what is the I/O cost?

Solution:

$M + \lceil \frac{M}{B-2} \rceil \times N = 1800 + \lceil \frac{1800}{34} \rceil \times 600 = 1800 + 31800 = 33600$

c. **[9 points]** For the sort-merge join with S as the outer relation and R as the inner relation, what is the I/O cost of sorting S? What is the I/O cost of sorting R? What is the I/O cost of merging?

Solution:

Sorting S:

$passes = 1 + \lceil \log_{B-1} (\lceil \frac{N}{B} \rceil) \rceil = 1 + \lceil \log_{35} (\lceil \frac{600}{36} \rceil) \rceil = 1 + \lceil \log_{35} 17 \rceil = 1 + \lceil 0.8 \rceil = 2$

$2N \times passes = 2 * 600 * 2 = 2400$

Sorting R:

$passes = 1 + \lceil \log_{B-1} (\lceil \frac{M}{B} \rceil) \rceil = 3$

$2M \times passes = 2 * 1800 * 3 = 10800$

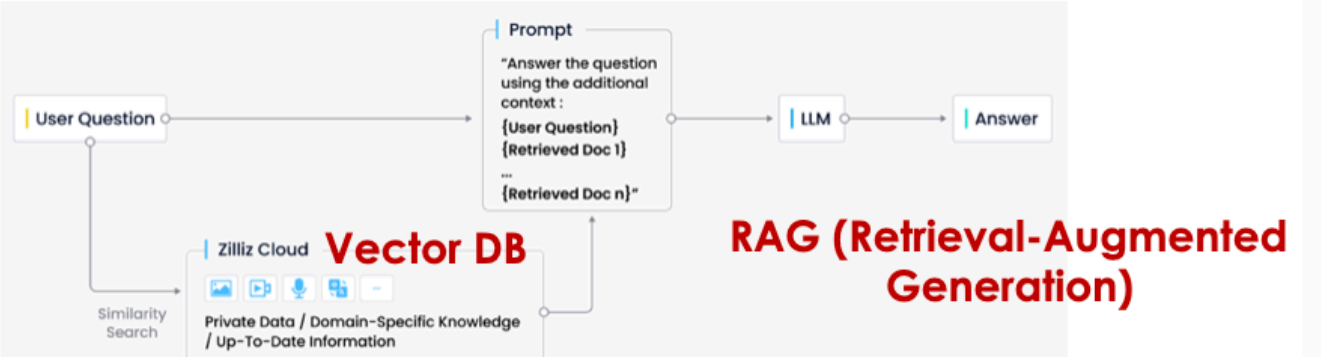
Merging:

$M + N = 1800 + 600 = 2400$

High Dimensional Vector Data

Data -> Vector -> Analytics, this is know as vector embedding

- Vector DBs can address many critical limitations of LLMs
  - hallucination: incorrect or fabricated answer
  - lacking domain-specific knowledge
  - up-to-date information



How to find similar vectors ?

- High dimension
- top k similar vectors

Euclidean distance $$d(\mathbf{A}, \mathbf{B}) = \sqrt{\sum_{i=1}^{n}(A_i - B_i)^2}$$

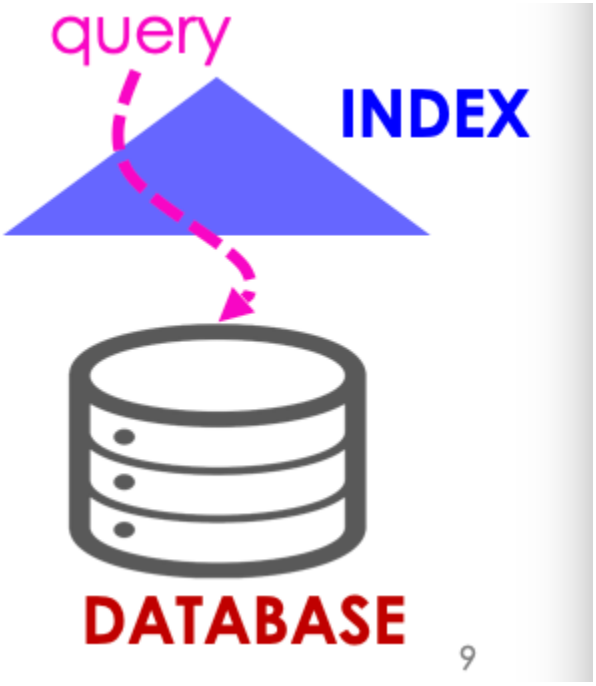
Cosine similarity $$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Dot product $$dot(\mathbf{A}, \mathbf{B}) = \sum_{i=1}^{n} A_i \cdot B_i$$

- sometimes you don't even know the exact answer (personalized recommendation)
- sometimes we just need fast performance

- hard to have both

Indexing and Searching



1. Preprocessing stage (offline): build index
   - build a proper index on the data vectors
   - we don't care too much about the time spent here
2. Online search stage
   - given a query, search the index to know potentially relevant vectors
   - this can filter out many non-relevant vectors
   - this stage is more important

Evaluation Metrics

- Query time -> performance
- space overhead, especially memory overhead -> cost
- Accuracy
  - the ratio between the returned results vs the true top k results
    Others
- index construction time
- update cost: new vectors

The outiline:

- introduction
- main memory vector index
- disk based vector index

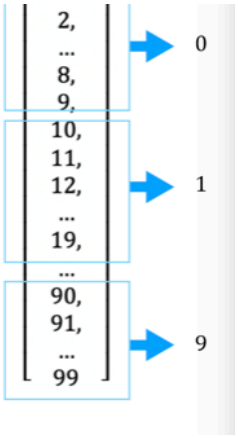
- vector search in databases

Vector indexes (main memory)

- Quantization based indexes (widely used in vector DBs, IVF_FLAT, IVF_PQ)
- Graph Based Indexes (NSW, HNSW, also used widely in vector DBs)
- tree-based indexes
- hash-based indexes

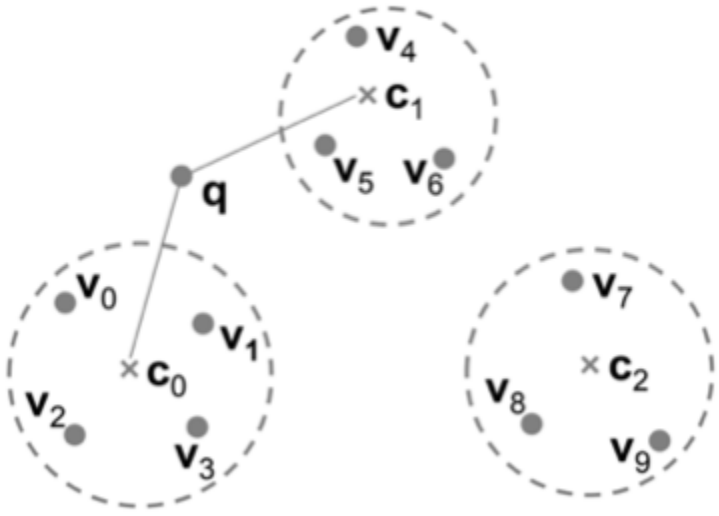
Quantization is a way of approximation

- Let's look at quantization in 1-dimensional space
  - $Q(x) = \left\lfloor \frac{x}{10} \right\rfloor$, where x is an input value
  - input = 3, $Q(3) = \left\lfloor \frac{3}{10} \right\rfloor = \lfloor 0.3 \rfloor = 0$
  - input = 3, $Q(91) = \left\lfloor \frac{91}{10} \right\rfloor = \lfloor 9.1 \rfloor = 9$
  - Those 99 integers can be quantized into a smaller set of 10 buckets



Quantization in high dimensional space

- it's basically clustering (k-means)



IVF_FLAT

- index phase
  - cluster n vectors into k clusters (quantization)

    - centroids: c0, ..., ck-1
- Search phase
    - given a query q, find the closest u clusters based on centroids
    - u: user-defined parameter
- only scan the vectors in the u clusters
- But how do we quickly compute the similarity between q and a vector vi, in a cluster?

A naive approach:

- A for-loop to compute dist(q,vj)
- d steps (where d is dimensionalty, d = 100)

Since we know the centroid c, we can pre compute the distance of (c,vj)
then dist(q,vi) = dist(q,c) + dist(c,vi)

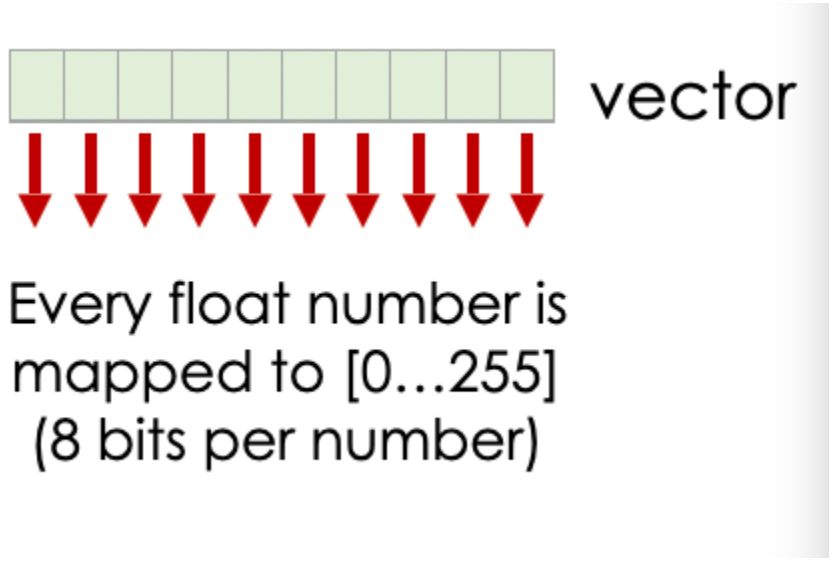
Now we only need 1 step (instead of d steps) to compute distance!

With compression we can reduce the space overhead of IVF_FLAT

Example: YouTube 8M data includes 1.4 billion vectors
each vector takes 1024 dimensions (each float takes 32 bits)
5.6TB space memory !

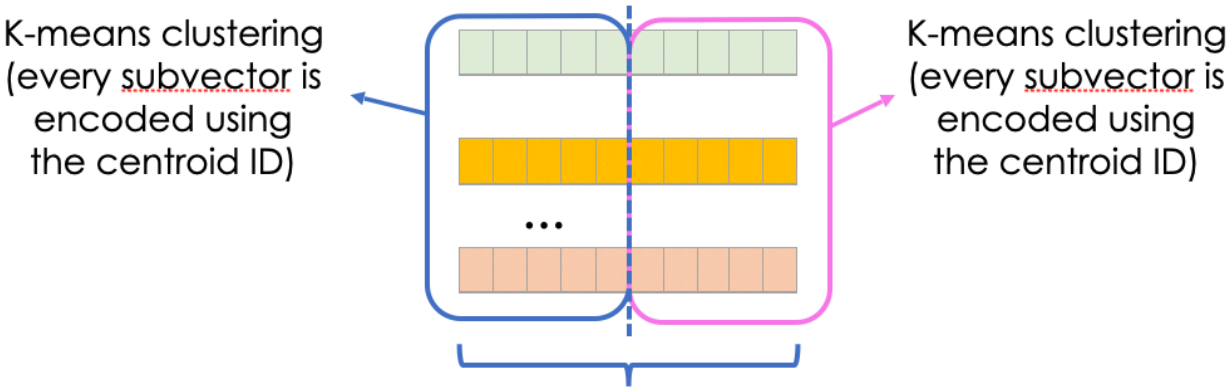
Basic Idea of Compression

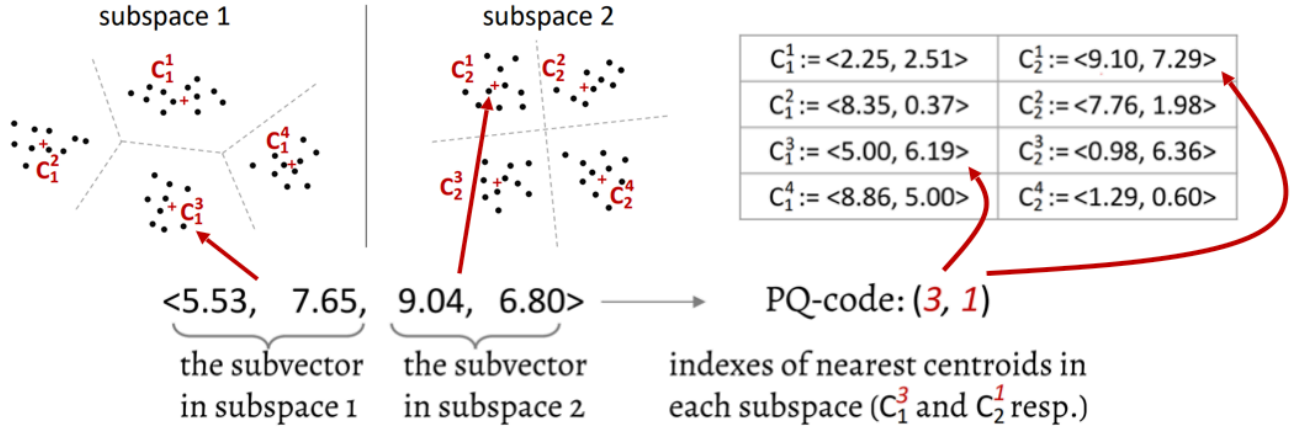
- instead of using 32 bits to represent a float number
- use L bits (L=8)
- think of 1-d quantization
- every float number in a vector is quantized into (0...2^L-1)
- The 1.4 billion vectors will take 1.4TB space (if L=8)



Compression: Product Quantization (PQ)
main idea: compress between multiple dimension together

- every vector is partitioned into M subvectors, M=8
- every subvector is compressed together using L bits
  To compress subvectors
- Each vector vi is partitioned into M subvectors (0 to m-1) (m subspace)
- all the vectors in the same subspace are compressed together using high dimensional quantization clustering (clustering)
  - All $v_0^0, v_1^0, v_2^0 \ldots, v_{n-1}^0$ are compressed together
  - All $v_0^1, v_1^1, v_2^1 \ldots, v_{n-1}^1$ are compressed together
  - Every subvector is represented using the centroid ID





Original vector is compressed as <5.00, 6.19, 9.10, 7.29>

Each vector is compressed using M * L bits (M=8, L=8)

Regardless of the dimensionality

- but the parameters can be tuned based on dimensionality

example: consider the 1.4 billion vectors again

- each vector will take 8 * 8, (8 bytes)
- the 1.3 billion vectors will take 11.2GB space

Very good way of compression
Another benefit of PQ: Fast distance computation

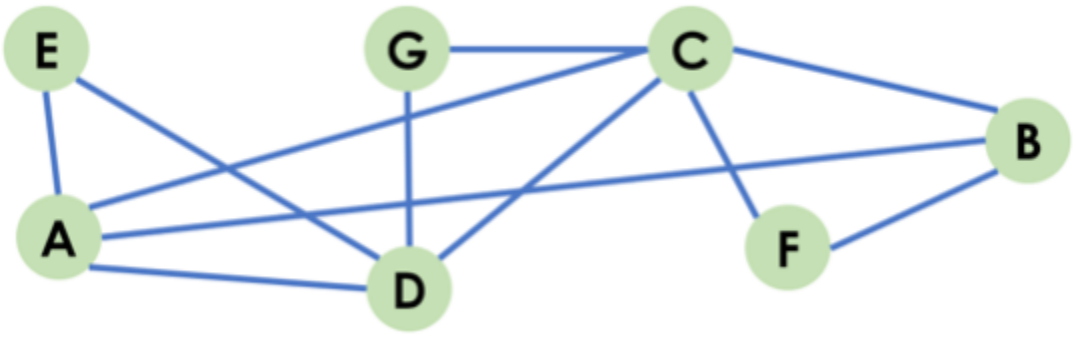
- all the distance in subspace can be precomputed
  Example:
  - Vector X → PQ code (3, 1)
  - Vector Y → PQ code (1, 5)
  - Dist(X,Y) = dist(3,1) + dist(1,5), where each part can be precomputed

IVF_PQ

- similar to IVF_FLAT
- difference is that
  - each cluster applies PQ
  - but using residual vectors (instead of the original in the cluster)
- Search process is the same

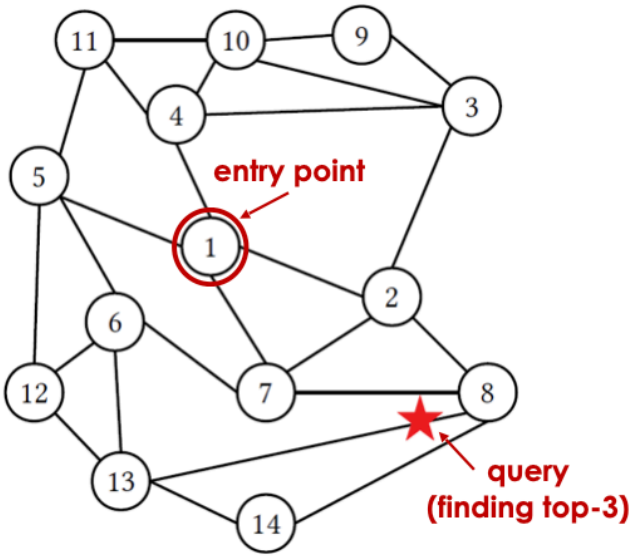
Graph-Based Vector Index

- main idea
  - for each vector, pre-compute the nearest neighbor
  - connect them using a graph
  - convert vector search problem to graph traversal problem

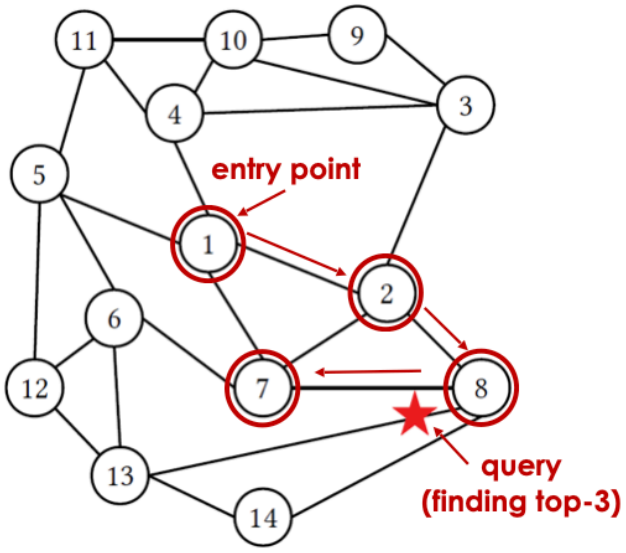


- Navigable small worlds(NSW)
  - add new vertices to the index

    - for each new vertex (vector), find the closest m neighbors seen so far and connect with them
    - Balance: index construction time and query performance
- Can be extended to KNN by maintaining a result set and a candidate set
- terminate if the min distance in the result set > max distance in the candidate set



- **q: candidate queue** (min-heap)
- **topk: result queue of size k** (max-heap)
  - Each heap stores the vector and distance to the query
- **Stop condition**
  - min distance in q > k-th max dist in topk
  - ➔ unseen nodes won't be in top k
- Initially, q = {1}, topk = { }
- Let x = pop min from q, which is 1
- Check the stop condition
- If met, stop
- Otherwise
  - Insert x to topk
  - Insert x's unvisited neighbors to q (as candidates)



- Initially, q = {1}, topk = { }
- Then x = 1, topk = {1}, q = {2, 4, 5, 7}
- Then x = 2, topk = {1, 2}, q = {8, 3, 4, 5, 7}
- Then x = 8, topk = {1, 2, 8}, q = {14, 13, 3, 4, 5, 7}
- Then x = 7, topk = {7, 2, 8}, q = {14, 13, 3, 4, 5, 6}
- Then x = 14, which is bigger than 7 (the k-th max distance in topk)
- ➔ No need to check the candidates anymore