CS4225 Cheatsheet AY23/24 —— @JasonYapzx

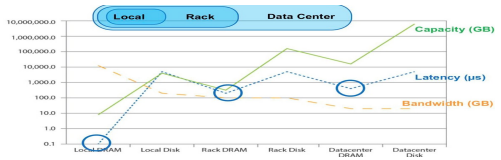
Infrastructure

Dynamically provision resources, scale up/down based on demand.

- **VM:** sharing of hardware resouce by running application in isoldated VM - higher overhead, each VM has own OS.
- **Containers:** Lightweight sharing or resource, share same OS.
  - *IaaS*: Infra, utility computing (EC2, Compute Engine etc.)
  - *PaaS*: Platform, takes care of hardware, platform to host
  - *SaaS*: Runs everything for you

Bandwidth vs Latency

- **Bandwidth:** max amount of data transmitted per unit time (GB/s)
- **Latency:** time taken for 1 packet go src → dest/back to source (ms)
- **Throughput:** Similar to bandwidth → capacity, data that was *actually transmitted* during a period of time



Disk reads more expensive than DRAM, has higher **latency** and lower **bandwidth**. They instead provide more capacity. As we move up storage hiearchy, costs for latency increases and bandwith decreases.

MapReduce

3 main challenges we have to deal with for distributed computing:

1. **Machine Failures:** Not all machines are available all the time
2. **Synchronization:** Do not know order of worker running, if any shared data manipulated etc. (*barriers* between `map()`/`reduce()`).
3. **Programming Difficulty:** Hard to debug multiple interacting services

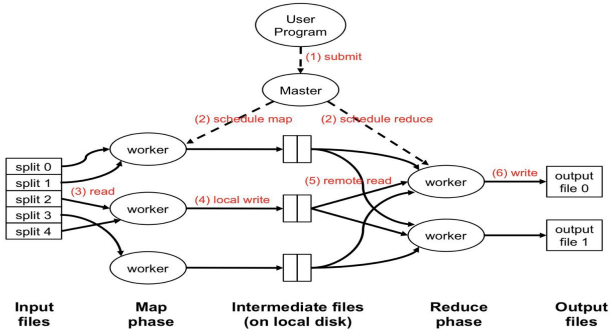
Map, Shuffle, Reduce

We generally specify 2 functions `map()` and `reduce()`. The shuffle stage is done by the framework, used to sort the items to the reducers:

- $map(k_1, v_1) \rightarrow List(k_2, v_2)$
- $reduce(k_2, List(v_2)) \rightarrow List(k_3, v_3)$
- values with the same key sent to the same reducer

Implementation

1. **Submit:** user submits program (`map()`/`reduce()`, and config)
2. **Schedule:** master schedules resources for tasks (no data)
3. **Read:** input files split into 128MB (1 task), worked on 1 @ a time
   - Map: worker iterates & computes over each `<key, value>` tuple
4. **Local write:** Each worker writes ouputs of `map()` to intermediate files on *local* disk. Files are partitioned by key.
5. **Local read:** Reduce worker responsible for > 1 keys. Data needed read from corresponding partition in mapper's local disk.
6. Reduce: Receiving all needed key value pairs, it computes `reduce()`.
7. Write: Output of `reduce()` written to HDFS.
8. Shuffle: Comprised of local write/remote read steps.



⋆ Task given to processes or unit of job **NOT worker**. Mappers/reducers are the processes running the task, not physical machines.
⋆ Splitsize: divide work into diff. machines for distributed computing:

- **Large:** not taking advantage of distributing computing.
- **Small:** Large overhead by master node.

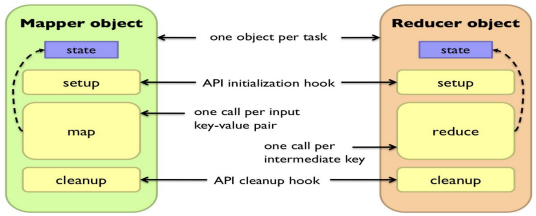
Partition and Combiner

Partitioners: Assignment of keys to reducers determined by a hash function (default), users can implement a custom partition to better spread load among reducers.
Combiners: Writing out map output to disk is expensive, reduce disk writes if possible by *locally aggregating* output from mappers.

- Generally, correct to use reducers as combiners if the reduction involves a binary operation that is both: **Associative:** $a + (b + c) = (a + b) + c$ AND **Commutative:** $a + b = b + a$

Guidelines for Basic Algorithmic Design

- **Linear scalability:** more nodes do more work at the same time, (datasize/compute resource)
- **Minimize disk/network I/O:** sequential vs random, send in bulk vs send in small chunks
- **Reduce memory working set:** portion of memory actively being used during algorithm execution



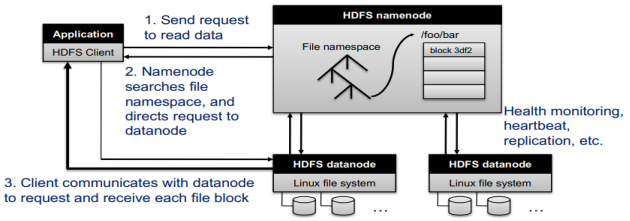
Secondary Sort

Each reducer's values arrive unsorted, if we want them to be sorted we can define a new *composite key*. Partitioner now needs to be customized to partition by K1 only, not $(K1, K2)$.

```
1 public int compareTo(DateTemperature pair) {
2     int compareValue == this.yearMonth.compareTo(pair↩
          .getYearMonth());
3     if (compareValue == 0) {
4         compareValue = temperature.compareTo(pair.↩
              getTemperature());
5     }
6     return -1 * compareValue; // sort descending
7     return compareValue; // sort ascending
8 }
```

Hadoop File System

- Files stored as chunks (fixed size)
- Reliability through replication, each chunk replicated across 3+ chunkservers. **No concurrent writing** of same file.
- Single master to coordinate access and metadata, simple centralized management.

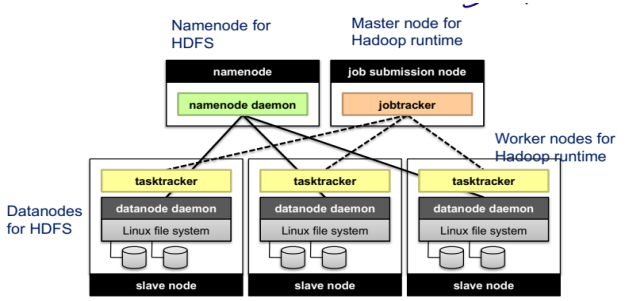


Q: How to perform replication when writing data?

A: Namenode decides which datanodes are to be used as replicas. The 1st datanode forwards data blocks to the 1st replica, which forwards them to the 2nd replica, and so on.

Namenode Responsibilities

- Managing file system namespace:
  - Contains file/dir structure, metadata, file-to-block mapping, access permissions etc, **no data moved through the here**
  - Coordinates file operations, directing read/write ops
- Maintaining overall health:
  - Periodic communication with datanodes, (monitoring check)
  - Block re-replication and rebalancing + Garbage collection
- What if namenode's data is lost
  - All files on system cannot be retrieved, no way to reconstruct them from raw data blocks e.g. Hadoop has backup namenodes.



MapReduce and DataMining

Relational Databases —— CS2102 Recap

Selection: σ_e: Map: take in a tuple (with tuple ID as key), emit only tuples that meet the predicate, no reducer needed.

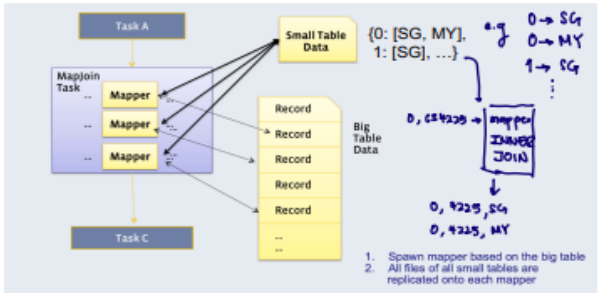
Projection ρ_l: Map: take in a tuple (with tuple ID as key), emit new tuples with appropriate attributes, no reducer needed (⇉ no shuffle).

Group by... Aggregation e.g. average sale price per product:

- **SQL:**
  `SELECT product_id, AVG(price) FROM sales GROUP BY product_id`
- **MapReduce:**
  - Map over tuples, emit `<product_id, price>`
  - Framework automatically groups tuples by key
  - Computes average in reducer, optimize with combiners

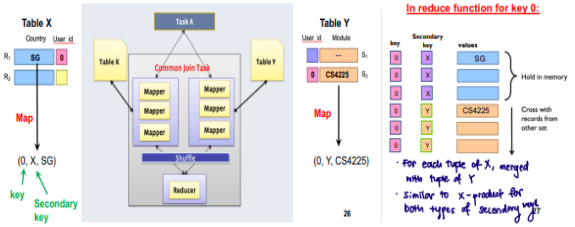
Inner Join: Include **only** tuples that satisfy the condition. Focusing on many-to-many relationships: a particular user ID can appear multiple times in both tables.

- **Broadcast ('MAP') Join:** Requires one table to **fit in memory**:
  - All mappers store a copy of the small table (hash table with keys). Iterate over big table and join records with small table
  - ↑ memory, ↓ I/O as no need to reduce, no emit.



- **Reduce-side ('COMMON') Join:** Does not require dataset to fit in memory, slower than broadcast.
  - Different mappers operate on each table, emit records (more I/O operations) with keys as variables to join by.
  - In reducer, use *secondary sort*, ensures all keys from X arrive before Y. Keys from X in memory, crossed with records from Y.

– ↑ I/O, ↓ memory, emits more tuples, no need to fit large table.



Similarity Search

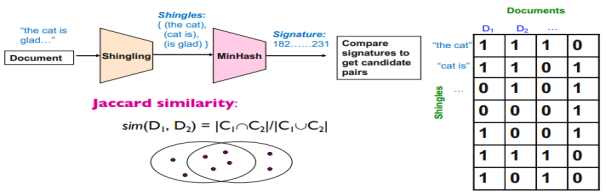
Define near problems as points that are *small distance* apart, low distance, high similarity.

- **Euclidean Distance:** $d(a,b) = \sqrt{\sum_{i=1}^{D}(a_i - b_i)^2}$
- **Manhanttan Distance:** $d(a,b) = \sum_{i=1}^{D}(|a_i - b_i|)$
- **Cosine Similarity:** $s(a,b) = \cos\theta = \frac{a \cdot b}{||a|| \cdot ||b||}$ (**direction only**)
- **Jaccard Similarity:** between sets AB: $s_{Jaccard}(A,B) = \frac{|A \bigcup B|}{|A \bigcap B|}$
- **Jaccard Distance:**: $d_{Jaccard}(A,B) = 1 - s_{Jaccard}(A,B)$

e.g. Finding Similar Documents —— Goals

- **All pairs similarity:** Given large number N of documents, find ALL duplicate pairs —— Jaccard distance below a threshold
- **Similarity search:** Given query document D, find all documents that are near duplicates of D

Steps for Similar Docs

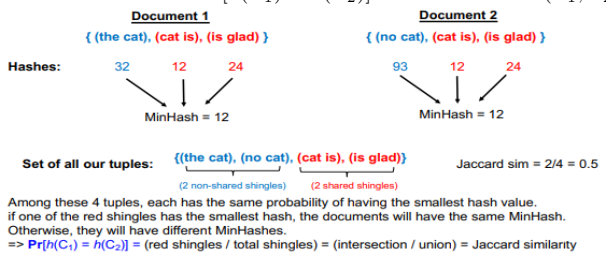


- **Shingling:** Convert documents to sets of short phrases
  - **k-shingle / k-gram:** is a sequence of k tokens that appears in the doc. (above example is k=2, set of 2-shingles: $S(D_1)$)
  - Matrix Representation, columns → documents, shingles → rows, compare using Jaccard similarity: $sim(D_1, D_2) = \frac{|C_1 \bigcap C_2|}{C_1 \bigcup C_2}$
- **Min-Hashing:** Convert sets to short signatures:
  - Signature: compressed block of data representing contents of document. Documents with same signature are candidate pairs for finding near-dupes as similarity is preserved.
  - **Motivation:** Need to compute pairwise Jaccard similarities for every pair of docs. $N(N-1)/2 \approx 5 * 10^{11}$ comparisions (slow!)

MinHash Overview

Hash each column **C** to a small **signature** $h(C)$ s.t.
1. $h(C)$ small enough to fit in RAM
2. highly similar documents usually have same signature
3. Find hash function $h(\cdot)$ s.t.
   - $sim(C_1, C_2)$ is high, high probability $h(C_1) = h(C_2)$
   - $sim(C_1, C_2)$ is low, high probability $h(C_1) \neq h(C_2)$

Key property: Pr that 2 docs have same MinHash signature = Jaccard similarities! $Pr[h(C_1) = h(C_2)] = Jaccard - Sim(C_1, C_2)$



Map Reduce Implementation

- **Map:** Read over document and extract shingles. Hash each shingle and take min of them to get MinHash signature. Emit `<signature, doc_id>`
- **Reduce:** Receive all documents with given MinHash signature. Generate all candidate pairs from these documents.
  - **Candidate pairs:** same final signatures. Either used as final output, or pairwise comparison to check *actually similar pairs*.

K-means algorithm

1. **Initialization**: Pick K random points as center
2. Repeat:
   - **Assignment:** assign each point to nearest cluster
   - **Update:** move each cluster center to avg of its assigned points
   - **STOP** if no assignments change

- MapReduce usually more suitable for Batch processing, Spark would be more used for iterative solutions (like K-means).
- This is a single iteration for K-means with MapReduce. This is *inefficient* as the mappers are mapping the entire dataset's data per iteration, incurring a high amount of network I/O traffic.
- To optimise this, we can combine each tuple's type into a hashmap, and emit the KVP of each item in the hashmap. Since each point is assigned to a cluster, key is cluster id, and the hashmap can accumulate all the coordinates of the points for this cluster.

```
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, centroid m)
class Reducer
    method REDUCE(clusterid n, points [p1, p2, ...])
        s <- INITPOINTSUM()
        for all point p in points:
            s <- s + p
        m <- COMPUTECENTROID(point s)
        EMIT(clusterid n, centroid m)
```

Tutorial 1 Summary

- Ensure Combiners have same **input** and **output** key value type, must be the same as mapper output type and reducer input type.
- Combiners are expected to not affect the result of the MapReduce operation regardless of however many times ran (0, 1, ... N times).
- We use combiners to transform a non-associate opertaion (e.g. finding average) into associative one (element-wise sum of numbers, count numbers and division done in reducer)

NoSQL

- Non-relational database, does not stores data in relational tables.
- `SQL` refers to traditional DBMS, usually used with NoSQL DB.
  - Horizontal scalability
  - Replicate/distribute data over many servers (Volume, Velocity)
  - Flexible schemas (Variety)
  - Simple call interface, weaker concurrency model than RDBMS
  - Efficient use of distributed indexes and RAM

Types of NoSQL Databases

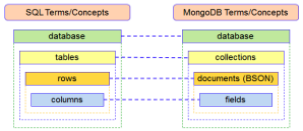
Key-Value Stores (Redis, DynamoDB)

Stores associations between **keys and values**, keys are primitives usually to be queried (`strings`, `ints`, `raw bytes`). Values can be primitive/complex (cannot be queried) - `ints`, `string`, `lists`, `JSON`, HTML fragments, BLOB (basic large object)

- Simple API: `GET` or `PUT` value associated with key (also have multi-get, multi-put, range queries)
- Suitable for small continuous reads/writes. Storing basic info with no clear schema, and no complex queries.
- e.g. User sessions, Caching, user data often processed individually.
- Persistence: if memory is in a disk (Dynamo, RocksDB) or just in big in-memory hash table(Memcached, Redis)

Document Stores (MongoDB, CouchDB)

Database → multiple collections → multiple documents. Document → JSON-like obj w/ **fields** & **values**.



- Allows some querying based on content of a document
- **Create:**

```
db.users.insert(
  {
    name: "sue",
    age: 26,
    status: "A"
  }
)
```

- **Update:**

```
db.users.update(
  { age: { $gt: 18 } },
  { $set: { status: "A" ↩
        } },
  { multi: true }
)
//$set = UPDATE action
```

- **Read:**

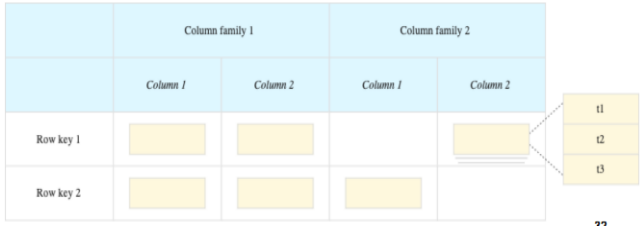
```
db.users.find(
  { age: { $gt: 18 } },
  { name : 1,
    address: 1
  } // 1 = include
).limit(5)
//$gt = greater than
```

- **Delete:**

```
db.users.remove(
    { status: "A" }
)

// remove() contains ↩
    remove criteria
```

Wide-Column Stores (BigTable, Cassandra, HBase)

Rows describes entities, related groups of columns grouped as **column families**. **Sparsity:** if a column not used for a row, no space used.



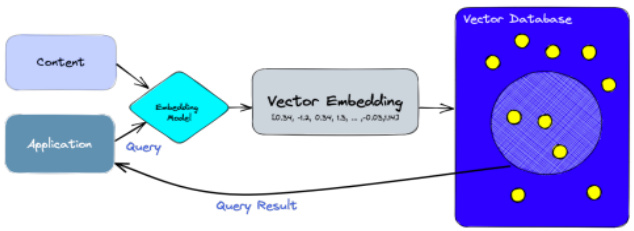
Graph Databases

Information on each edge and each node to see how different nodes relate to one another (more later).

Vector Databases (Milvus, Redis, Weaviate, MongoDB Atlas)

Store **vectors** (each row represents a point in d dimensions), → usually dense, numerical and high dimensions.

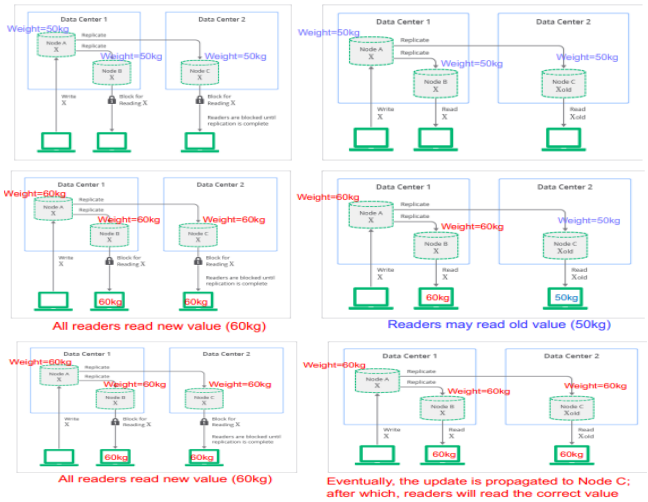
- Allows fast **similarity search**: given a query, retrieve similar neighbours from database. Using **locality-sensitive hashing** (LSH) which is similar to min-hashing.
- Features: scalability, real-time updates, replication.
- LLMs often used to convert some texts into vectors/embeddings. Useful for search, recommendation, clustering applications.
- To quickly search for similar embeddings, efficient and scalable vector DBs are needed.



Strong vs Eventual Consistency

Depends on the nature of the queries, e.g. finance applications whereby we need all queries to return the same value → Strong.

- **Strong**: any reads immediately after an update must give the same result on all observers
- **Eventual**: if the system is functioning and we wait long enough, eventually all reads will return the last written value



ACID vs BASE

- Relational DBMS provide strong (ACID) guarantees
- NoSQL systems relax this to weaker BASE approach
  - **Basically Available:** basic read/write operations available most of the time (availability & low latency)
  - **Soft State:** without guarantees, we only have some probability of knowing the state at any time
  - **Eventually Consistent:** Constract to *STRONG*
- **Implications:** Eventual offers better availability at the cost of much weaker consistency guarantee.
- (there are recent systems that are configuarable for multiple consistency levels —— 'tunable consistency')

Duplication

When we join in normal RDBMS, we join along the common `id` between both tables. How about NoSQL? NoSQL usually uses Duplication (denormalization) rather than joining.

- Storage is cheap, duplicate data to improve efficiency.
- Tables are designed around the queries we expect to receive. Beneficial especially when we need to process a fixed type of queries.
- Leads to new problem, what if user changes their name → programmed into multiple tables.

Pros

+ **Flexible / dynamic schema**: suitable for less well-structured data
+ **Horizontal scalability**: we will discuss this more next week
+ **High performance and availability**: due to their relaxed consistency model and fast reads / writes

Cons

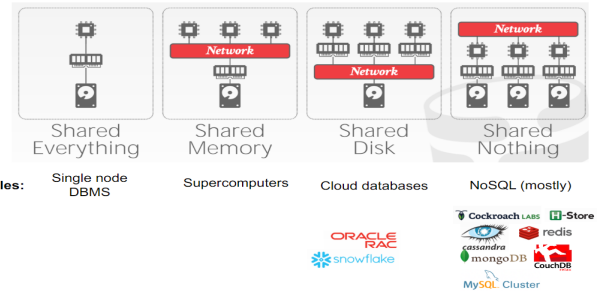
- **No declarative query language**: query logic (e.g. joins) may have to be handled on the application side, which can add additional programming
- **Weaker consistency guarantees**: application may receive stale data that may need to be handled on the application side

Distributed Databases

Assumptions: All nodes are well-behaved (follows protocol designed for them), not adversarial / trying to corrupt the DB — Byzantine Fualt Tolerant, (blockchains solve this).

- **Scalability:** allow DB size to scale by adding more nodes
- **Availability / Fault Tolerance:** one node fails, others nodes serve
- **Latency:** Each request served by closest replica, ↓ latency, when database is distributed globally
- Abstract from users how data is physically distirbuted, partitioned and replicated. Any query to a single node should work the same as one in the distributed database.

Distributed Database Architectures



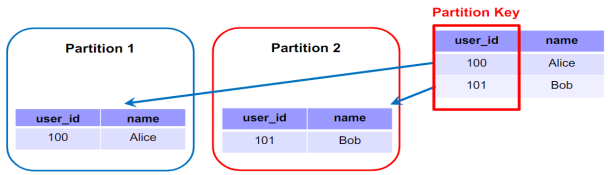
Data Partitioning

Table Partitioning

Put different tables, or collections on different machines. **NOT SCALABLE** as each table cannot be split across multiple machines.

Horizontal Partitioning

Different tuples stored in different nodes, (aka **sharding**).



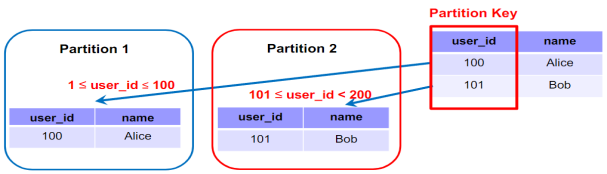
- **Partition Key:** (Shard key) is used to decide which node each tuple will be stored on, (same shard key, same node)
  - If we need to filter tuples based on column often, or `GROUP BY` column, then that column is a **suitable partition key**
  - e.g. filter tuples by `user_id=100`, then `user_id` is the partition key. All `user_id=100` data stored in same partition.
  - Data from other partitions ignored (partition pruning), saves time as we do not have to scan for these objects.

⋆ cityid as key, when is this good / bad:

- Good if we aggregate data only within individual citiies. Bad if there is too few cities (low **cardinality**). This would cause a lack of scalability.
- Similar problems would occur if some cities have too **high frequencies**. These can be mitigated by using **composite keys**.

Horizontal Partitioning —— Range Partitioning

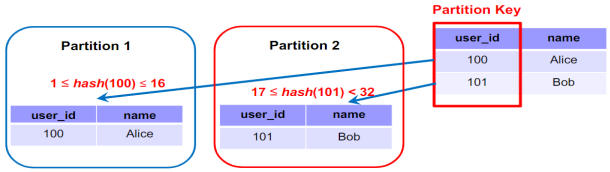
Split partition key based on range of values



- Beneficial if we need range-based queries. If the user queries for `user_id` < 50, all data in partition 2 can be ignored
- **BUT** range partitioning can lead to imbalanced shards, e.g. many rows have `user_id = 0`
- Splitting the range automatically handled by a balancer (tries to keep shards balanced).

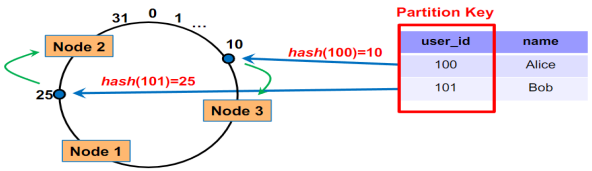
Horizontal Partitioning —— Hash Partitioning

Hash partition key, then divide that into partitions based on ranges



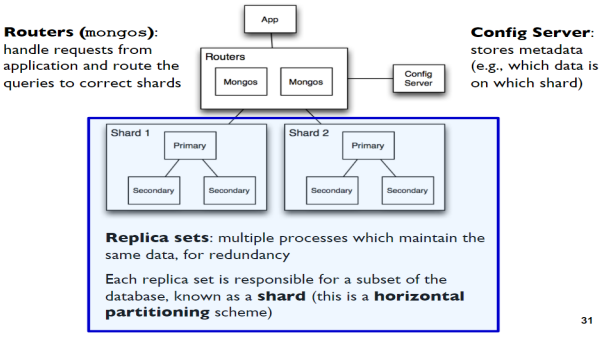
- Hash f^n automatically spreads out partition key values evenly
- For previous approaches —— how to add / remove a node? If we completely redo the partition, a lot of data may have to be moved around, which is inefficient. **Answer:** consistent hashing!

Consistent Hashing

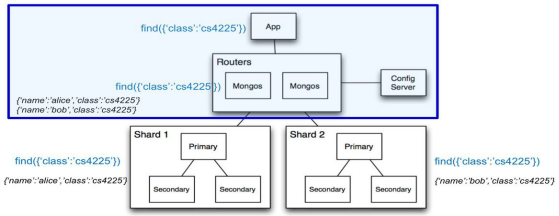


- Think of the output as a hash function lying on a circle
- **How to partition:** each node has a 'marker' (rectangles), Each tuple placed on the circle, and assigned to the node that comes clockwise-after it.
- To **delete a node**, we re-assign all its tuples to the node clock-wise after this one. To **add a node**, we add a new marker, re-assigning all tuples which now belong to the new node. (only repartitioning minimal set of nodes)
- **Simple replication strategy:** replicate a tuple in the next few additional nodes clockwise after the primary node used to store it.
- **Multiple markers:** we can also have multiple markers per node. For each tuple, we still assign it to the marker nearest to it in the clockwise direction. (Basically add more markers within ranges)
  - **Benefit:** when we remove a node, its tuples will not all be reassigned to the same node. So, this can balance load better.

Query Processing in NoSQL



- **Partition Pruning example:** when reading a specific value of shard key, config server can determine that the query only goes to one shard (the one that contains that value of the shard key). —— Same for writing
- But if the query is based on a key other than the shard key, it is relevant to all shards, and thus will go to all shards



1. Query is issued to a router (`mongos`) instance
2. With help of config server, `mongos` determines which shards to query → router decides which shards are relevant
3. Query is sent to relevant shards (*partition pruning*)
4. Shards run query on their data, and send results back to `mongos`
5. `mongos` merges the query results and returns the merged results to the application

Replication in MongoDB

Common configuration: 1 primary, 2 secondaries.

- **Writes:**
  - Primary receives all write operations
  - Records writes onto its operation log
  - Secondaries replicate this operation log, apply it to local copies of their own data (synchronizing data) then **ack** the operation
- **Reads:**
  - User can configure read preference, decides whether we read from secondary (default) or primary
  - Allow reading from secondaries can decrease latency and distribute load (improve throughput), but allows for reading stale data → due to eventual consistency
- **Elections:** If primary node fails, nodes conduct and election (protocol to choose which secondary promotes to be primary)

Reasons for Scalability and Performance of NoSQL

- **Horizontal Partitioning:** as we get more and more data, we can partition it into more shards, even if individual tables are very large. (improves speed due to parallization)
- **Duplication:** or denormalization unliked relational DB, queries require looking up multiple tables, duplication in NoSQL allows queries to only go to 1 collection.
- **Relaxed consistency guarantee:** prioritize availability over consistency (might return stale data).

Designing MapReduce

⋆ **Key mapper should emit:** key used to group data for reducers
⋆ **Value mapper should emit:** should give reducers all the information they need to perform the task
⋆ **Evaluating efficiency:** main considerations: disk & network I/O, determined by amount of data emitted by mappers (reduced by combiners), memory working set (mapper's intermediate state).

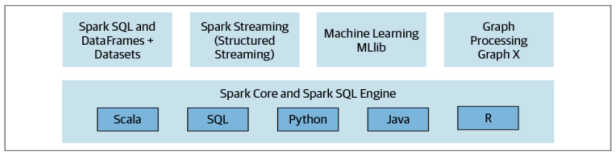
Spark Motivation

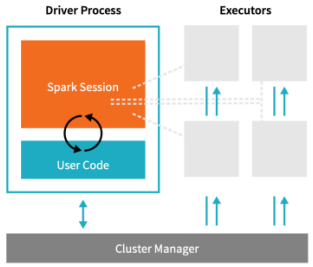
Issues with Hadoop MapReduce

- **Network and diskI/O costs:** intermediate data has to be written to local disks, and shuffled across machines
- **Not suitable for iterative processing**, each individual step modelled as 1 MR job.

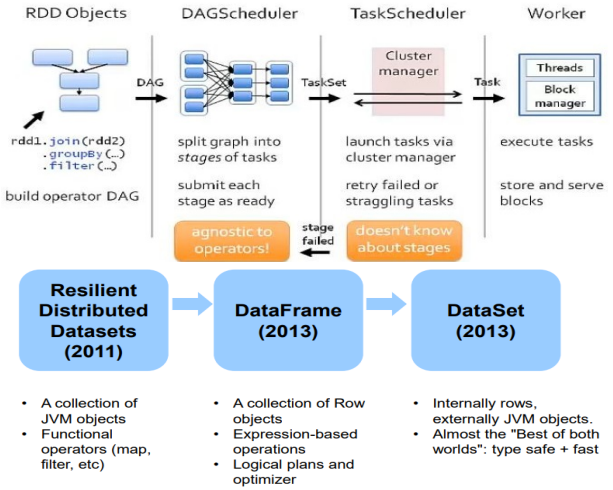
Spark stores much of the intermediate results in memory → much faster, when insufficient then **spills to disk (need I/O)**. Spark is also much easier to program compared to Hadoop.

Spark Components and Architecture



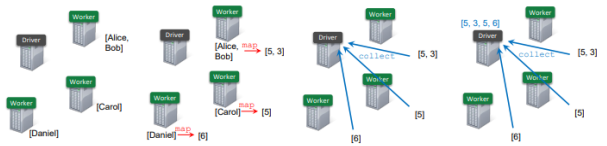


- **Driver Process** responds to user input, manages Spark applications and distributes work to **Executors** (processors) run code assigned to them and sends results back to driver
- **Cluster Manager** (YARN, Mesos, Spark's, K8s) allocates resources when requested
- In **local mode** all processes run on the same machine. Worker nodes can have multiple executors.



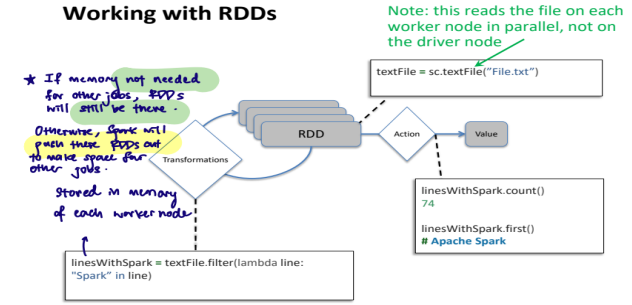
Resillient Distributed Datasets (RDDs)

Achieve fault tolerance through **lineages**. Represent a collection of objects that is **distributed over** machines. Transformations and actions are executed in parallel, results only sent to driver in final step.



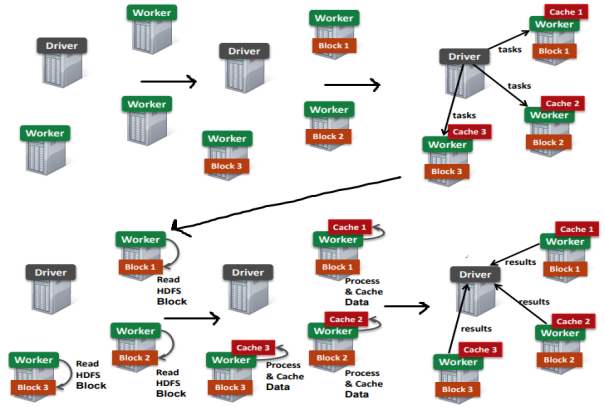
- **immutable**, cannot be changed once created.
- **Transformations** are a way of 'changing' RDDs to another.
  - Represents transformation mapping string → length
  - Transformations are lazy, they are not executed yet unless an **action** is called on it. **Advantage:** Spark optimizes the query plan to improve speed (by removing uneeded operations).
  - e.g. `map, order, groupBy, filter, join, select`
- **Actions** trigger Spark to compute a result from a series of transformations.
  - `collect()` here is an action, retrieving all elements of RDD to driver node
  - e.g. `show, count, save, collect`

```python
# RDD of names distributed over 3 partitions
dataRDD = sc.parallel(["Alice", "Bob",
    "Carol", "Daniel"], 3)

# Transformations
nameLen = dataRDD.map(lambda s:len(s))

# Actions - [5, 3, 5, 6]
nameLen.collect()
```



Spark Cache

A singular RDD is stored in a distributed manner on different workers

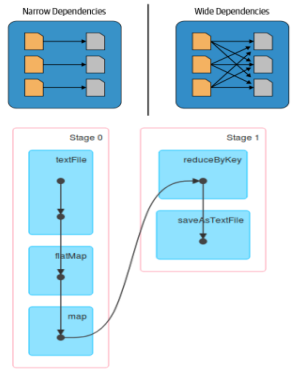



```python
lines = sc.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR")↩
    )
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache() # removes need to recompute

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

- `cache()`: saves an RDD to memory of each worker node
- `persist(options)`: save RDD to memory/disk/off-heap memory
- Cache when expensive to compute, or needed multiple times.
- If worker nodes have insufficient memory, they evict LRU RDDS.

Directed Acyclic Graph (DAG)

Internally, Spark creates a DAG representing all the RDD objects and **how they will be transformed**. Transformations construct this graph; actions trigger computations on it.




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

Narrow and Wide Dependencies

- **Narrow:** each partition of the parent RDD used by at most 1 partition of child RDD (`map, flatMap, filter, contains`)
- **Wide:** opposite of narrow, each partition of parent RDD used by multiple child partitions of RDD (`reduceByKey, groupBy, orderBy`)
- Consecutive narrow dependencies grouped together as **stages**.
  - **Within stages:** Spark performs consecutive transformations on the same machines.
  - **Accross stages:** Data needs to be **shuffled** i.e. exchanged across partitions. (similar to MapReduce) involving writing intermediate results to disk.
  - Minimizing **shuffling** is a good practice for improving performance.

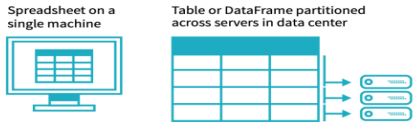
Narrow and Wide Dependencies

Unlike Hadoop, Spark does not use replication to allow fault tolerance.

- Spark tries to store all data in memory not disk. Memory capacity is much more limited than disk, simply duplicating all data is expensive.
- **Lineage approach:** if worker node goes down, replace it by a new worker node. Use the DAG to recompute data in the lost partition.

DataFrames and Datasets

- **DataFrames:** represents a table of data similar to SQL tables/DataFrames in pandas.
- Compared to RDDS, this is a higher level interface, has transformations that resemble SQL operations
  - DataFrames/Datasets recommended interface for working with Spark —— easier to use than RDDs and almost all task can be done with them.
  - DataFrames operations still ultimately compiled down to RDD operations by Spark.



DataFrames

DataFrames is expression based compared to functional based in RDDs (our code affects the efficiency of the program), we just need to tell Spark what we want it to do:

```
# Reading in a DataFrame from CSV file
flights = spark\.read\.option("inferSchema", true)\
  .option("header", true)\
  .csv("/mnt/defg/flight-data/csv/2015-flights.csv")

# Sorts by count and outputs first 3 rows (action)
flights.sort("count").take(3)

# Output: Array([United States, Romania, 15],
  [United States, Croatia ...])
```

We can also use SQL queries to transform DataFrames. (Takes DataFrame → returns DataFrame).

```
# Create a Table View first
flights.createOrReplaceTempView("2015-flights")

maxSql = spark.sql(
"""
SELECT DEST_COUNTRY_NAME, SUM(count), as DEST_TOTAL
FROM 2015-flights
GROUP BY DEST_COUNTRY_NAME
ORDER BY SUM(count) DESC
LIMIT 5
""")
maxSql.collect()
```

OR we can run the previous query with the `pyspark.sql.functions` library. Generally these functions take in either `strings` or 'column objects' representing columns.

```
from pyspark.sql.functions import desc

flights\.groupBy("DEST_COUNTRY_NAME")\.sum("COUNT")
  \.withColumnRenamed("SUM(count)", "DEST_TOTAL")
  \.sort(desc("DEST_TOTAL"))
  \.limit(5)
  \.collect() # action
```

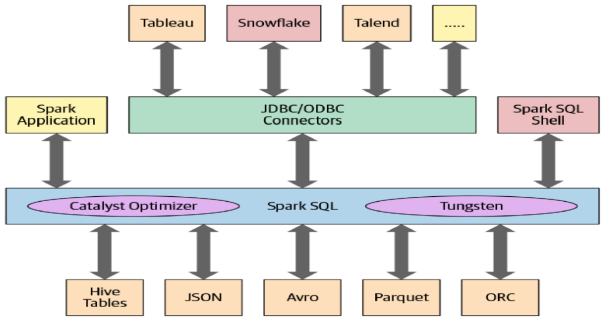
Datasets

Similar to DataFrames, but are type-safe. In Spark(Scala) DataFrame is just an alias for Dataset[Row]. Datasets are **NOT** available in `Python` and `R` since these are dynamically typed languages.

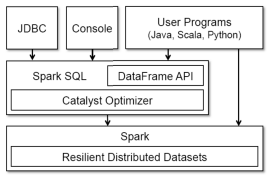
- Dataset `flights` is type safe —— it is of the Flight class
- Now when calling `collect()` it will also return objects of the 'Flight' class instead of Row objects.

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

val flightsDF = spark.read
  .parquet("/mnt/defg/flight-data/csv/flights.csv")
val flights = flightsDF.as[Flight]
flights.collect()
```

Spark SQL



- Unified Spark components and permits abstraction to DataFrames/Datasets in Java, Scala, Python, R.
- Keep tracks of schema, support optimized relational operations.



RDD vs DataFrame

- **RDD:** Instructs Spark how to compute
  - Intention is to be completely opaque to Spark
  - Spark does not understand the structure of data in RDDs (arbituary Python objects) or semantics of user functions (contain arbituary code)
- **DataFrame:** Tells Spark what to do, instead of HOW
  - Code more expressive + simpler, using Domain specific language (DSL). High level DSL operators to construct query.
  - Spark can inspect or parse this query, understand our intention. It optimizes it or arrange operations for efficient execution.

RDD vs. DataFrame

- RDD

```
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average

agesRDD = (dataRDD
  .map(lambda x: (x[0], (x[1], 1)))
  .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
  .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

- DataFrame

```
# Create a DataFrame
data_df = spark.createDataFrame([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)], ["name", "age"])
# Group the same names together, aggregate their ages, and compute an average
avg_df = data_df.groupBy("name").agg(avg("age"))
# Show the results of the final execution
avg_df.show()
```

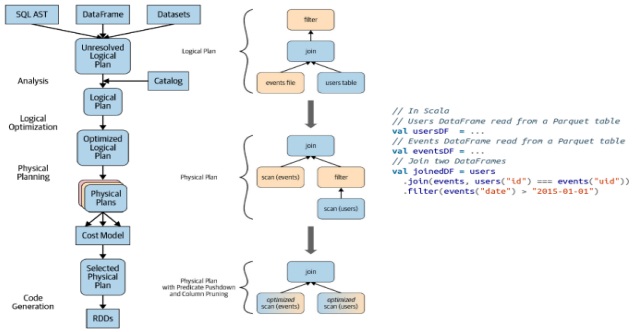
```
+------+--------+
|  name|avg(age)|
+------+--------+
|Brooke|    22.5|
| Jules|    30.0|
|    TD|    35.0|
| Denny|    31.0|
+------+--------+
```

→ more readable

The Catalyst Optimizer

Takes a computational query and converts it into an execution plan through four transformational phases:

1. Analysis
2. Logical optimization
3. Physical planning
4. Code generation

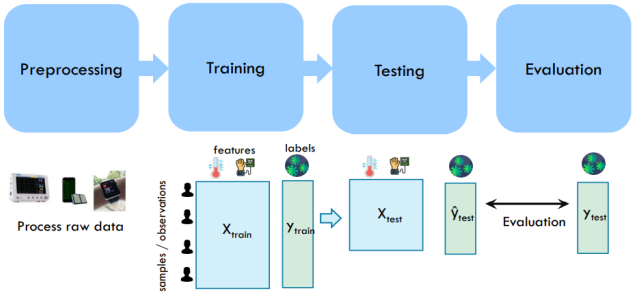


Benefit of Logical Plan is the performance parity across languages. Logical plans across all languages generates the same physical plan, thus same perfomance across all languages.

Machine Learning with SparkML

- **Classification:** *Categorize* samples into classes, given training data
- **Regression:** Predict *numeric* numbers, given training data

Typical Machine Learning Pipeline



Preprocessing

Data Quality —— Missing Values

- **Why?** Information not collected —— people declined to share
  - *Missing at random:* missing values are randomly distributed. If data is instead missing not at random, the missingness itself may be important information.
- **How to handle missing values?**
  - Eliminate objects (rows) with missing values.
  - OR Fill in the missing values:
    - based on **median / mean** of attribute

* **regression** model to predict value given other attributes
* **Dummy Variable:** optionally insert a column: 1 if variable missing, 0 if not missing

```
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

Convert categorical features into numerical ones. Numerical features often assigned in a way that represents the *ordinal relationship / inherent order* among categories.

- e.g. Risk rating [Low, Medium, High] → [0,1,2] .
- Application of algorithms that handle numerical features (LR).

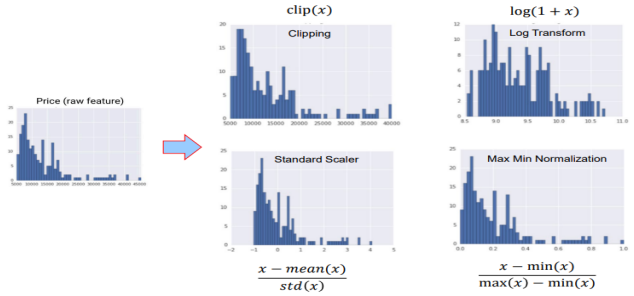
One Hot Encoding

Convert discrete features to series of binary features.

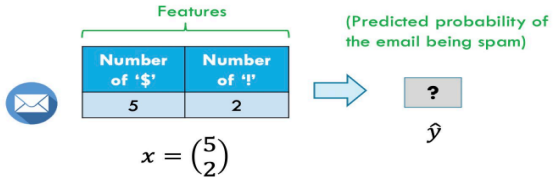
- e.g. first record has group 2, set second binary feature to 1, rest to 0
- Useful when there is **no ordinal relationship** among categories, ensures that categorical variable does not imply any relationship.

Normalization

Sensative to the range of input features (0-1, or 1-10000), hard to converge hence we need to normalize to scale features to similar ranges.

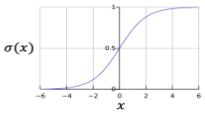


Training/Testing —— Spam classification example



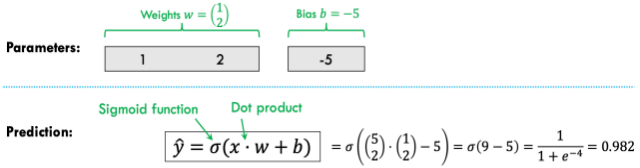
Logistic Regression —— Sigmoid Function

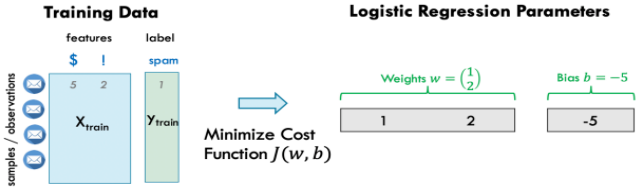
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



As $x \to -\infty, \sigma(x) \to 0$
As $x \to \infty, \sigma(x) \to 1$
If $x == 0, \sigma(0) = 0.5$

Sigmoid function $\sigma(x)$ maps real numbers to range $(0, 1)$. Defined ↑

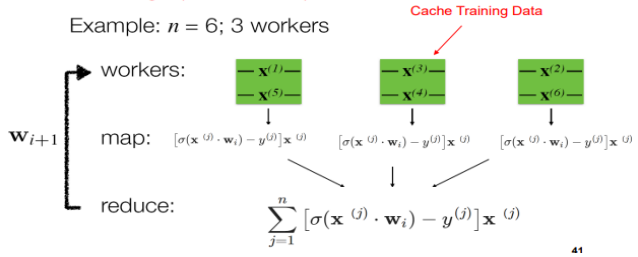




- Machine Learning involves the fitting ***parameters*** of a model (here w, b) by minimizing a ***loss / cost*** function.
- Here cost function is J ***Cross Entropy Loss***. → Model's prediction as a probability, if its closer to real value, lower loss value.

Vector Update: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \sum_{j=1}^{n} [\sigma(\mathbf{x}^{(j)} \cdot \mathbf{w}_i) - y^{(j)}]\mathbf{x}^{(j)}$

Weights (broadcast variable)



41

Evaluation

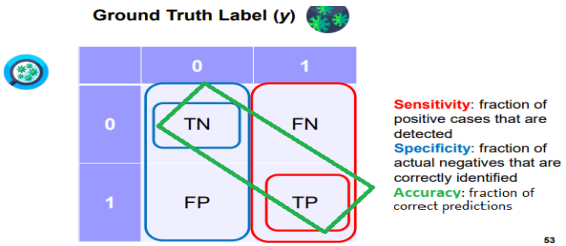
In a binary classification setting, we classify objects into 2 categories, and each category we classify them either correctly or wrongly. To evaulate our binary classification, we can use a **confusion matrix**.

True/False Positives/Negatives

True Positive	correctly output positive
True Negative	correctly output negative
False Positive	wrongly outputs positive
False Negative	wrongly outputs negative

Evaluation Metrics

Formulas	
Sensitivity	$\frac{TP}{TP+FN}$
Specificity	$\frac{TN}{TN+FP}$
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$



53

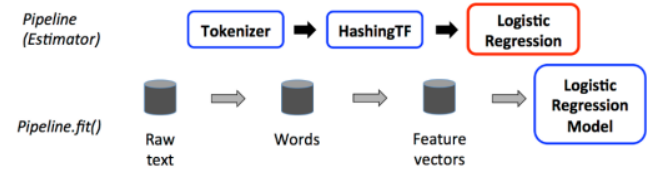
Evaluating Regression Model

- **Mean Absolute Error:** $MAE = \frac{1}{n}\sum_n^{i=1} |y_i - \hat{y}_i|$
- **Mean Squared Error:** $MSE = \frac{1}{n}\sum_n^{i=1} (y_i - \hat{y}_i)^2$
- **Root Mean Squared Error:** $RMSE = \sqrt{\sum_n^{i=1} (y_i - \hat{y}_i)^2}$
- **R Squared Value:** closer to 1, better model fits data
  1. $SS_{res} = \sum_i (y_i - f_i)^2 / \sum_i e_i^2$
  3. $\bar{y} = \frac{1}{n}\sum_n^{i=1} y_i$
  2. $SS_{tot} = \sum_i (y_i - \bar{y})^2$
  4. $R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$

Machine Learning Pipelines

Build complex pipeline out of simple building blocks: encoding, normalization, feature transformation, model fitting.

- Better code reuse: without pipelines, repeat alot of code, between training/test pipelines, cross-validation, model variants etc.
- Easier to perform cross validation, hyperparameter tuning.



Transformers

mapping DataFrames to DataFrames: e.g. one-hot encoding, tokenization. Transformer object has `transform()` method to perform its transformation.

- Transformers output a new DataFrame which ***append*** their result to the original DataFrame.
- Similarly a fitted model (e.g. logistic regression) is a Transformer that transforms a DataFrame into one with the predictions appended.

Estimator

Algorithm that takes in data and outputs a fitted model. For example, a learning algorithm (LogisticRegression object) can be fit to data, producing the trained logistic regression model.

- `fit()` method which returns a **Transformer**.

```
from pyspark.ml.classification import LogisticRegression
training = spark.read.format("libsvm")
  .load("data/mllib/sample_libsvm_data.txt")
lr = LogisticRegression(maxIter=10)
lrModel = lr.fit(training)

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

Training Time

Pipeline chains together Transformers + Estimators → ML workflow.

- Pipeline is an Estimator, when `Pipeline.fit()` is called:
  - Starting from beginning of pipeline:
  - Transformers it calls `transform()`
  - Estimators, it calls `fit()` to fit data and returns a fitted model

Test Time

Output of `Pipeline.fit()` is the estimated pipeline model (`type PipelineModel`)

- It is a transformer consisting of a series of Transformers
- When it calls `transform()`, each stage's `transform()` method is called

Streams

Data arrives over time, streaming approaches are designed to process their input **as it is received**.

- Input elements enter at a rapid rate from **input ports** receiving data from a sensor, from a TCP connection, from a file stream or message queue.
- Elements of a stream reffered to as tuples.
- Stream is potentially infinite; system cannot store the entire stream accessibly

Examples:

- data that is **high volume** and **constantly arriving over time**
- sensor, online user activity, financial transactions/stock trades data

Stateful Stream Processing

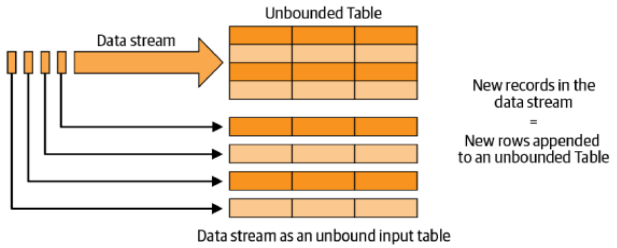
- not just performing trivial –transformations
- ability to store and access intermediate data
- state can be stored and accessed in many different places including program variables, local files or embedded / external databases

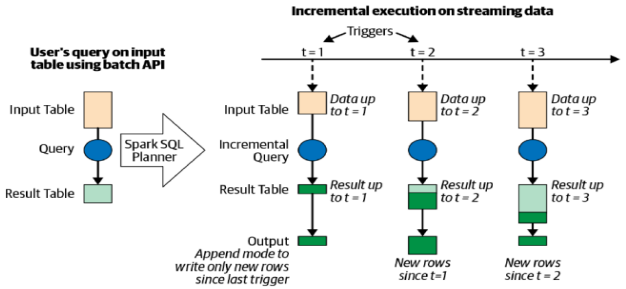
Micro-Batch Stream Processing

- Structured Streaming uses a micro-batch processing model
  - divides data from input stream in to micro batches
  - each batch processed in the Spark cluster in a distributed manner
  - small **deterministic tasks** generate output in micro-batches
- **Advantages:**
  - quickly and effectively recover from failures
  - deterministic nature ensures end-to-end exactly-once processing
- **Disadvantages:**
  - Latencies for a few seconds (usually OK for most apps)
  - Applications may incur more than a few seconds delay in other parts of the pipeline

Philosophy of Structured Streaming

- Writing stream processing pipelines similar to batch pipelines:
  - Single unifed programming model and interface for batch/stream processing
- Structured Streaming programming model: data stream as an unbounded table

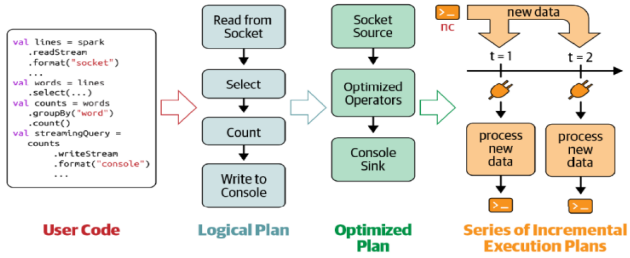




Users express query on streaming data using a batch-like API and Structured Streaming incrementalizes them to run on streams.

Defining a Streaming Query

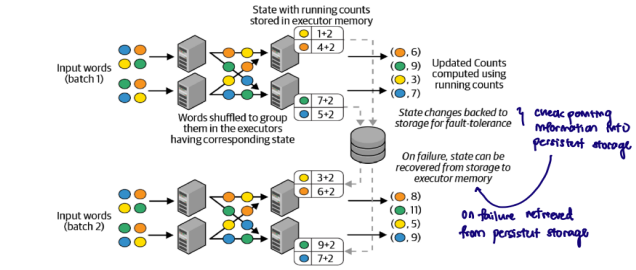
1. Define input sources
2. Transform data
3. Define output sink and output mode:
   - Output writing details (where, how to write output)
   - Processing details (how to process data, how to recover from failure)
4. Specify processing details:
   - **Triggering details:** when to trigger discovering, processing of newly available streaming data (e.g. size of micro-batch, after ever X records)
   - **Checkpoint location:** store streaming query process info for failure recovery (fault tolerance)
5. Start the query



Data Transformation

- **Stateless Transformation:**
  - Process each row individually, no need info from previous row
  - Projection operations: `select()`, `explode()`, `map()`, `flatMap()`
  - Selection operations: `filter()`, `where()`
- **Stateful Transformation:**
  - `DataFrame.groupBy().count()`
  - in every micro-batch, incremental plans adds count of new records to previous count generated by previous micro-batch
  - Partial count communicated between plans is the *state*
  - *state* maintained in memory of Spark executors, checkpointed to configured location to tolerate failures

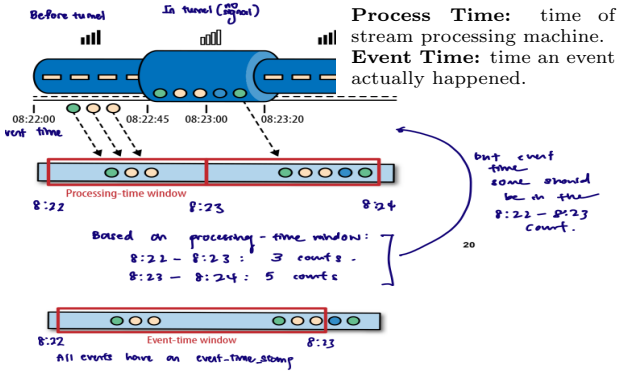
Distributed State Management in Structured Streaming



Stateful Streaming Aggregation

- 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:
  - `sum()`, `mean()`, `stddev()`, `countDistinct()`, `collect_set()`, `approx_count_distinct()`

Time Semantics



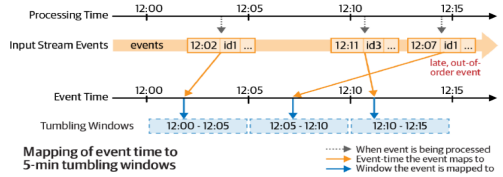
Process Time: time of stream processing machine.
Event Time: time an event actually happened.

- Event time decouples the processing speed from results
- Operations based on event time are **predictable** and results are **deterministic**
- Event time window computation will yield same result no matter how fast the stream is processed/when events arrive at operator!

Aggregations with Event-Time Windows

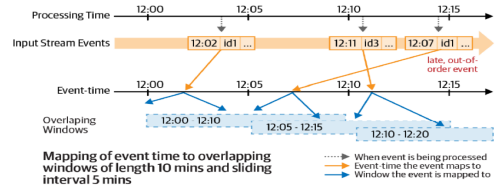
- **Tumbling Window:** No overlap in windows

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

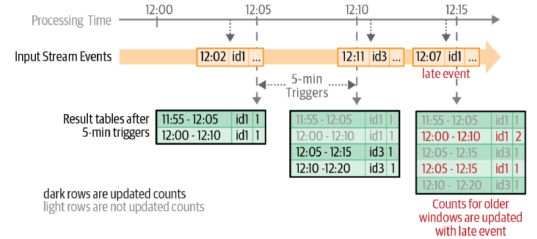


- **Sliding Window:** Overlapping window length

```
(sensorReadings
  .groupBy("sensorId", window("eventTime", "10 minute", "5 minute"))
  .count())
```

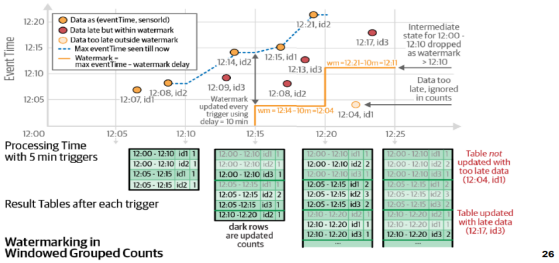


–
– Updating counts in result table after each 5-minute trigger:



- **Watermarks:** *Latest Event Time* – *Watermark Time*, drop packets that come before this time after event window has passed.

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



26

Performance Tuning

Besides tuniny Spark SQL Engine, other considerations include:

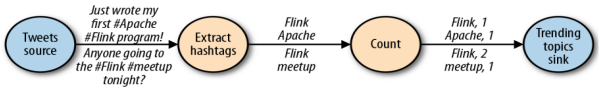
- Cluster resource provisioning appropriately to run 24/7
- Number of partitions for shuffles set much lower than batch queries
- Setting source rate limits for stability
- Multiple streaming queries in the same Spark application

Flink — Stream Native Processing!

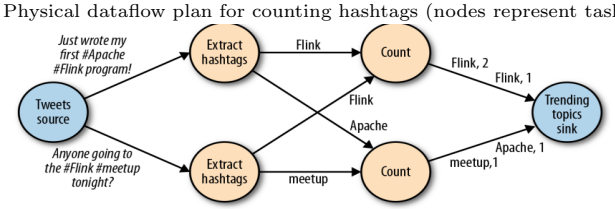
Event-driven streaming application, can think each of the processors as a streaming processing.

Dataflow Model

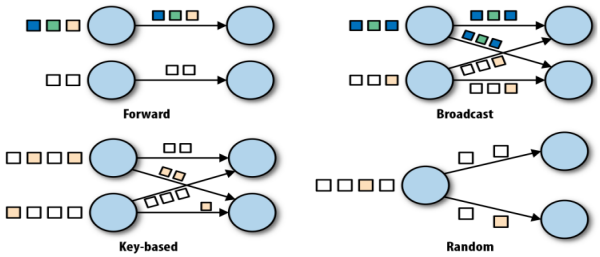
- Logical dataflow graph to continuously count hashtags (node are operators, edges are data dependencies)



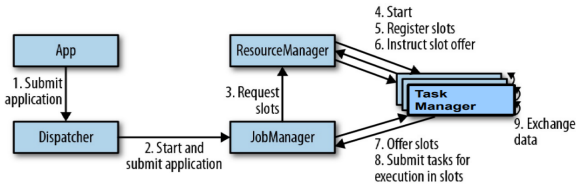
- Physical dataflow plan for counting hashtags (nodes represent tasks)



- Data Exchange Strategies



Flink System Architecture

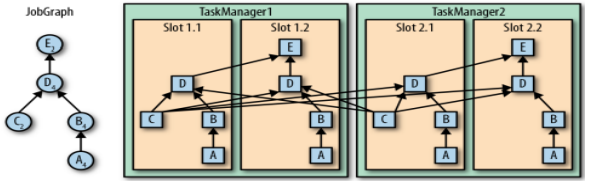


- Each task manager is similar to a worker node in Spark. Different messages handled by different task managers.
- Job Manager similar to the Driver in Spark.

Task Execution

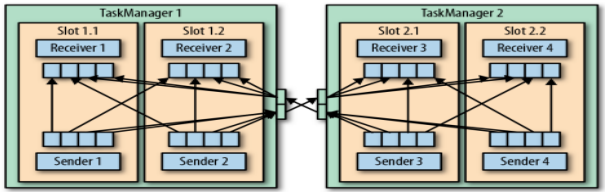
Task manager can execute the following at the same time:

1. number at the bottom indicates no. of parallel processing
2. distributed storage / computation taken advantage of, `A` and `B` in the same slot does not need to be shuffled
3. `C` and `D` are a `JOIN` occupation, thus require shuffling



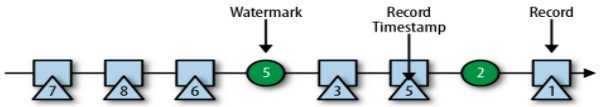
1. tasks of same operator (data parallelism)
2. tasks of different operators (task parallelism)
3. tasks from different applications (job parallelism)
4. **Task Manager:** offers certain number of processing slots to control number of tasks it is able to execute concurrently.
5. **Processing slot:** can execute one slice of an application — one parallel task of each operator of the application.

Data Transfer in Flink



- Tasks running application continuously exchanging data
- TaskManagers take care of shipping data from sending to receiving tasks asyncrhonously(ideally in the same slot A→B)
- Network component of a TaskManager collects records in buffers before they are shipped.

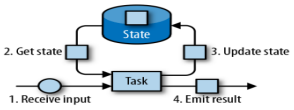
Event Time Processing

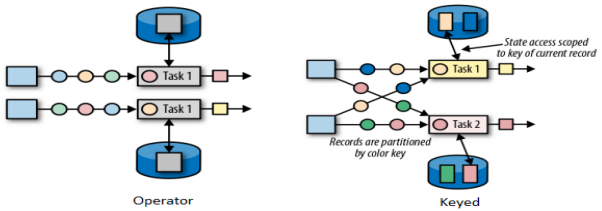


- **Timestamps:** every record must be accompanied by a timestamp
- **Watermarks:**
  - Flink event-time application must also provide Watermarks
  - derives current event time at each task in an Flink
  - Watermarks are implemented as special records holding a timestamp as a Long value. They flow in a stream of regular records with annotated timestamps.

State Management

Stateful stream processing task, state comprises: *all data maintained by a task* and used to *compute results of a function.*

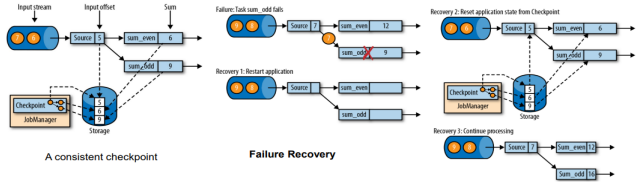




- **Operator State**: scope to operator task.
  - All records processed by same parallel task access to same states
  - Operator state cannot be accessed by another task of the same / different operator
- **Keyed State:** Maintains one state instance per key-value. Partitions all records with same key to operator task that maintains state for this key.
- **State Backend**
  - *Local State Management:*
    - task of a stateful operator typically reads/updates its states for each incoming record.
    - each parallel task locally maintains its state in memory to ensure fast state accesses.
  - *Checkpointing:*
    - TaskManager may fail at any point, hence its storage must be considered **volatile**
    - Checkpointing of state of a task to **remote + persistent storage** (can be distributed file system / Database)

Checkpoints

Consistent checkpoints: similar to Spark micro-batch checkpoints. Cannot achieve short delays, if application needs real-time processing, this would not work for it. However it is reliable.

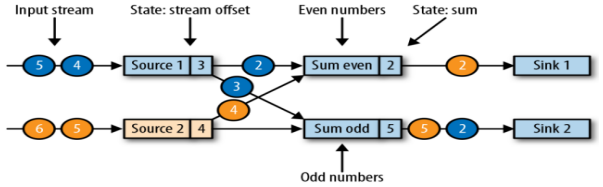


1. Pause ingestion of all input streams
2. Wait for all in-flight data to be completely processed, (all tasks processed input data)
3. Take a checkpoint by copying state of task to **remote, persistent storage.** Checkpoint is complete when all tasks finished copying
4. Resume ingestion of all streams
5. **Failure Recovery:**
   - Restart the whole application
   - Reset all states of stateful tasks to latest checkpoint
   - Resume all processing of tasks

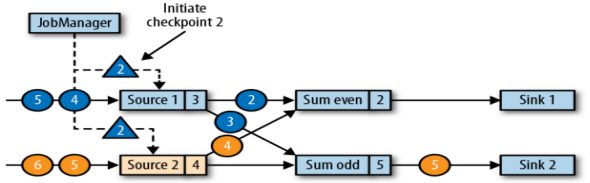
Flink's checkpointing Algorithm:

1. based on Chandy-Lamport algorithm for distributed snapshots
2. no pausing application, decouples checkpointing from processing
3. some tasks continue processing while others persist their state
   - uses special record called a *checkpoint barrier*
   - checkpoint barriers injected by source operators into regular stream, **cannot be overtaken / bypassed** by other records
   - carries a checkpoitn ID to identify checkpoint it belongs to and logically splits a stream in to 2 parts
   - all state modification due to records tha precede a barrier are included in the barrier's checkpoint; all modifications due to records following the barrier included in a later checkpoint

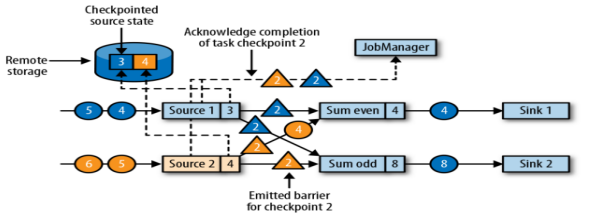
- Streaming application with two stateful sources, two stateful tasks, and two stateless sinks



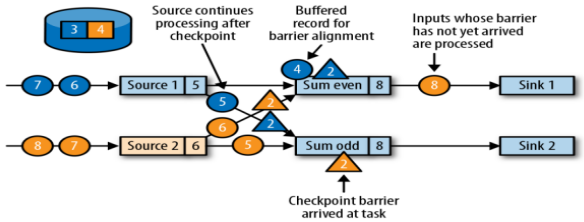
- JobManager initiates a checkpoint by sending a message to all sources



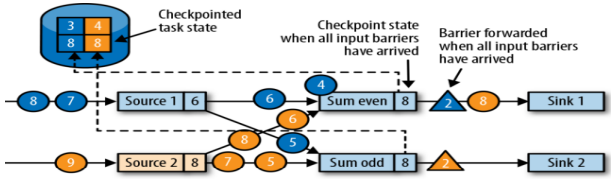
- Source checkpoint their state and emit a checkpoint barrier



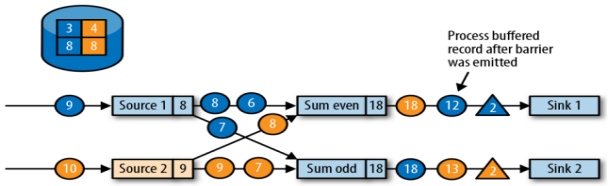
- Tasks to wait receive barrier for each input partition. Records from input streams after which a barrier already arrived are buffered. All other records are regularly processed.



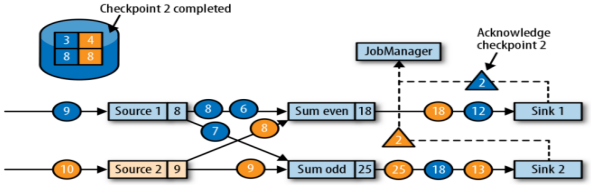
- Tasks checkpoint their state once all barriers recevied, then forward the checkpoint barrier.



- Tasks continue regular processing after barrier forwarded



- Sink acknowledges reception of checkpoint barrier to JobManager. Checkpoint is complete when all tasks have acknowledged successful checkpointing of their states.



Spark vs Flink

- **Spark:** micro-batch processing (few seconds latency). Checkpoints for each micro-batch **done synchronously**. Watermark: configuration to determine when to drop late events.
- **Flink:** real-time stream processing (few milliseconds latency). Checkpoints done distributedly in a **asynchronous manner** (more efficient → lower latency). Water mark is a special record to determine when to trigger event-related results.
  - Flink uses late handling functions (related to watermark) to determine when to drop late events.

Graphs

We use graphs to allow us to see the **relationships** rather than individual data of a node.

- **Node:** represent objects
- **Edges:** represents relationships, can be *undirected* or *directed*

Page Rank Algorithm

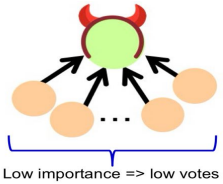
Web as a graph, nodes are our webpages, edges are our hyperlinks.

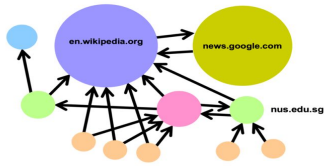
- All web pages are not equally *important*
- Measuring importance of pages necessary for web-task

Links as votes

Page is more important if it has in-links → assuming in-links harder to manipulate as anyone can outlink their page from wikipeida.

- **Naive solution:** Rank page based on *number* of in-links. → malicious users can create many dummy pages to up its rank!
- **Solution:** make page's votes proportional to its own importance, if dummy pages have low importance, they contribute less to votes.
  - Links from important pages count more —— recursive definition.
  - Recursively defines importance of a page based on the importance of the pages linking to it.





Voting formula

Each link's vote proportional to the importance of its source page.

- For each page j define its importance/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 votes on its in-links
  - Analogy: each page receives a certain amount of candies from its incoming neighbors. It distributes these candies evenly to its outgoing neighbors.

$r_j = r_i/3 + r_k/4$

Importance of node j

Importance node j receives from its incoming edges

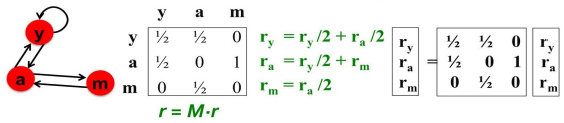
$$r_j = \sum i \to j \frac{r_i}{d_i}$$

- d_i number of out-links
- Sum of importance of pages linking to j, each divided by number of out-links
- r_j importance of j

3 equations, 3 unknowns, no unique solution. To force uniquessness of importance, we can add this constraint of $r_y + r_a + r_m = 1$. Solving these via substitution / Gaussian elimination works for small examples, need a new formula for large web-sized graphs.

Matrix formula

- Stochastic adjacency matrix **M**
  - Let page i has d_i outlinks, if $i \to j$, then $M_{ji} = \frac{1}{d_j}$ else $M_{ji} = 0$
  - **M** is a **column stochastic matrix**, columns sums to 1
- **Rank vector** r:vector with an entry per page
  - r_i is the importance of score of page i
  - $\sum_i r_i = 1$
- Flow equations can be written $\mathbf{r = M \cdot r}$
- Column headers refer to source node
- Row headers are where source nodes points to



Power Iteration Method

Given a web graph with n nodes, where the nodes are pages and edges are hyperlinks. **Interpretation:** each node starts with equal importance of $\frac{1}{N}$. During each step, each node passes its equal importance along its outgoing edges to its neighbours.

Power iteration: a simple iterative scheme

- Suppose there are N web pages
- Initialize: $\mathbf{r}^{(0)} = [1/N,\ldots,1/N]^T$
- Iterate: $\mathbf{r}^{(t+1)} = \mathbf{M} \cdot \mathbf{r}^{(t)}$ — Equivalent! → $r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$
- Stop when $|\mathbf{r}^{(t+1)} - \mathbf{r}^{(t)}|_1 < \varepsilon$

d_i .... out-degree of node i

$|x|_1 = \sum_{1 \le i \le N} |x_i|$ is the L1 norm
Can use any other vector norm, e.g., Euclidean

Random Walk Interpretation

Imagine a random web surfer:

1. At time $t = 0$, surfer starts on a random page
2. At any time t, surfer is on page i
3. At time $t + 1$, surfer follows an out-link from i uniformly at random
4. Process repeats indefinitely

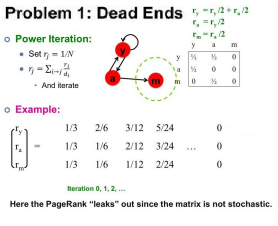
- Let $p(t)$ ... vector whose i^{th} coordinate is the prob. that surfer is at page i at time t so $p(t)$ is a probability distribution over pages
- Thus at $t + 1$, $p(t + 1) = M \cdot p(t)$

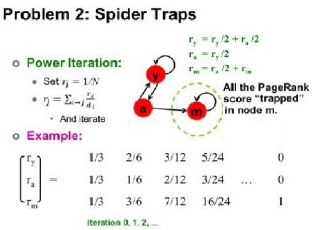
Stationary Distribution: as $t \to \infty$, probaility distribution approaches a tsteady state representing the long term probability that a random walker is at each node (PageRank scores)

PageRank with Teleports

Does $r_j = \sum i \to j \frac{r_i}{d_i}$ or equivelently $r = Mr$ converge, converge to what we want and are the results reasonable? Not always due to:

1. **Deadends:** no out-links, causes importance to *leak*
2. **Spider Traps:** all outlinks within a group, stuck in trap and *absorbs importance*





Solution (Spider Traps): Each time step, surfer has 2 options:

- Prob. β follow a link at random OR $1 - \beta$ jump to a random page
- Common values for β rand between $0.8 - 0.9$
- Never get stuck in a spider trap by teleporting out of it with a finite number of steps

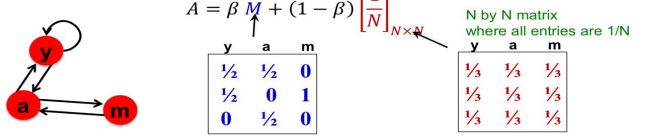
Solution (Deadends): If at deadend, always teleport

- Follow random teleport links with probability 1.0 from deadends
- Eventually, for each deadend m we can preprocess the random walk matrix M by making m connected to **every node** (including itself)
- Make matrix column stochastic by always teleporting when at a dead end, since matrix column was not stochastic

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N}$$

Teleport term ($(1-\beta)\frac{1}{N}$) is added, if any deadends exists we assume to have **already preprocessed** them by added connections from them to every other node. The rest is similar to simplified PageRank.

- We can also write this as a matrix equation, by defining the **Google Matrix *A***:



- **PageRank equation (matrix form): $r = A \cdot r$**
- In practice β = 0.8, 0.9 (5-10 steps on average before teleport)

Problems with PageRank

- Measures generic popularity of page (does not consider specific topics) → Topic-Specific PageRank
- Uses single measure of importance (less dimensions) → Hubs-and-Authorities
- Susceptible to Link spam, artifically boost page rank → TrustRank

Topic-Specific PageRank

Evaluating Web pages according to how close they are to a particular topic.

- Allow search queries to be answered based on interests of user
- Random walker has small probability of teleporting at any step
  - **Standard PageRank:** Any page with equal probability (to avoid spider traps/deadends)
  - **Topic Specific PageRank:** Topic specific set of relevant pages

$$A_{ij} = \beta M_{ij} + (1-\beta)\frac{1}{|S|} \rightarrow \ if\ i\ \in S$$

$$A_{ij} = \beta M_{ij} + 0 \rightarrow \ otherwise$$

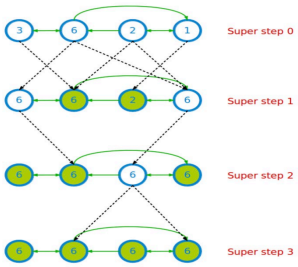
PageRank Implementation

Characteristics of Graph Algorithms

- Local computations at vertex + passing messages to other vertex
- Algorithms implemented from view of single vertex
  - User only implements a single function `compute()` to describe the algorithm's behaviour in this step
  - Abstracting away scheduling/implementation details

Pregel: Computational Model

- Computations consists of a series of **supersteps**
- Each superstep, framework invokes **user-defined function**, for each vertex (parallel)
- `compute()`:
  - **read messages** sent to v in superstep s - 1
  - **send messages:** to other vertices reading in superstep s + 1
  - **read/write value:** of v and value of outgoing edges



- Termination: vertext can choose to deactivate itself, woken up if new message received
- Computation halts when all vertices are inactive.

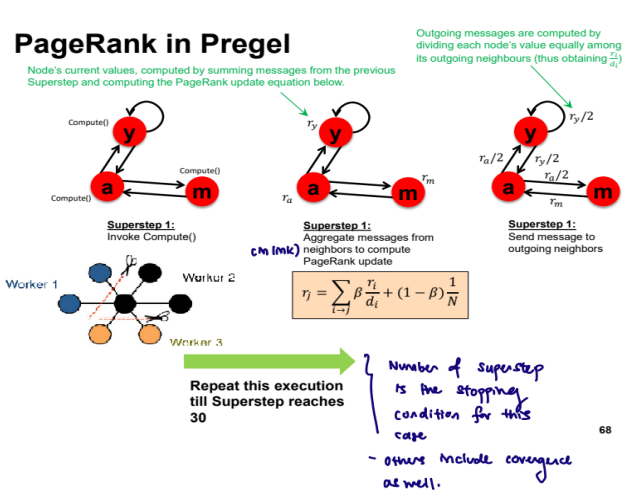
```
def compute(v, messages):
    changed = False
    for m in messages:
        if v.getValue() < m:
            v.setValue(m)
            changed = True
    if changed:
        for neighbour in v.neighbours:
            sendMessage(w, v.getValue())
    else:
        voteToHalt()
```

Other Graph Processing Projects

- **Giraph:** Open source implementation of Pregel (Facebook)
- **Spark GraphX/GraphFrame:**
  - Extends RDDs to Resilient Distributed Property Graphs
  - Join Vertex Table and Edge Table to capture the relationship
- **Neo4j:**
  - Graph database + Graph processing
  - SQL like interface: Cypher query language

Pregel: Implementation

- Vertices are hashed/partitioned and assigned to workers (edge cut)
- Each worker maintains state of its portion of the graph **in memory**
- In each superstep, each worker loops through its vertices and executes `compute()`
- Messages from vertices are sent, either to vertices on same worker or to vertices on different workers
- Messages **buffered locally**, sent as a batch to reduce network traffic
- **Fault Tolerance:**
  - Checkpointing to persistent storage
  - Failure detected through heartbeats
  - Corrupt workers reassigned/reloaded from checkpoints



PageRank in Pregel

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

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

Compute sum of incoming messages

PageRank update

$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$

Send outgoing messages

Stop after fixed no. of iterations

Note: this algorithm implicitly assumes that the nodes' values (*MutableValue()) have been correctly initialized based on the initialization scheme that the user wants. In practice, you would generally do the initialization during superstep 0.

69

PYP Summaries

1. Diff. between split vs combined jobs (Hadoop)

- S1. By combining Job 1, Job 2 and Job3 to Job4, the amount of input disk I/O of Job 4 is smaller than the total amount of input disk I/O for Job 1, Job 2 and Job 3.
- S2. According to scalability analysis on the Map task, Job 4 has better task parallelism than other jobs (Job 1, Job 2 and Job 3).
- S3. According to network I/O analysis in the shuffling, the Shuffling stage of Job 4 has a smaller amount of network I/O than Job 3.

A: S1 is true as we avoid the need for disk I/O in between map-reduce jobs, e.g. between job 1 and job 2 (where the outputs are written to HDFS).

S2 is false: **Task parallelism** refers to the extent to which the job can be broken down into sub-tasks that can be executed in parallel. Since the map functions in both cases (either job 1-3 and job 4) are just filtering and emitting, both cases are easily parallelizable by breaking down the input into chunks of a certain size (e.g. 128MB).

S3 is false: the amount of data shuffled in job 3 and job 4 is the same (since the exact same operations are happening in both cases).

2. Topic Specific Page-Rank

Assume that we use topic-sensitive PageRank to obtain scores for webpages for the 'data science' topic. There is a set A of webpages, none of which are in the 'data science' seed set (i.e. teleport set) S. Assume that for each page in A, if it has any incoming links, these links can only come from other pages in A. Is it true that all pages in A have a topic-sensitive PageRank score of 0? Explain why or why not.

- Yes. All random walk paths from the seed set S start outside A and will never enter A, so the Topic Sensitive PageRank algorithm will assign 0 score to nodes in A.

In topic-sensitive PageRank, the node with the highest topic-sensitive PageRank score is always in the teleport set.

- False, A node could be outside the teleport set, but receive a lot of PageRank from its incoming neighbors.

3. Watermark library help

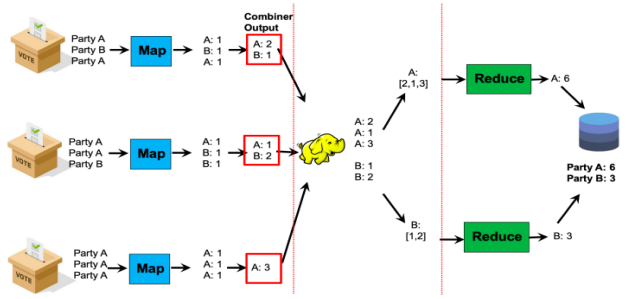
```
sensorReadings
.withWatermark("eventTime", "10 minutes")
.groupBy("sensorId",
    window("eventTime", "10 minutes", "5 minutes")
).count()
```

- `.withWatermark()`
  - `eventTime`: attribute in data representing the timestamp of when event occured
  - `10 minutes`: allowed lateness / maximum out-of-order time for events (Events that arrive up to 10 minutes after their event time still considered in processing.)
- `.groupBy()`
  - `sensorId`: attribute by how data is grouped
  - `window()`
    - `eventTime`: attribute in data representing the timestamp of when event occured
    - `10 minutes`: window size, considers events in the last 10 min
    - `5 minutes`: sliding interval of window, when trigger result table
    - the above function basically reads the window is of size **10 mins**, sliding every **5 minutes**. e.g. 12:10 - 12:20, 12:15 - 12:25 are 2 examples of windows and so on.

4. Implement multiple hash functions Jacc Sim.

```
hashes = set() // set of 100 hashes
def mapper(doc_id, doc):
    shingles = create_shingles(doc)
    for hash in hashes:
        minhash = 0
        for shingle in shingles:
            minhash = min (hash.hash(shingle), ↩
                minhash)
    emit((hash.hash_id, minhash), doc_id)

def reducer1((hash.hash_id, minhash), doc_ids):
    doc_pairs = generateDocPairs(doc_ids)
    for pair in doc_pairs:
        emit(pair, 1)

def reducer2(doc_pair, values):
    count = sum(values)
    emit(doc_pair, count / 100)
```

5. MapReduce Diagram



Find k-shingles (k=1) where it appears in document 0 and 1. Assume input is clean and use a combiner (not in-mapper combiner). Reducer output should be `<shingle, 1>`

```
map(docID, line):
  tokens = line.splt()
    for token in tokens:
      emit(token, docID)
reducer(token, docIDs):
  if 0 and 1 in docIDs:
    emit(token, 1)
combiner(token, docIDs):
  if 0 in docIDs:
    emit(token, 0)
  if 1 in docIDs:
    emit(token, 1)
```

- `map()` use tokens as keys, each `reduce()` only handles 1 word
- `combine()` combines multiple `(token, docID)` tuples into a single token, it does not change the final output of the reducer.
- combiner same input as reducer, if there are multiple 0s or 1s, no difference to output since we want **presence of words** not the total count.

6. Columnar vs Row based databases

- **Compression:** Compression speeds much faster on columnar databases, literally how data is compressed. Since columnar databases have uniform type, easier to compress as well.
- **Vectorized Processing:** Operations are applied to entire vectors or columns of data at once. Vectorized processing designed to optimize analytical queries and operations.

7. Spark Bottlenecks

```
Line 0: maxSql = spark.sql("""
Line 1: SELECT ip, sum(time) as total
Line 2: FROM traceA
Line 3: WHERE time>0.1
Line 4: GROUP BY ip
Line 5: ORDER BY sum(time) DESC''
Line 6:""")
Line 7: maxSql.collect()
```

1. Line 1 - 5, what are potential performance bottlenecks:
   - if data frame has been in the RAM, Line 3, the potential I/O cost by reading the trace ***OR*** size of data generated, Line 4/5 bottleneck due to network I/O of shuffling
2. Runs program for different trace, but operations in line 5 run slowly
   - Data skew cause stragglers, filter condition generates more result
3. Runs for other data input sets, some fail, identify fix for program
   - Line 7, returns too many results, add `LIMIT` to query

```
sales_df.join(products_df, 'product_id')
  .groupBy('product_id')
  .agg(sum('price'))
  .orderBy(sum('price'), ascending=0)
  .show(5)
```

- **Line 1:**
  - `JOIN` 2 tables, if both tables are super big (cannot fit into memory)
  - AND not already partitioned using `product_id`, can take awhile to partition 2 super large tables across different machines.
  - cause a lot of network shuffling, before sort and merging at each partition.
- **Line 3:**
  - If grouped data after line 2 highly skewed (many sales record for 1 `product_id`)
  - Line 3 will have task stragglers, certain tasks takes a longer time to complete
- **Line 4:**
  - Depending on the number of unique `product_id`, the global sorting (a wide transformation requiring network shuffling) done by orderBy can also be a bottleneck.

9. Spark reading a large log file

Ran the following Spark program to process a massive log file, on a large cluster. However, the program crashed. Explain why this is the case, and which line(s) you would modify to fix the problem.

```
Line 1: info = spark.textFile\(hdfs://"log)
Line 2: info = info.map(lambda s: s.split\( ")[2])
Line 3: info.collect()
Line 4: info.filter(lambda s: \"hadoop in s).count()
Line 5: info.filter(lambda s: \"spark in s).count()
```

`collect()` sends all the data to the master node, causing a crash as the dataset is too large to fit in memory. We can simply delete line 3, as it does not affect the results of the program.

8. True False Questions

A. NoSQL denormalization

NoSQL databases often use denormalization to duplicate data across multiple tables or documents as storage is cheap. This makes it easier to query the data, as all the necessary information is available in a single table / document.

B. Spark DAG

- Spark creates (DAG) to record the transformations into a few stages. Should try to avoid transformation across stages to improve performance.
  - : Across stage transformation (with wide dependencies) needs data to be shuffled across different servers (i.e. network shuffling). This is a very expensive operation and should be avoided.

C. Catalyst Optimizer in different languages

In this case (i.e. join function), all the languages will achieve similar performance through DataFrame API. API / Query language only provides a **logical plan**, and Catalyst Optimizer will generate the **optimized logical plan**, physical plan and RDD codes, which is the same no matter which API language you are using.

D. Pregel workers update vertices in the same worker only

Messages from each vertex are sent to all the neighboring vertices which can be on the same worker or on a different worker

E. Always put data into DRAM, since faster

DRAM is more expensive and limited than disk space, and is also not persistent. (Thus, for storing massive amounts of data which does not have to be accessed frequently, or for data we want to persist even when the server goes down, using disk is more appropriate).

F. Undirected Graph PageRank, no spider-traps?

True, spider-traps occur when random walks enter but cannot exit a part of the graph. This cannot happen for undirected graphs.

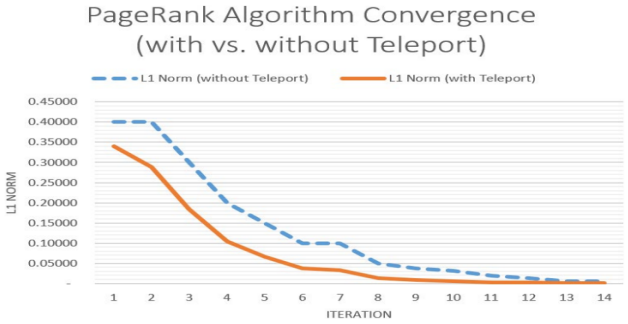
8. PageRank w/ and w/o Teleport

Rachel tried below two algorithms to compute the importance factors: PageRank with and without Teleport. She initialized all the important factors as $0.2(=\frac{1}{5})$ and then monitors the algorithms convergence performance using L1 Norm, i.e. the sum of the absolute values of the importance factor change for each user, denoted as below equation:

$$L1\ Norm = |I(A)^{t+1} - I(A)^t| + |I(B)^{t+1} - I(B)^t| + |I(C)^{t+1} - I(C)^t| +$$

$$|I(D)^{t+1} - I(D)^t| + |I(E)^{t+1} - I(E)^t|$$

Rachel plots the L1 norm curves by iterations for both algorithms in the below figure. She noticed PageRank with Teleport converges faster than PageRank without Teleport. Please help Rachel to figure out the reasons behind this and provide your explanations.



- Teleport is designed to solve the spider trap / dead ends problems.
- For this specific problem, there is No spider trap and No dead ends.
- However, Teleport **still helps the random walker to occasionally jump** to the less popular /important nodes so as to **spread out the importance factors** among all the nodes in the graph. This helps the algorithm to converge faster.
  - 0 marks if only talk about spider-trap/dead-ends
  - 2 marks if idea conveys that Teleport helps to **spread importance factors** among all nodes in graph

9. Java vs Hadoop HDFS

Jim use k-means on Hadoop cluster on 1 server. Performance is much slower than implementation written on Java from scratch.

- Hadoop incurs significant performance slowdowns due to disk I/O and network I/O (during the shuffle phase, we have to write data to disk and also send it across the network from mappers to reducers)

Jim runs task on 32 servers, finds that time taken by program does not significantly improve.

- Some **clusters are huge** (e.g. containing the majority of the data points), results in limited benefits of parallelism during the reduce stage (since the points in each cluster are only handled by 1 worker). OR
- The bottleneck may be the **latency of reading and writing from HDFS** (e.g. consider the case where the data size is small). As such, increasing the number of workers does not improve efficiency.

10. Data Centers for Spark

- **High-speed data network, higher bandwidth, lower latency**
  - Less aggressive IMC (In-mapper combiner).
  - More aggressive "move data to compute"
  - Cost plan adjustment in data partitioning.
  - Lower compression in data frame.
- **Have Advanced CPUs, more cores, higher clock freq:**
  - More aggressive compression.
  - More parallelism in a single node.
  - Query optimization by the more parallelism within a node.

Gradient Descent (optional)

Want to find w, b to minimize $J(w, b)$. Starting at an arbitrary point, move following the steepest downward slope (gradient) and continue until convergence.

- Given n training samples with d features:
  - Update Rule (from iteration i to iteration $i+1$)

$$w_{i+1} = w_i + \alpha \nabla J(w_i)$$

  - Weight Vector Update:

$$w_{i+1} = w_i - \alpha \sum_{n}^{j=1} [\alpha(x^j \cdot w_i) - y^j] x^j$$

Tutorial 2 — Summary

Pros/Cons of NoSQL, BASE

- **Basically Available:** Reading and writing operations are available as much as possible, but without consistency guarantees (e.g. read may not get the latest updated value).
- **Soft state:** The state of the system is always 'soft' or changing with inputs, until it reaches 'eventual consistency'.
- **Eventually consistent:** The system will eventually become consistent (e.g. multiple reads eventually return the same value).
- Pros:
  - Performance: NoSQL systems often sacrifice strong consistency while aiming for low latency and high availability (e.g., due to removing the need for locks and other concurrency mechanisms)
  - Scalability: NoSQL systems are straightforwardly horizontally scalable (due to sacrificing strong consistency)
- Cons:
  - Outdated data: outdated data may cause mistakes or require additional work to handle on the application side

Key Store vs Document Store vs Graph

- **Key-value store:**
  - Improves scalability and efficiency – writing or reading user pages is faster.
  - No need for complex queries or based on the content of user pages – just reads and writes.
  - May be acceptable for user pages to be slightly stale – then eventual consistency is acceptable
- **Document store:**
  - Flexible schema may be beneficial (e.g. special types of vehicles may require different sets of fields OR some students input more fields into their profile, others may not follow)
  - Unlike key-value stores, document stores are more suitable for queries based on fields of a document
- **Graph DB:**
  - could be useful if certain use-cases crucially involve relationships:
  - recommending courses to students, recommending study groups for students, managing complex prerequisites / time conflicts between courses to recommend a timetable, etc.

Tutorial 3 — Summary

Spark more iterative than Hadoop

- Spark stores most of its intermediate results in memory, making it much faster, especially for iterative processing

Resilliency of Spark/HDFS

In HDFS, each chunk is replicated for three times by default. In contrast, in Spark, RDD uses lineage for reliability. What is a major problem if Spark also uses replications for reliability?

- Consumes a lot of memory; memory is more scarce than disk space

True that in the Spark runtime, RDD cannot reside in the hard disk?

- False. RDD can also be in the disk if out of memory

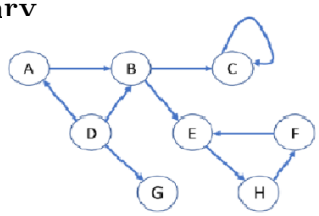
Explain how to speed up program:

```
1 lines = spark.textFile("hdfs://log")
2 errors = lines.filter(lambda s: s.startwith("INFO"))
3 info = errors.map(lambda s: s.splti("\t")[2])
4
5 #info.cache() -> add here
6
7 info.filter(lambda s: "hadoop" in s).count()
8 info.filter(lambda s: "spark" in s).count()
```

- add `info.cache()` (or `info.persist()`) before L4, to cache RDD in memory (or hard disk) so it doesn't have to be re-computed in L5.
- because of `count()`, an **action**, HDFS lods into memory and follows the DAG lineage.

Tutorial 5 — Summary

- How many deadends?
  - $\{G\}$ (no outlinks → decay rank)
- How many spider traps?:
  - $\{C\}$
  - $\{E, F, H\}$
  - $\{B, C, E, F, H\}$
  - $\{A, B, C, E, F, H\}$
  - All outlinks within group (absorb importance)

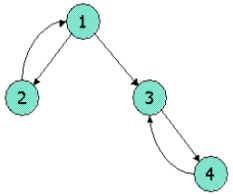


Random Walker

In Topic-specific PageRank, random walker will teleport to any page with equal probability.

- False. Random walker will only teleport to a topic-specific set of "relevant" pages.

Topic Specific Page Rank Guide



Write down the Topic-Specific PageRank equations for the following link topology. Assume that pages selected for the teleport set are nodes 1 and 2 (where teleports go to either node with equal probability). Assume further that the teleport probability, (1 - β), is 0.3.

Recall the topic sensitive pagerank (TSPR) equations:

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

Basically, the only difference here (compared to the non-topic sensitive case in the last 2 questions), is that the teleport terms are only distributed among the nodes in the teleport S, instead of being distributed over all nodes. Let r_1, r_2, r_3, r_3 denote the importance of the 4 nodes.

$r_1 = 0.7\, r_2 + 0.15$
$r_2 = 0.35\, r_1 + 0.15$
$r_3 = 0.35\, r_1 + 0.7\, r_4$
$r_4 = 0.7\, r_3$

Pseudo-code for compute()

`compute()` function for the PageRank with teleport ($\beta = 0.85$) over vertices algorithm in Pregel / Giraph. Set the initial PageRank value as 1/N (N is the number of vertices), Run 30 iterations and then stop.

```
1  compute(v, messages):
2    if getSuperStep() == 0:
3      v.setValue(1 / getNumVertices())
4    if getSuperStep() >= 1:
5      sum = 0
6      for m in messages:
7        sum += m
8      v.setValue(0.15 / getNumVertices() + 0.85 * sum)
9    if getSuperStep < 30:
10     sendMsgToAllEdges(v.getValue() / len(↩
           getOutEdgeIterator))
11   else:
12     voteToHalt()
```