CS4225 — BIG DATA SYSTEMS FOR DATA SCIENCE

PROF. BRYAN HOOI KUEN YEW

arsatis[*]

CONTENTS

[*]author: https://github.com/arsatis

1 INTRODUCTION

Lecture 1
12th August 2022

Challenges of big data:

- **Volume**: scale of data.
- **Variety**: different forms of data.
- **Velocity**: rate of data streaming.
- **Veracity**: uncertainty of data.

Data lifecycle:

1. **Ingest**:
   (a) Understand problem.
   (b) Ingest data.
2. **Transform**:
   (a) Explore & understand data.
   (b) Clean & shape data.
3. **Analyze**:
   (a) Evaluate data.
   (b) Create & build models.
4. **Output**:
   (a) Communicate results.
   (b) Deliver & deploy model.

1.1 *Cloud computing*

Recently, computer systems have moved on from traditional architectures (e.g., the figure on the left) to virtualized and containerized architectures, thus enabling the rise of cloud computing.

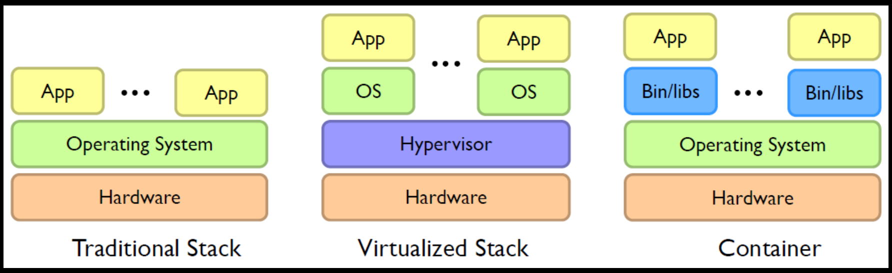


Figure 1: Enabling technology: virtualization & containerization.

Everything-as-a-Service (XaaS): with cloud computing, the vast number of products, tools, and technologies that are now delivered to users as a service over the internet, e.g.:

- **Infrastructure-as-a-Service (IaaS)** (a.k.a. utility computing):
  - Users rent computing cycles from cloud providers.
  - Providers only provide machines, and users are allowed to do anything they want (i.e., greater flexibility, but also requires user to have more knowledge of what they want to do).
  - E.g., Google Compute Engine, Google Colab, etc.
- **Platform-as-a-Service (PaaS)**:
  - Cloud providers provide hosting for web applications and takes care of maintenance.
  - Users have to provide their own content.
  - E.g., Google App Engine, Github pages, etc.
- **Software-as-a-Service (SaaS)**:
  - User does not have to implement anything by themselves, but simply use existing software features.
  - E.g., Gmail, Dropbox, Zoom.

1.2 *Data centers and data processing*

Terminology:

- **Bandwidth**: maximum amount of data transmitted per unit time.
  - When transmitting large amounts of data, bandwidth informs us the approximate time the transmission will take.
- **Latency**: time taken for 1 packet to travel from source to destination (i.e., *one-way*), or from source to destination and back (i.e., *round-trip*).
  - When transmitting very small amounts of data, latency informs us the approximate time the transmission will take.
- **Throughput**: rate at which data is *actually transmitted* across the network during some period of time.

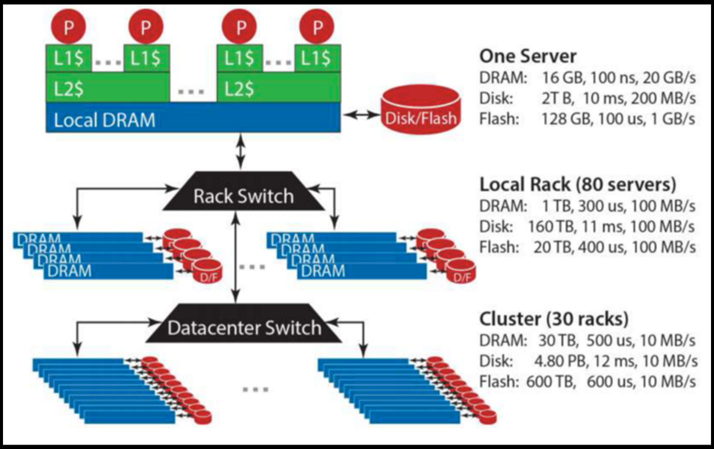

P
L1$
L2$
Local DRAM
Disk/Flash
One Server
DRAM: 16 GB, 100 ns, 20 GB/s
Disk: 2T B, 10 ms, 200 MB/s
Flash: 128 GB, 100 us, 1 GB/s
Rack Switch
Local Rack (80 servers)
DRAM: 1 TB, 300 us, 100 MB/s
Disk: 160 TB, 11 ms, 100 MB/s
Flash: 20 TB, 400 us, 100 MB/s
DRAM
D/F
Datacenter Switch
Cluster (30 racks)
DRAM: 30 TB, 500 us, 10 MB/s
Disk: 4.80 PB, 12 ms, 10 MB/s
Flash: 600 TB, 600 us, 10 MB/s

Figure 2: Storage hierarchy in data centers.

In data centers,

- From to local servers to clusters, *capacity increases*, but *latency increases* and *bandwidth decreases* (since communication costs increase).
- **Disk reads are expensive**, and are slower by orders of magnitude in both *latency* and *bandwidth*.

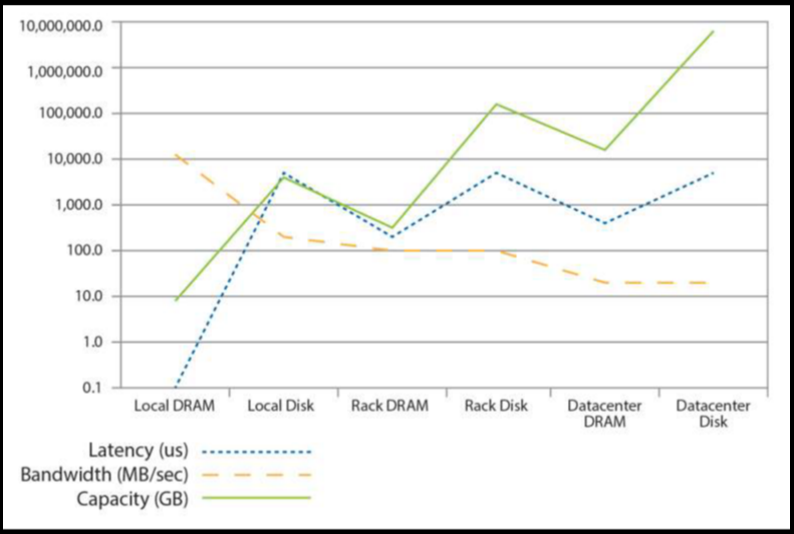


Figure 3: Cost of moving data around data centers.

Within servers, DRAMs are typically much faster than disks, but are also more expensive.

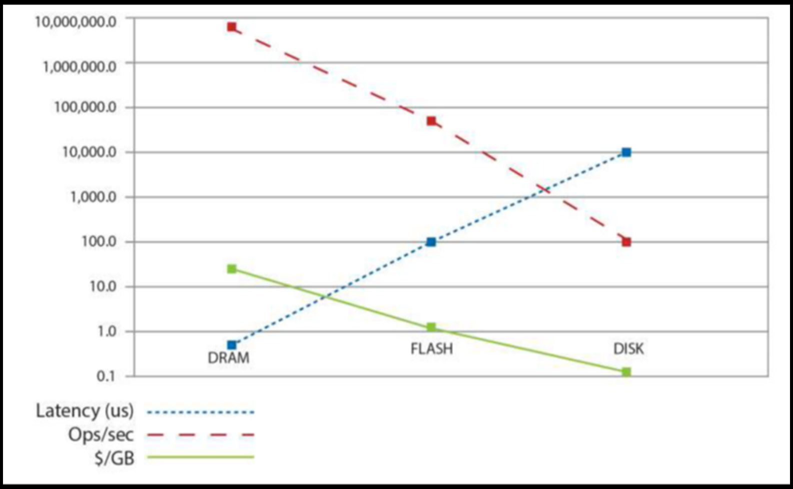


Figure 4: Cost of moving data within a server.

In light of these constraints, massive data processing revolves around these key ideas:

- Scale out/horizontally (i.e., combine more machines), not up/vertically (i.e., increase the computing power of each individual machine).
- Move processing to the data (since clusters have limited bandwidth).
- Process data sequentially and avoid random access (since seeks are expensive).
- Seamless scalability: the speed of data processing should grow faster in proportion to the number of machines used.

2 MAPREDUCE

Lecture 2
19th August 2022

Challenges of big data infrastructure:

1. **Machine failures**: servers have limited lifetimes.
2. **Synchronization**: difficult, because we don't know
   - the order in which workers run.
   - the order in which workers access shared data.
   - when workers interrupt each other.
   - when workers need to communicate partial results.
3. **Programming difficulty**: concurrency is difficult to reason and debug.

⇒ MapReduce provides an API of data centers, hiding system-level details from developers and separating the *what* from the *how*.

2.1 *Basics*

Algorithms for big data problems typically contains the following steps:

1. Iterate over a large number of records.
2. Extract something of interest from each. (**Map**)
3. **Shuffle** and sort intermediate results.
4. Aggregate intermediate results. (**Reduce**)
5. Generate final output.

MapReduce provides a functional abstraction for these steps:

- **map**: $(k_1, v_1) \to List(k_2, v_2)$.
- **reduce**: $(k_2, List(v_2)) \to List(k_3, v_3)$.

with a runtime/backend that handles:

- Scheduling: assigning workers to map and reduce tasks.
- Data distribution: moving processes to data.
- Synchronization: gathers, sorts, and shuffles intermediate data.
- Errors & faults: detects worker failures and restarts.

2.1.1 Implementation

Terminology:

- **Map task**: basic unit of work (typically 128MB).
- **Worker**: single physical machine that can handle multiple map tasks.
- **Mapper/Reducer**: the *process* executing a map or reduce task.
- **Map function**: a single call to the user-defined map $(k_1, v_1) \to List(k_2, v_2)$ function.
  - Note: a single map task can involve many calls to a map function (i.e., each (k, v)-pair in a split will produce one call).

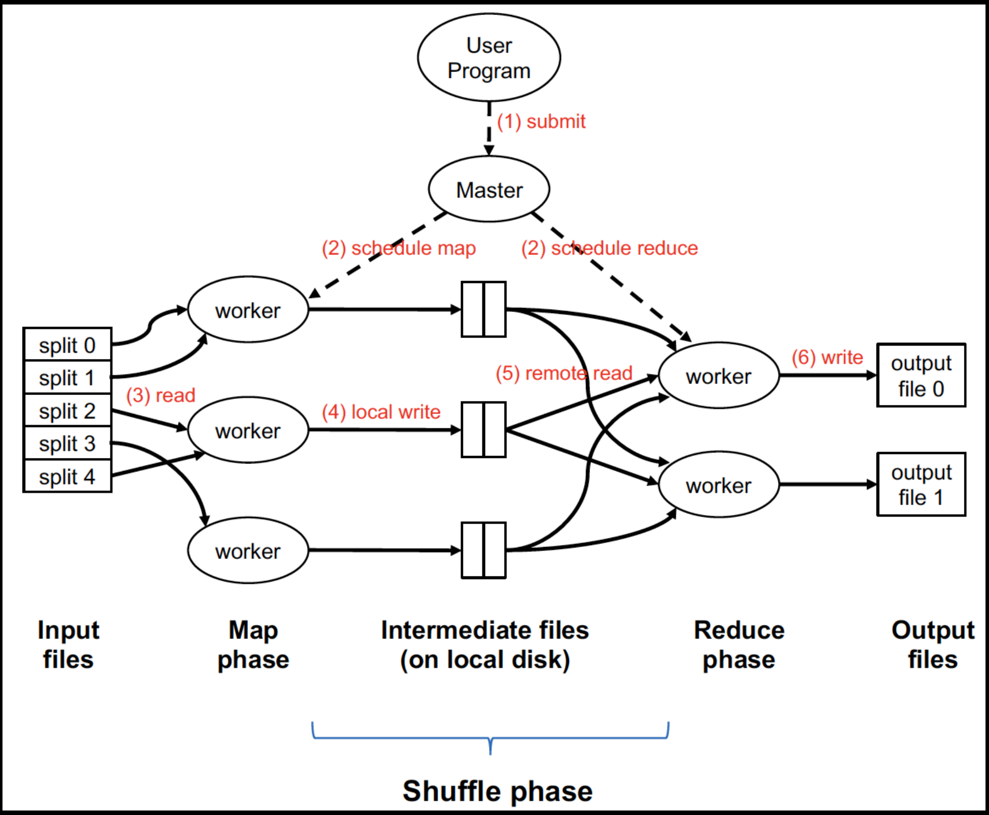


Figure 5: Implementation of MapReduce.

Other details:

- There is a barrier between the map and reduce phases (i.e., all tasks must complete the map phase before any task can enter the reduce phase).
- Keys arrive at each reducer in sorted order.

2.2 *Optimizations*

In addition to map and reduce, programmers may also optionally specify **partition** and **combine** functions (i.e., optimizations to reduce network traffic).

- **Partition**:
  - By default, the assignment of keys to reducers is determined by the hash function: `hash(k) % num_reducers`.
  - Users can optionally implement a custom partition (e.g., to better spread out the load among reducers).
- **Combine**:
  - Combiners *locally aggregate* output from mappers (i.e., akin to "mini-reducers").

- Users have to ensure that the combiner does not affect the correctness of the final output (regardless of how many times it runs).
  - It is generally correct to use reducers as combiners if the reduction involves a binary operation that is (1) **associative** and (2) **commutative** (e.g., sum, max, min).

Lecture 3
26th August 2022

Additionally, if we want each reducer's values to arrive in some sorted ordering, we can perform **secondary sort**:

1. Define a new *composite key* (e.g., (K_1, K_2) where K_1 is the original key and K_2 is the variable to be sorted).
   - Hadoop will automatically sort all pairs by the composite key.
2. Define a custom partitioner which partitions by K_1 only.

2.3 *Performance guidelines*

- **Linear scalability**: more nodes can do more work at the same time.
- **Minimize I/Os** in hard disk (e.g., do sequential instead of random reads) and network (e.g., bulk instead of multiple send/recvs).
- **Memory working set** of each worker/task **should be just right** (large $\rightarrow$ high failure probability; small $\rightarrow$ more workers required).

2.4 *Hadoop distributed file system (HDFS)*

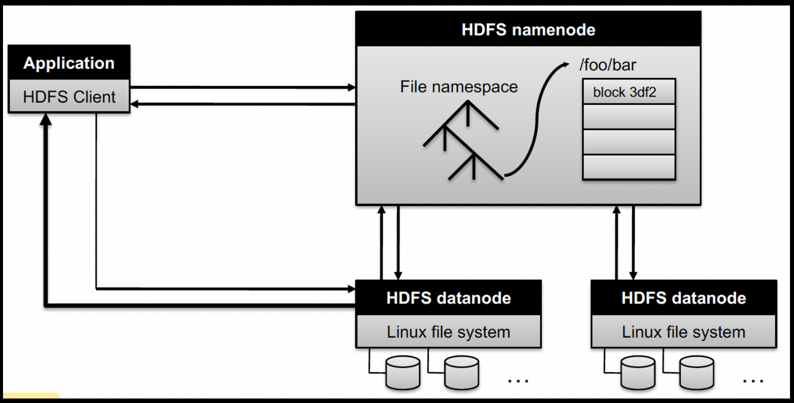


Figure 6: HDFS architecture.

Assumptions:

- High component failure rates.
- Modest number of huge files, typically stored as 128MB chunks.

- Files are write-once, and mostly append-only.
- Large streaming reads instead of random access.

File read/write:

1. Application sends request to *namenode* (i.e., master node).
2. Namenode searches its directory to obtain the list of blocks in this file and where they are stored.
3. Namenode directs the client to read each needed block from the datanode storing it.

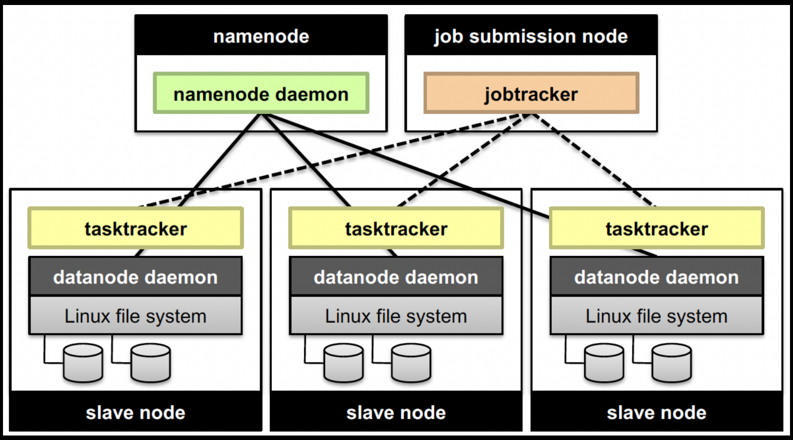


Figure 7: Node interaction within HDFS.

The namenode has several responsibilities, including:

- Managing the file system namespace:
  - Holds directory structures, metadata, file-to-block-mapping, access permissions, etc.
  - Directs clients to data nodes for reads and writes.
  - However, no data is moved through the namenode.
  - If the namenode's data is lost, all files on the filesystem cannot be retrieved (typically, backup & secondary namenodes are provided for resilience purposes).
- Maintaining overall health:
  - Periodic communication with data nodes.
  - Block re-replication (i.e., for data safety/in case a worker crashes) and rebalancing (i.e., load balancing of workers).

3 DATA MINING

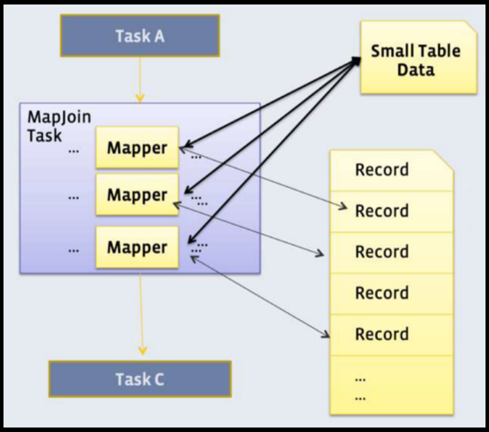
Lecture 4
2nd September 2022

3.1 *Relational databases*

A **relational database** is comprised of tables. Each table represents a relation (i.e., a collection of tuples), and each tuple consists of multiple fields/attributes.

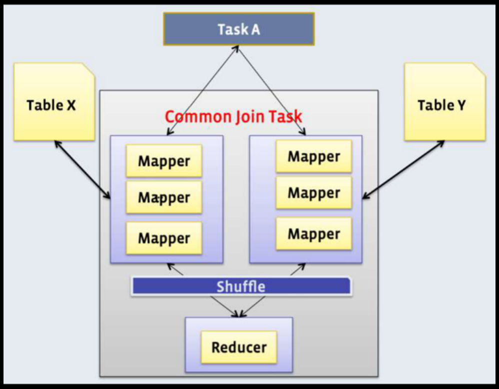
Some operations on relational databases include:

- **Projection** ($\pi_{A_1,\dots,A_n}$).
  - *Map*: take in a tuple (with tuple ID as key) and emit new tuples with appropriate attributes.
  - No reducer needed.
- **Selection** (σ_c).
  - *Map*: take in a tuple (with tuple ID as key) and emit only tuples that meet the predicate.
  - No reducer needed.
- **Aggregations** (e.g., group by).
- **Joins** (e.g., left/right, inner/outer).
  - **Broadcast/Map join**: requires one of the tables to fit in memory.
    1. All mappers store a copy of the small table in the format of a hash table.
    2. Iterate over the respective partitions of the big table, joining the records with the small table.



  - **Reduce/Common join**: does not require a dataset to fit in memory, but slower than broadcast join.
    1. Different mappers operate on each table and emit records, with key as the variable to join by.
    2. Secondary sort to ensure that all keys from table X arrive at the reducer before table Y.

3. Hold the keys from table X in memory and cross them with records from table Y.



3.2 *Similarity search*

Many problems can be expressed as finding "similar" objects, or objects which are apart by a "small distance". We define such points as **near neighbours**.

Some distance/similarity metrics include:

- **Euclidean distance**:

$$d_{\text{Euclidean}}(\vec{a}, \vec{b}) = ||\vec{a} - \vec{b}||^2 = \sqrt{\sum_{i=1}^{D} (a_i - b_i)^2}$$

- **Manhattan distance**:

$$d_{\text{Manhattan}}(\vec{a}, \vec{b}) = ||\vec{a} - \vec{b}|| = \sum_{i=1}^{D} |a_i - b_i|$$

- **Cosine similarity**: only considers direction but not distance.

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

- **Jaccard similarity**: considers similarity between two *sets* A, B.

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

- **Jaccard distance**:

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

For tasks involving similar documents, goals may include:

- **All pairs similarity**: given a large number N of documents, find all "near duplicate" pairs (e.g., with Jaccard distance below some threshold).
- **Similarity search**: given a query document D, find all documents which are "near duplicates" of D.

3.2.1 Near-duplicate document search

Similarity search can be done in a 2-step approach:

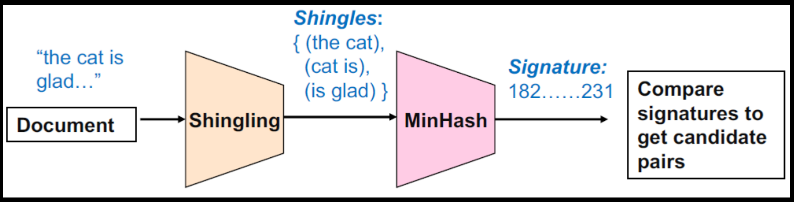

"the cat is glad…"
Document
Shingling
Shingles:
{ (the cat), (cat is), (is glad) }
MinHash
Signature:
182……231
Compare signatures to get candidate pairs

Figure 8: An example of the process of similarity search.

1. **Shingling**: convert documents to sets of k-grams/k-shingles.
2. **Min-hashing**: convert sets of k-shingles to short signatures, while preserving similarity.
   - *MinHash* gives a fast approximation to the result of using Jaccard similarity to compare all pairs of documents.
     - Key property: $Pr\{h(C_1) = h(C_2)\} = s_{\text{Jaccard}}(C_1, C_2)$.
   - Key idea: hash each column C to a small signature $h(C)$ such that:
     (a) $h(C)$ is small enough that the signature fits in RAM, and
     (b) highly similar documents have the same signature w.h.p.
   - Steps:
     (a) Use a hash function h to map each shingle to an integer.
     (b) Compute $h(C)$ as the minimum over all integers.
     (c) Documents with the same signature → **candidate pairs**.

     Optional steps include:
     - Further compare candidate pairs to verify their similarity.
     - Use multiple hash functions (e.g., $N = 100$) to generate N signatures for each document, and define candidate pairs as those with a sufficient number (e.g., ≥ 50) of signature matches.

3.2.2 MapReduce implementation

- *Map*:

1. Read over each document and extract its shingles.
2. Hash each shingle and take the min to get the *MinHash* signature.
3. Emit (signature, docID) pairs.

- *Reduce*:

  1. Receive all documents with a given *MinHash* signature.
  2. Generate all candidate pairs from these documents.

  (Opt) Compare each pair to check if they are actually similar.

3.3 *Clustering*

Goal: separate unlabelled data into groups of similar points.

K-means algorithm:

1. **Initialization**: pick K random points as centers.
2. **Repeat**:
   (a) **Assignment**: assign each point to the nearest cluster.
   (b) **Update**: move each cluster center to average of its assigned points.

   until no changes in assignment.

3.3.1 MapReduce implementation

```
function MAPPER
    function CONFIGURE
        $c \leftarrow$ LOADCLUSTERS
        H $\leftarrow$ INITASSOCIATIVEARRAY
    function MAP(id $i$, point $p$)
        $n \leftarrow$ NEARESTCLUSTERID(clsuters $c$, point $p$)
        $p \leftarrow$ EXTENDPOINT(point $p$)
        $H\{n\} \leftarrow H\{n\} + p$
    function CLOSE
        for each clusterid $n \in$ H do
            EMIT(clusterid $n$, point $H\{n\}$)
function REDUCER
    function REDUCE(clusterid $n$, points $[p_1, p_2, ...]$)
        $s \leftarrow$ INITPOINTSUM
        for each point $p \in$ points do
            $s \leftarrow s + p$
        $m \leftarrow$ COMPUTECENTROID(point $s$)
        EMIT(clusterid $n$, centroid $m$)
```

4 NOSQL

Lecture 5
9th September 2022

Traditional NoSQL refers to non-relational databases, i.e., a relational model is not utilized. Modern NoSQL stands for "Not only SQL", i.e., relational and non-relational databases are used alongside one another for the tasks which they are most suited for.

We will explore the key types of NoSQL databases.

4.1 *Variants*

- **Key-value stores**:
  - Stores associations between keys and values.
  - Keys are usually primitives, and can be queried.
  - Values can be primitive or complex, and cannot be queried.
  - Typical operations:
    - General: **get**, **put**.
    - Optional: **multi-get**, **multi-put**, **range queries**.
  - Suitable for:
    - Small continuous reads and writes.
    - Storing basic information with no clear schema.
    - When complex queries are rarely required.
  - Implementation:
    - **Persistent**: data is stored persistently to disk (e.g., DynamoDB).
    - **Non-persistent**: simply an in-memory hash table (e.g., Redis).
- **Document stores**:
  - A database can contain multiple **collections** (akin to tables in RDBMS).
  - A collection can contain multiple **documents** (akin to rows/tuples in RDBMS).
  - A document is a JSON-like object with **fields** and **values** (akin to columns in RDBMS; however, different documents can have different fields).
  - Typical operations: **CRUD**.
  - E.g., MongoDB, CouchDB.
- **Wide column stores**:
  - Similar to RDBMS, except:
    - Related groups of columns can be grouped as **column families**.
    - **Sparsity**: if a column is not used for a row/tuple, it would not take up space (i.e., it is not assigned a `null` value).

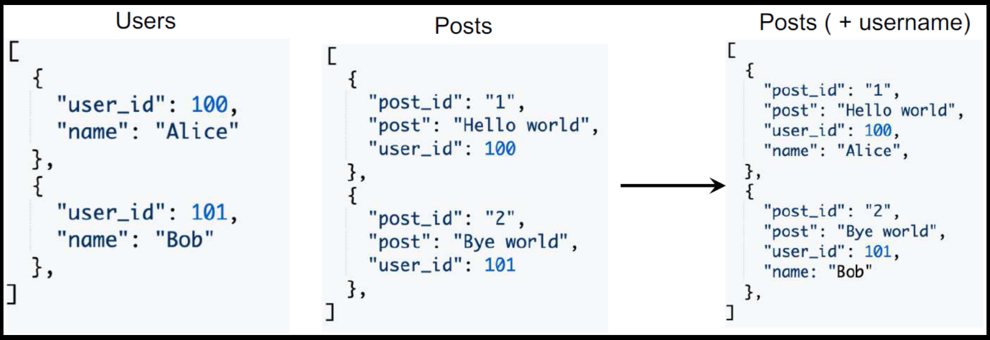
  - E.g., Cassandra, HBase.
- **Graph databases**:
  - Stores **nodes** and **relationships** between nodes instead of tables or documents.
  - E.g., Neo4j.

4.2 *Concepts*

RDBMS provide stronger **ACID** guarantees (i.e., atomicity, consistency, isolation, durability), whereas NoSQL systems provide relaxed **BASE** guarantees (i.e., basically available, soft state, eventually consistent).

- **Basically available**: basic reading and writing operations are available most of the time.
- **Soft state**: without guarantees, we only have some probability of knowing the state at any time.
- **Eventually consistent**: if the system is functioning and we wait long enough, all reads will eventually return the last written value.
  - This is in contrast to **strong consistency**, which requires that any reads immediately after an update must give the same result to all observers.
  - Eventual consistency offers better **availability**, at the cost of a much weaker consistency guarantee.

Duplication/Denormalization of data (i.e., copying attributes from one document to another) can be used to improve query processing efficiency.



```
[
  {
    "user_id": 100,
    "name": "Alice"
  },
  {
    "user_id": 101,
    "name": "Bob"
  },
]
```

```
[
  {
    "post_id": "1",
    "post": "Hello world",
    "user_id": 100
  },
  {
    "post_id": "2",
    "post": "Bye world",
    "user_id": 101
  },
]
```

```
[
  {
    "post_id": "1",
    "post": "Hello world",
    "user_id": 100,
    "name": "Alice",
  },
  {
    "post_id": "2",
    "post": "Bye world",
    "user_id": 101,
    "name: "Bob"
  },
]
```

However, with duplication/denormalization, updates will have to be propagated to multiple tables.

NoSQL systems have the following features:

(+) **Flexible/Dynamic schema**: suitable for less-structured data.

(+) **Horizontal scalability**.

(+) **High performance and availability**: due to relaxed consistency model and ability to replicate/distribute data over many servers.

(−) **No declarative query language**: query logic (e.g., joins) has to be handled on the application side.

(−) **Weaker consistency guarantees**: application may receive stale data.

To determine whether a NoSQL system is preferred over RDBMS, we could consider:

- Whether denormalization is suitable.
- Whether complex queries are utilized.
- Whether consistency is important.
- Whether there is a need for availability.
- Whether the volume of data is high.

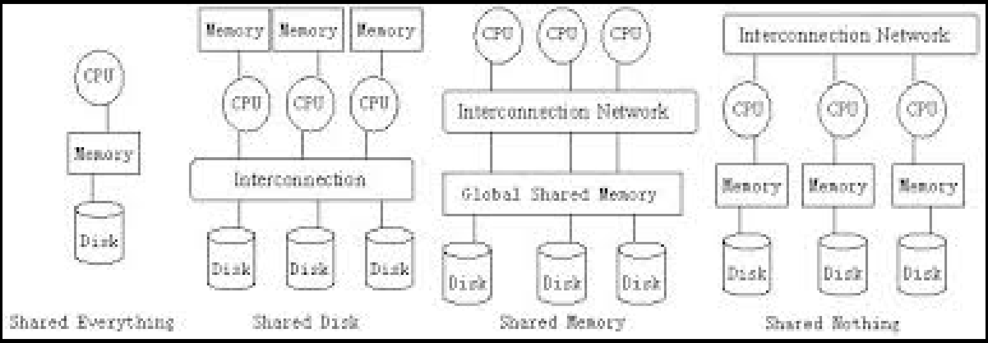
5 DISTRIBUTED DATABASES

Lecture 6
23rd September 2022

Features:

- **Data transparency**: users should not be required to know how the data is physically distributed, partitioned, or replicated.
- Assumption: all nodes in a distributed database are *well-behaved*.

There are several types of distributed database architectures:



- **Shared everything**: CPU, memory, and disk are shared across nodes (i.e., single-node DBMS).
- **Shared memory**: memory and disk are shared across nodes (e.g., supercomputers).
- **Shared disk**: only the disk is shared across nodes (e.g., cloud DBMS).
- **Shared nothing**: nodes communicate over a network, and nothing is shared across nodes (e.g., NoSQL).

5.1 *Data partitioning*

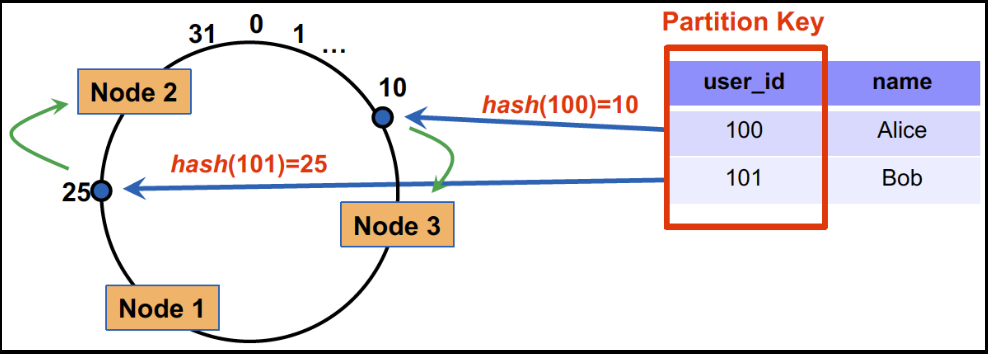
There are two ways of partitioning data:

- **Horizontal partitioning** (i.e., sharding): different tuples are stored in different nodes.
  - **Partition keys** are used to decide which node each tuple will be stored on.
  - Two ways of horizontal partitioning:
    - **Range partition**: splitting partition key based on range of values; good if we need range-based queries, but may lead to imbalanced shards.
    - **Hash partition**: hashing partition key, and dividing the hashes into partitions based on ranges; hash function can be selected such that partition key values are spread out roughly evenly.
      - Consistent hashing can be employed to reduce the cost of adding/removing nodes.
- **Vertical partitioning**: creating tables with fewer columns, and using additional tables (placed on separate nodes) to store the remaining columns.

partition keys are typically selected from columns which often serve as parameters for `filter` or `group by`.

Consistent hashing:

1. Each node is assigned a position/marker on a circle.
2. Each tuple is placed on the circle (based on its hash value), and assigned to the node that comes clockwise after it.



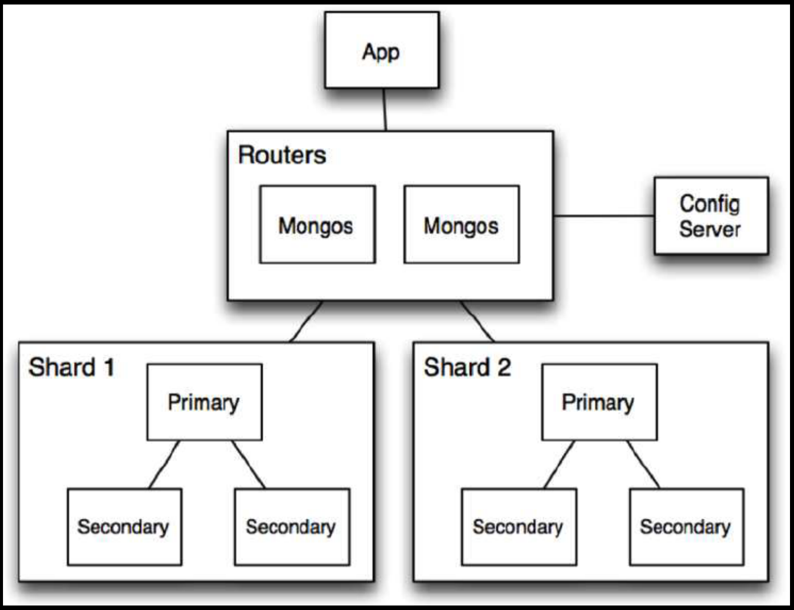
user_id	name
100	Alice
101	Bob

- To delete a node, we simply re-assign all its tuples to the node clockwise after it.
- To add a node, we can split the largest node into two, and assign half of its tuples to the new node.
- **Fault tolerance**: we can 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 on the circle.
  - When we remove a node, its tuples will be distributed across more nodes, thus enabling better load balancing.

5.2 *Distributed query processing*

MongoDB has the following architecture:



- **Routers (mongos)**: handle requests from the application, and route the queries to correct shards.
- **Config server**: stores metadata about which data is on which shard.
- **Shards**: store data and run queries on their data.
  - Each shard consists of primary and secondary nodes, which maintain the same data for better fault tolerance.

For a read/write query,

1. The query is issued to a router (i.e., mongos) instance.
2. With the help of config server, router determines which shards should be queried.
3. The query is sent to relevant shards (i.e., partition pruning).
4. Shards run the query on their data, and send results back to the router.
5. Router merges the query results, and returns the merged results to the application.

6 SPARK

Lecture 7
7th October 2022

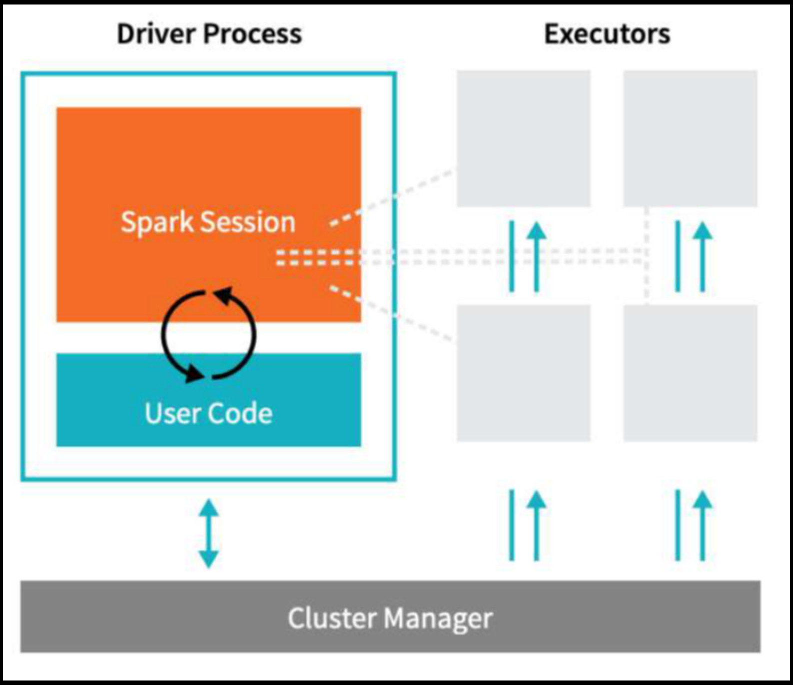
Issues with Hadoop MapReduce:

1. **Network and disk I/O costs**: intermediate data has to be written to local disks and shuffled across machines.
2. **Unsuitable for iterative processing** (i.e., modifying small amounts of data repeatedly), since each step has to be modelled as a MapReduce job.

Benefits of Spark:

1. Stores most intermediate results in memory → much faster (esp. for iterative processing).
   - Spark spills to disk (requires disk I/O) when memory is insufficient.
2. Ease of programmability.

6.1 *Architecture*



Spark consists of:

- **Driver process**: response to user input, manages the spark application, and distributes work to *executors*.
- **Executors**: run code assigned to them, and send the results back to the driver.

- **Cluster manager** (e.g., Spark's standalone cluster manager, YARN, Mesos, or Kubernetes): allocates resources when requested by the application.

6.2 *APIs*

1. **Resilient distributed datasets (RDDs)**:

   Properties:

   - **Immutable**, cannot be changed once created.
   - Represent a collection of objects distributed over machines.
   - Achieves fault tolerance through *lineages*.

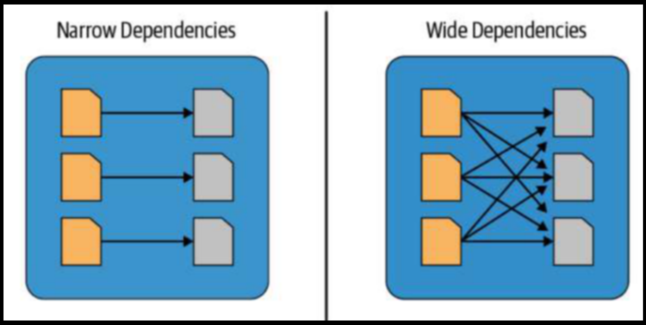
   Using RDDs:

   - **Transformations**: transforming RDDs into RDDs.
     - Transformations are **lazy** (i.e., will not be executed until an *action* is called on it) $\longrightarrow$ allows Spark to optimize query plan to improve speed (e.g., removing unneeded operations).
     - E.g., `map`, `filter`, `order`, `groupBy`, `join`, `select`.
   - **Actions**: trigger Spark to compute a result from a series of transformations.
     - E.g., `show`, `count`, `save`, `collect`.
   - **Caching** helps to speed up jobs by a large margin (e.g., when results are expensive to compute, and need to be reused multiple times).
     - `cache`: saves an RDD to the memory of each worker node.
     - `persist`: saves an RDD to memory, disk, or off-heap memory.

     Caching operations are also lazy.

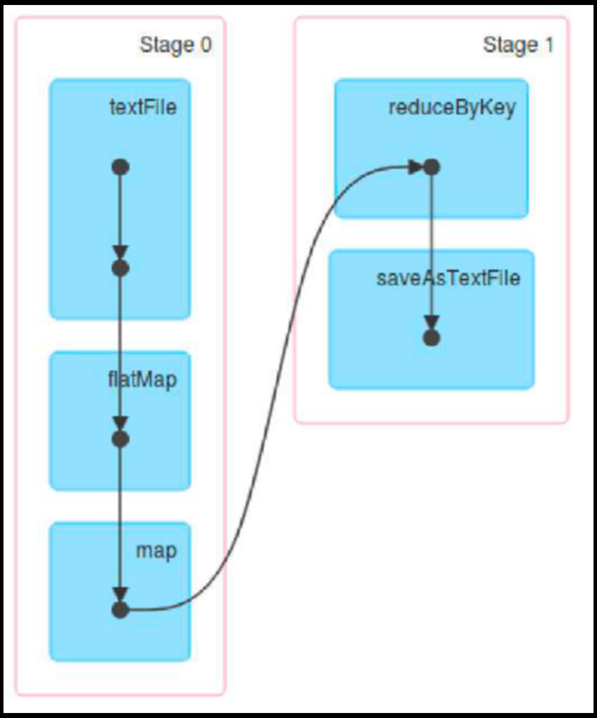
   Transformations and actions on RDDs are executed in parallel, and results are only sent to the driver in the final step.

   Internally, Spark creates a DAG which represents all RDD objects and how they will be transformed.



   - **Narrow dependencies**: each partition of the parent RDD is used by at most 1 partition of the child RDD.
   - **Wide dependencies**: each partition of the parent RDD is used by multiple partitions of the child RDD.

- In the DAG, consecutive *narrow dependencies* are grouped together as **stages**.



  - **Within stages**, consecutive transformations are performed on the same machine → very fast.
  - **Across stages**, data needs to be shuffled/exchanged across partitions, and intermediate results need to be written to disk → slow.

Fault tolerance:

- *Replication* is not used for fault tolerance, since memory capacity is much more limited than disk → duplicating data is expensive.
- **Lineage approach**: if a worker node goes down, it is replaced with a new worker node, and the DAG is used to recompute the data in the lost partition.
  - Much more efficient, since within-stage computations are inexpensive.

2. **DataFrames**:

Properties:

- Represent tables of data (similar to SQL).
- Higher level interface compared to RDDs, contains transformations resembling SQL operations.
  - Ultimately, all DataFrame operations are compiled down to RDD operations by Spark.

  - SQL queries can also be used to transform DataFrames (i.e., using `spark.sql("<sql query>")`.

3. **Datasets**: similar to DataFrames, but are *type-safe* (thus, they are not available in dynamically-typed languages such as Python and R).

6.3 *Spark MLLib*

Lecture 8
21st October 2022

Using Spark's MLLib, we can build complex pipelines out of simple building blocks (e.g., normalization, feature transformation, model fitting, etc.). This allows for better code reuse, and makes it easier to perform cross validation and hyperparameter tuning.

Pipelines consist of the following building blocks:

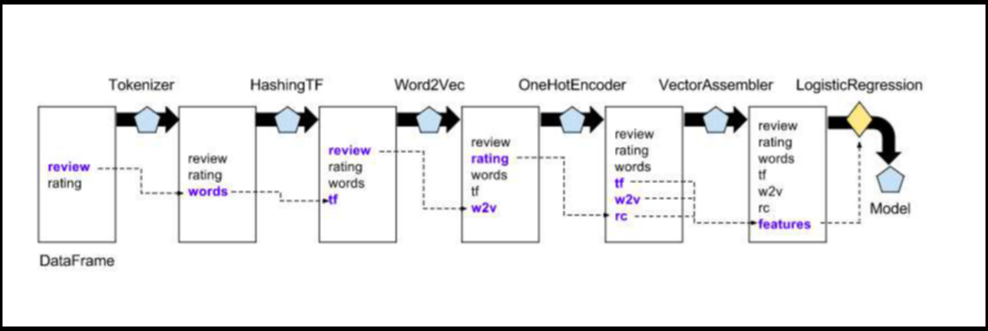
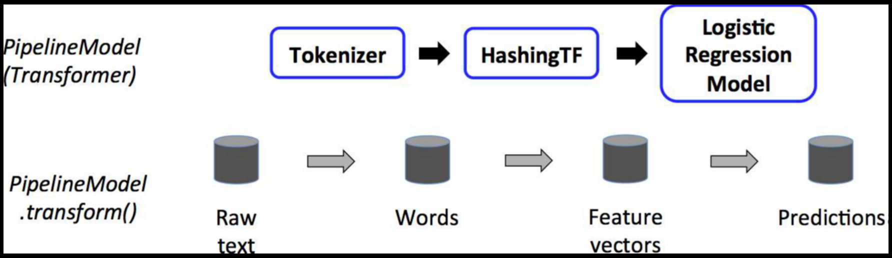


Figure 9: Blue pentagons represent *transformers*, whereas yellow diamonds represent *estimators*.

- **Transformer**: maps DataFrames to DataFrames.
  - Transformers have a `transform()` method which performs the transformation.
  - Generally, transformers output a new DataFrame, which will be appended to the original DataFrame.
- **Estimator**: algorithm which takes in data and outputs a fitted model.
  - Estimators have a `fit()` method which returns a transformer.
  - The pipeline itself is also an estimator (since it returns a fitted model, which is a transformer).



A pipeline chains together multiple transformers and estimators to form an ML workflow. The output for `Pipeline.fit()` is the estimated pipeline model, which is a transformer that consists of a series of transformers.

7 GRAPHS

Lecture 9
28th October 2022

Graphs contain:

- **Nodes**: represent objects (e.g., journals).
- **Edges**: represent relationships (e.g., citations).
  - Can be *undirected* or *directed*.

7.1 *PageRank*

The web can be modelled as a directed graph, where nodes represent *web-pages* and edges represent *hyperlinks*. Measuring the **importance** of pages is necessary for many web-related tasks (e.g., search and recommendation).

The importance of the page can be determined by its number of **in-links** (e.g., votes). Specifically,

- For each page j, define its importance/rank as r_j.
- Page j's importance is the sum of the votes on its in-links.
- If page j has n out-links, each link gets $\frac{r_j}{n}$ votes.

In other words, the importance r_j for a page j is represented by the following equation:

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

where d_i rep. the number of out-links (i.e., out-degree) of node i. The **flow equations** of PageRank consist of N equations and an additional constraint:

$$\begin{aligned} r_1 &= \sum_{i \to 1} \frac{r_i}{d_i} \\ &\vdots \\ r_{\mathrm{N}} &= \sum_{i \to \mathrm{N}} \frac{r_i}{d_i} \\ \sum_i r_i &= 1 \qquad \text{this constraint enforces uniqueness.} \end{aligned}$$

The flow equations can also be formulated as a matrix multiplication problem. Specifically, we have:

- **Column stochastic matrix** M, where

  $$M_{ji} = \begin{cases} \frac{1}{d_i} & \text{if } i \to j \\ 0 & \text{otherwise} \end{cases}$$

  and all columns sum to 1.
- **Rank vector** r, where each r_i rep. the importance score of page i, and $\sum_i r_i = 1$.

Therefore, the flow equations can be written equivalently as $r = M \cdot r$.

7.1.1 Power iteration

Given a web graph with N nodes, we can use **power iteration** to compute r.

1. Initialize $r^{(0)} = [\frac{1}{N}, ..., \frac{1}{N}]^T$.
2. Repeat $r^{(t+1)} = M \cdot r^{(t)}$ until $|r^{(t+1)} - r^{(t)}|_1 < \epsilon$.

$|x|_1 = \sum_i |x_i|$ is the L_1 norm. Other vector norms may be used (e.g., Euclidean norm).

Intuitively, each node is assumed to have equal importance initially (i.e., $\frac{1}{N}$), and during each step, each node passes its current importance along its outgoing edges to its neighbours.

7.1.2 Random walk formulation

PageRank can also be formulated as a random walk on the web graph. Specifically, the following process is assumed:

1. At time $t = 0$, a web surfer starts on a random page.
2. At time $t + 1$, the surfer follows an out-link from i uniformly at random (process repeats indefinitely).

Let $p(t)$ rep. the vector whose i^{th} coordinate rep. the probability that the surfer is at page i at time t (i.e., $p(t)$ is a probability distribution over all pages). We observe that as $t \to \infty$, the probability distribution approaches a **steady state**, representing the long term probability that the random walker is at each node.

This steady state probability is equivalent to the rank of each page computed from the flow equations.

7.1.3 Teleportation

The naïve formulation of PageRank $r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$ (or equivalently, $r = \mathrm{M} \cdot r$) has several issues:

- The formulation may not always converge (e.g., when the graph is a simple cycle).
- The presence of **dead ends** and **spider traps** may cause the random walk to get stuck.

spider trap: group of nodes where all out-links are within the group.

To resolve these issues, teleportation can be employed. Specifically:

Typically, $0.8 \leq \beta \leq 0.9$.

- At each time step, the surfer will follow a link at random with probability β, and jump to some random page with probability $1 - \beta$.
- If the node is a dead-end, the surfer will randomly teleport with probability 1.

Thus, the PageRank equation can be rewritten as

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

or in matrix form,

$$\mathrm{A} = \beta \mathrm{M} + (1 - \beta)\left[\frac{1}{\mathrm{N}}\right]_{\mathrm{N} \times \mathrm{N}}$$

$$r = \mathrm{A} \cdot r$$

where $\left[\frac{1}{\mathrm{N}}\right]_{\mathrm{N} \times \mathrm{N}}$ rep. an N by N matrix where all entries are $\frac{1}{\mathrm{N}}$.

Even with teleportation, some problems still persist:

- PageRank only measures the *generic popularity* of a page (i.e., does not consider popularity based on specific topics).
- PageRank only uses a single measure of importance.
- PageRank is still susceptible to link spam (i.e., artificial link topographies can be created to boost page rank).

7.1.4 Topic-specific PageRank

Instead of generic popularity, we can evaluate webpages by how close they are to a particular topic (e.g., sports, history). This can be done by biasing the random walk, by changing the **teleport set** to a topic-specific set of "relevant" pages (e.g., retrieved from DMOZ).

Formally, this changes the formulation of PageRank to

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

$$r = A \cdot r$$

7.2 *Streams*

Streaming/Online approaches are designed to process data/input **as it is received**. Streams may potentially contain an infinite number of elements/tuples.

Opp: offline/batch processing.

7.2.1 Pregel

Pregel is a computational model where computation is defined as a series of **supersteps**. It can be used to execute algorithms on large graphs (e.g., PageRank).

In each superstep, the framework invokes a **user-defined function `compute()` for each vertex**, which specifies the behaviour at a single vertex v and a superstep s, such as:

- **Read messages** sent to v in superstep $s-1$.
- **Send messages** to other vertices that will be read in superstep $s+1$.
- **Read/Write the value** of v and the value of its outgoing edges.

Vertices can choose to deactivate themselves, and they are woken up if new messages are received. Computation halts when *all* vertices are inactive.

Pregel uses a master-worker programming pattern:

- Vertices are **hash-partitioned** and assigned to workers.
- Each worker maintains the states of its graph partitions *in memory*.
  - In each superstep, each worker loops through its vertices and executes `compute()`.
  - Messages from vertices are sent to other vertices (buffered locally before being sent as a batch, to reduce network traffic).
- Fault tolerance is ensured by checkpointing to persistent storage, and detecting failure through heartbeats.

hash partition: randomly assigned to worker nodes based on some hash function.

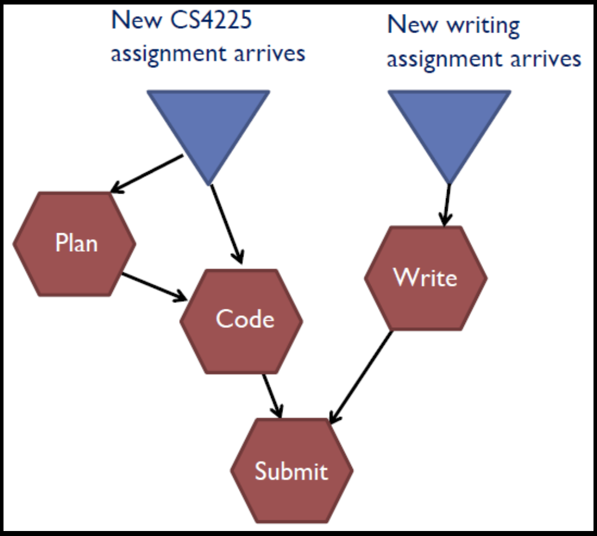
7.2.2 Stream algorithms

- **Reservoir sampling**: given a stream of items $\{a_1, a_2, ...\}$, maintain a **uniform random sample** of fixed size over time.
  - This allows us to estimate almost any statistics (e.g., mean, variance, median, etc.) of the data distribution in a streaming manner.
  - Algorithm:
    1. The first B items are inserted into the reservoir.
    2. When receiving a_t, replace a random item in the reservoir with a_t with probability B/t.
- **Running mean & variance**: given a stream of values $\{x_1, x_2, ...\}$, keep track of the mean and variance of all values that have appeared in the stream so far.
  - Running mean algorithm:
    1. When receiving x_i:
       (a) Update current number of items: $n+ = 1$.
       (b) Update current sum: $\text{sum}+ = x_i$.
    2. When queried, return sum/n.
  - Running variance algorithm:
    1. When receiving x_i:
       (a) Update current number of items: $n+ = 1$.
       (b) Update current sum: $\text{sum}+ = x_i$.
       (c) Update current sum of squares: $\text{sum\_sq}+ = x_i^2$.
    2. When queried, return the formula for variance: $1/(n-1) * (\text{sum\_sq} - (\text{sum}^2/n))$.

    The above approaches can be extended to compute mean/variance of a sliding window (i.e., **moving average/variance**).
- Checking whether items have appeared in a stream → **bloom filters**.
- Counting the number of distinct elements → **Flajolet-Martin/HyperLogLog**.
- Estimating number of appearances of an element → **count min sketch**.

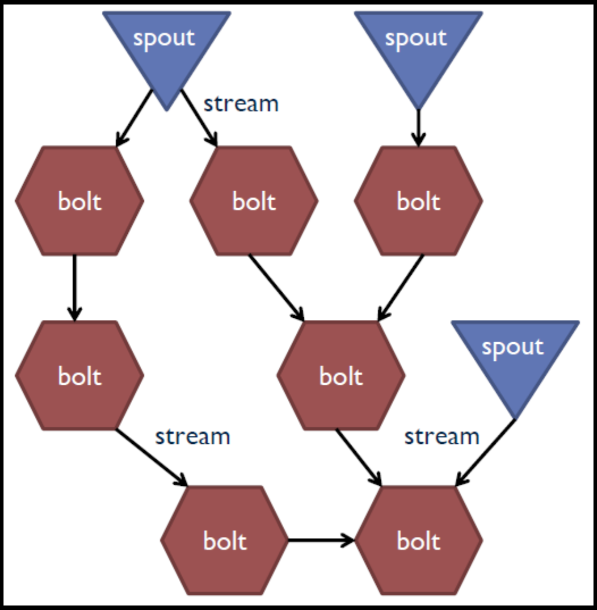
7.2.3 Storm

Storm is a **stream data processing system** which is able to perform computations on a flowchart-like graph, e.g.:



To do computation in Storm, the user creates a **topology** (i.e., computation graph), where:

- Nodes hold processing logic (i.e., transformation over its input).
- Edges indicate communication between nodes.
- Each topology corresponds to a Storm job.



More specifically, each topology consists of the following features:

- **Streams**: an unbounded sequence of tuples, to be transformed by the processing elements of a topology.
- **Spouts**: stream *generators*, may propagate a single stream to multiple consumers.

- **Bolts**: stream *subscribers* and *transformers*, process incoming streams and produce new ones.
  - Bolts are executed by multiple executors in parallel (i.e., tasks).
  - When a bolt emits a tuple, the task it is assigned to is based on the choice of **stream grouping**:
    - **Shuffle grouping**: tuples are randomly distributed across tasks.
    - **Field grouping**: tuples are distributed based on the value of a (user-specified) field.
    - **All**: tuples are replicated across all tasks of the target bolt.

* * *