Principles of Big Data Systems

1. **Scale "out", not "up"**: Combining many cheaper machines (horizontal scaling) instead of increasing the power of each individual machine (vertical scaling); high performance-cost ratio
2. **Seamless scalability**: Performance should scale linearly with the number of machines
3. **Move processing to the data**: Move tasks to the machine where the data is stored to minimise bandwidth
4. **Sequential access > Random access**: To minimise disk access

Challenges : 4 "V"s

1. **Volume**: Performance, Cost, Reliability & Algorithm Design Complexity
2. **Velocity**: Performance, Cost, Reliability & Algorithm Design Complexity
3. **Variety**: Data format (e.g. text, image) and how to integrate them
4. **Veracity**: Data accuracy; dirty and noisy data causes data uncertainty

Data Center Architecture

Storage Hierarchy

- **Server**: 1 server = RAM + disk
  - Disk access: RAM → Disk
- **Rack**: 1 rack = multiple servers stack on top of each other. Servers within a rack are connected via a rack switch
  - Disk access across servers within same rack: RAM → Rack Switch → RAM → Disk
- **Cluster**: 1 cluster = multiple racks. Racks within a cluster are connected via a datacenter/cluster switch
  - Disk access across servers from different racks: RAM → Rack Switch → Datacenter Switch → Rack Switch → RAM → Disk
- Capacity increases from local server to rack to datacenter
- Capacity in disk >> Capacity in RAM

Data Access costs

- **Latency** = <u>sum</u> of latency caused in all components in data access
  - RAM (ns) << Rack Switch ≈ Datacenter Switch (µs) << Disk (ms)
- **Bandwidth** = <u>minimum</u> of the bandwidths among all components
  - RAM (GB) >> Disk (MB) ≈ Rack Switch >> Datacenter Switch (MB)
- Latency increases from local server to rack to datacenter
- Bandwidth decreases from local server to datacenter
- Large data: Bandwidth matters; Small data: Latency matters

<u>MapReduce</u>

- Runs within a distributed file system with a master node

1. User submits MapReduce program and config to master node
2. Master node schedules resources to **workers** for map and reduce tasks
3. Split large input file into chunks, typically 128MB each.
4. **Map**: Read file chunk(s) and calls `map()` for each input key-value pair in a chunk; Outputs key-value pairs
   - 1 chunk (= **map task**), 1 worker. 1 worker, 1 or more chunks
   - Read: Scheduler tries its best to allocate the worker in the machine which the chunk is located in to perform the read (local read > remote read)
   - Output is written to intermediate files on <u>local disk</u> (ie. local write), partitioned by key: To allow sequential reads later by the reducers
5. **Shuffle**: Order and group same keys together
6. **Reduce**: Read and aggregate the list of values and returns the result
   - 1 key, 1 reducer. 1 reducer, 1 or more keys
   - Set of keys in a reduce task are processed in sorted order
   - Local/Remote read: Same/Diff worker used for map for that key
   - Streaming read: Start copying output from mappers as soon as they arrive (but DO NOT PROCESS THEM YET)
7. Write (local/remote) result to disk

- **All** map jobs must complete before reducers can execute
- Chunk too big: Limited parallelism + cannot fit in memory
- Chunk too small: High overhead; master node must keep scheduling + More random access to read chunks

Optional Optimisations

- **Partition**
  - Specify custom behaviour for allocating keys to reducers
  - Improves load balancing, maximising bandwidth
- **Combiner**
  - Locally aggregate mapper
    - Reduces network I/O by reducing size of intermediate results before sending them to reducers
  - Reads intermediate results from mapper (incurs local disk I/O)
  - Runs on **same machine as the mapper**
  - Combiner and reducer are often the same function
    - This function must be **associative** and **commutative**
  - May not necessarily run. Hence, the **output type must also match with the output of the mapper (and combiner input)**
  - **In-memory combiner**: setup() + emit in cleanup(). Does not incur disk I/O since it reads the intermediate results from the map() task (**same chunk**) from memory
    - Reduces disk I/O since fewer kvps are written to disk

<u>Hadoop Distributed File System (HDFS)</u>

- **Assumptions**
  1. Scale "out", not "up"
  2. High component failure rates
  3. Modest number of huge files
  4. Files are write-once, mostly appended to
  5. Large sequential reads instead of random access
- **Design Decisions**
  1. Files are stored as chunks of 128MB
  2. Each chunk is replicated across ≥ 3 chunkservers (for reliability)
     - Too many replicas: Storage overhead + I/O cost
  3. Single master (namenode) to coordinate access and keep metadata
- **Data Replication**
  1. Namenode decides which datanodes to use as replicas
  2. 1st datanode forwards data blocks to the 1st replica, which forwards them to the 2nd replica and so on
  - Offers greater flexibility; improves read I/O
  - No more than 1 per node; no more than 2 in the same rack

Namenode

- Manages file system namespace
  - Holds file/directory structure, metadata, permissions etc
  - Directs clients to datanodes for read/write
  - [illegible]
- Manages overall health
  - Periodic communication with datanodes
  - Block re-replication and rebalancing
  - Garbage collection
- Backups and secondary namenodes to recover from namenode failure

Scalability Analysis

- Max # map tasks = $\lceil \text{input size} / \text{chunk size} \rceil$. Linear to the input size?
  - Assign 1 map task to 1 mapper
- Max # reduce tasks = # distinct keys. Linear to the input size?
  - Assign 1 key to 1 reducer

I/O Analysis

- Reading: *mainly* disk I/O
  - Usually able to read from local disk due to data replication. Remote reads will incur also network I/O
- Shuffle & sort: Disk and network I/O
- Output: Disk and network I/O
  - Network I/O comes from data replication
- **Map**:
  - Local Disk I/O [read] = chunk size = 128MB
  - Local Disk I/O [write] = # key-value pairs or # emits
  - I/O from Intermediate results: size of variables used (usually negligible)
- **Shuffle**:
  - Network I/O w/o combiner = Disk I/O [write] from Map
    - <u>Combiners</u> may reduce this
- **Reduce**:
  - Network I/O [read] = taken into account by shuffle
  - Network I/O [write] = negligible (size of 1 key-value pair)
  - Disk I/O [read] = negligible (usually a remote read), unless there are other sources of inputs
  - Disk I/O [write] = negligible (size of 1 key-value pair)
  - I/O from Intermediate results: size of variables used (usually negligible)

Memory Consumption Analysis

- Memory allocation for map/reduce functions (<u>memory working set</u> for intermediate results)
  - Variables and intermediate data structures

<u>Join Algorithms</u>

Broadcast Join

- One of the input tables can fit into main memory (to prevent disk I/O)
- Each mapper will contain a copy of the small table in main memory
- Each mapper receives a chunk of records from the large table, then probes the small table in memory
- Emits (join key value, joined tuple); no shuffling, <u>no need reducer</u> if just join
- Each chunk + small table must both fit in main memory

Reduce-side Join

- Different mappers operate on each table (each mapper only operate on one of the tables)
- Emits (join key value, tuple) + flag to indicate which table → shuffle → tuples with the same join key values will end up in the same reducer

Data Mining

- $s_{Jaccard}(A,B) = \frac{|A \cap B|}{|A \cup B|}$, $d_{Jaccard}(A,B) = 1 - s_{Jaccard}(A,B)$

Similar Documents

1. **Shingling** – Convert documents into sets of short phrases
   - k-gram: sequence of tokens with length of k
   - Convert each document D_i into a set of k-shingles C_i,
     - e.g. 2-gram, "The cat is cute" → {"The cat", "cat is", "is cute"}
   - $s_{Jaccard}(D_i, D_j) = \frac{|C_i \cap C_j|}{|C_i \cup C_j|}$; often represented as a matrix M with all possible shingles as the rows and documents as the columns and M_{ij} = 1 if D_j contains shingle i and 0 otherwise
   - $\frac{N(N-1)}{2}$ pairwise comparisons, N = # documents → each comparison needs to compare every row in C_i and C_j → very slow
2. **MinHash** – Compute signature of a document
   - Fast approximation of the result of using $s_{Jaccard}$ to compare all pairs of documents by converting large sets to short signatures (**fits in RAM**) while preserving similarity
   - Candidate pairs have same signature
     - $\Pr[\text{MinHash}(D_i) = \text{MinHash}(D_j)] = s_{Jaccard}(D_i, D_j)$
   - $\text{MinHash}(\text{D}_\text{i}) = \min\{ h(s_j), \forall s_j \in \text{Shingles}(\text{D}_\text{i}) \}$
   - Use N different independent h to generate N signatures for each document. Then, (D_i, D_j) are candidate pairs if $\geq k$ of the signatures are equal

MapReduce Implementation

- **Map**
  1. Receives D_i as input
  2. Shingle and compute MinHash
  3. Emit (signature, document id i)
- **Reduce**
  1. Receives (signature, document id i)
  2. Generate all candidate pairs
  3. Optionally, compare each pair to check if they are actually similar

Clustering

K-Means Algorithm

1. **Initialize:** Pick k random points as centers
2. **Repeat** until no assignments changed
   2.1. Assign each point to nearest cluster
   2.2. Move each cluster center to average of its assigned points

MapReduce Implementation (of a single iteration)

- **Map**
  1. Receives point P_i as input
  2. Find nearest cluster C_j for P_i
  3. P_i ← ExtendPoint(P_i)
  4. Emit (j, P_i)
- **Reduce**
  1. Receives (j, P_i)
  2. Calculate centroid m (new cluster center) for cluster C_j
  3. Emit (j, m)
- Disk I/O: $O(nmd)$, n = # points, m = # iterations, d = # dimensions

MapReduce Implementation 2 (of a single iteration)

- **Map**
  1. CONFIGURE: Initialize hash table H
  2. Receives point P_i as input
  3. Find nearest cluster C_j for P_i
  4. P_i ← ExtendPoint(P_i): Adds an extra dimension with constant value 1 to record the # of points in cluster j for those chunks
  5. $\boldsymbol{H}[\boldsymbol{j}] \leftarrow \boldsymbol{H}[\boldsymbol{j}] + P_i$ (partial vector summation)
  6. CLOSE: For each cluster_id k in H, Emit $(k, H[k])$, i.e. emit all the set of points in each cluster
- **Reduce**
  1. Receives $(j, [P_1, P_2, \dots, P_n])$
  2. Sum all points in $[P_1, P_2, \dots, P_n]$ to get vector S
  3. Calculate centroid m for cluster C_j by taking S/# points
  4. Emit (j, m)
- Disk I/O: $O(kmd)$, k = # clusters, m = # iterations, d = # dimensions

Apache Spark

- Each iteration in Hadoop MapReduce incurs network I/O from shuffling and local disk I/O to store intermediate results.
- Hence, Hadoop MapReduce is not suitable for iterative processing since each individual step has to be modelled as a MapReduce job
- Spark stores most intermediate results in memory (but can spill to disk when memory is insufficient)
- Architecture:
  - **Driver Process:** responds to user input, managed Spark application and distributes work to Executors
  - **Executors**: Runs code and send results back to driver
  - **Cluster Manager**: Allocates resources

Resilience Distributed Datasets (RDD) – "HOW"

- **Immutable**; create and partition using `parallelize`
- **Transformation**: returns a new RDD. Transformations are **lazy**
  - Allows Spark to optimize query plan before execution
- **Action**: Spawns a Spark job and triggers execution of transformations
- Transformations and actions are executed in parallel on each partition across the different workers
- **Caching**: An RDD can be cached to memory of each worker node using `cache` or `persist` (gives option to save to memory or disk)
  - If out-of-memory: least recently used RDD will be evicted
  - Cache RDD if it is expensive to compute or is used multiple times

Directed Acyclic Graph/Lineage

- Spark builds a DAG representing the series of transformations leading up to an action
- An action will trigger the series of transformations in the DAG
- **Narrow Dependencies**: Each partition of the parent RDD is used by at most 1 partition of the child RDD
- **Wide Depencencies**: Each partition of the parent RDD is used by multiple partitions of the child RDD (e.g. orderBy, reduceByKey):
  - Network shuffling
  - Write immediate result to **disk** for persistency to prevent reshuffling if a machine fails
- Consecutive narrow dependencies are grouped together within a **stage**. All transformations within a stage are executed within the same machine (i.e. can be performed locally)
- Data is shuffled across stages (wide dependencies) → try to minimise this

Fault Tolerance

- A faulty worker node can be replaced with a new one, and use the DAG to recompute the RDDs in (only) the lost partition
- Unlike Hadoop, Spark does not do replication due to memory scarcity

DataFrames – "WHAT"

- Immutable; a dataframe is like a small SQL table
- Ultimately compiled down to RDDs using Tungsten, but easier to use
- **Transformation**: returns a new dataframe; similar to a SQL operation. Transformations are **lazy** and executed when an action is triggered
- **Action**: returns a value other than a dataframe; similar to SQL aggregate functions
- **Catalyst optimizer** analyzes the lineage and possibly re-arranges the transformations to derive an optimized execution plan

Machine Learning

Data Pre-processing

- **Missing values**
  - Numerical: Fill in with mean/median
  - Categorical: Fill in with mode
  - Add "missing" indicator column
- **Categorical Encoding** – spark can only handle numerical features; encode categories into numbers
  - eg. [Low, Medium, High] = [0, 1, 2]
  - e.g. [A, B, C] = [100, 010, 001] (one-hot encoding, if there's no ordinal relationship among categories)
- **Normalization** – normalize features to restrict them within a range

Model Training (Classification)

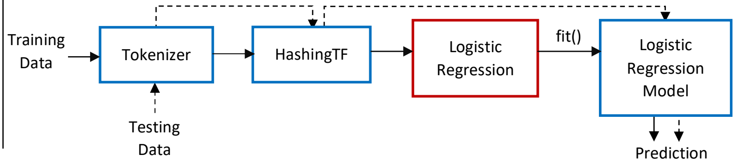
- Use logistic regression = $\sigma(w \cdot x + b) = \hat{y}$
- Use Cross Entropy Loss (same as CS2109S)

Model Evaluation

- **Classification**: Accuracy, Precision, Recall, F1 (same as CS2109S)
- **Regression**: MSE, MAE, Root MSE (RMSE), R-Squared Value (higher the better)

Spark ML Pipeline

- **Transformer**: Maps dataframes to dataframes
  - Instead of outputting a new dataframe, it appends the result to the original dataframe as new columns. Have transform() method
- **Estimator**: Takes in data and outputs a fitted model
  - They have a fit() method which returns the fitted model (which is a transformer). fit() is an action. Pipeline is an estimator
  - Pipeline is an estimator (solid). PipelineModel is a transformer (dashed)

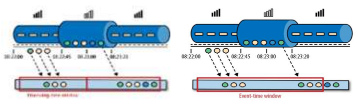


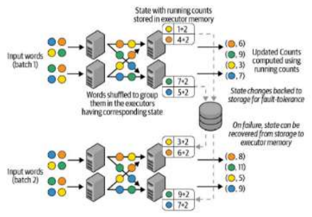
Stream Processing

- Handling data as it is received vs operating on the full dataset
- Stateful stream processing: Ability to store and access intermediate data for recovery

Streaming with Spark

- Divides data into **micro-batches**, then process each batch in a distributed manner
- Spark generates logical and optimized plan only once for all microbatches, i.e each batch uses exactly the same optimized plan
- Advantages
  1. **Quickly** and **efficiently recover** from failures
  2. **Deterministic**; ensures end-to-end exactly-once processing through checkpointing to stable storage, maintaining the state across failures
  3. Ability to **reuse code** for batch processing for stream processing – treat data stream as an unbounded table
- Disadvangates
  1. Longer **latency**; a record from a micro-batch that arrives earlier need to wait for later records in order to be processes
- Steps:
  1. Define input source(s)
  2. Transform data (batch-processing API)
  3. Define output sink (where and how to output)
  4. Specify processing details:
     a. How often to trigger results (e.g. every 5 minutes)
     b. Checkpoint location: To save current state for failure recovery
- Checkpointing:
  - For stateful transformations, in every micro-batch, update the state from the previous batch
  - State is stored on the memory/disk, and checkpointed to a persistent, fault-tolerant location such as the HDFS or Amazon S3

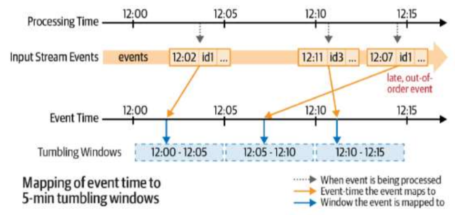



State with running counts stored in executor memory
Input words (batch 1)
Words shuffled to group them in the executors having corresponding state
Updated Counts computed using running counts
State changes backed to storage for fault-tolerance
On failure, state can be recovered from storage to executor memory
Input words (batch 2)

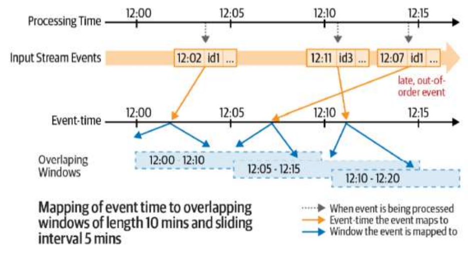
Time-based Aggregations

- Aggregations not based on time (e.g. group aggregations, count, etc) can be performed by processing each batch, saving the new state and propagate the new state for future batches (as shown above), but not time-based aggregations
- **Processing Time**: The time the machine starts to process the record
  - Non-deterministic; suppose a failure causes a rollback to the previous state. Then, the previous state may contain the diff records, depending on the speed at which the records arrive
- **Event Time**: The time the event actually happens
  - Deterministic; suppose a failure causes a rollback to the previous state. Then, the previous state will always contain the same records

- An event is allocated to an event time window by its event time, regardless of what time it actually arrives:
  - Tumbling windows: Windows do not overlap. An event will be allocated to at most 1 window

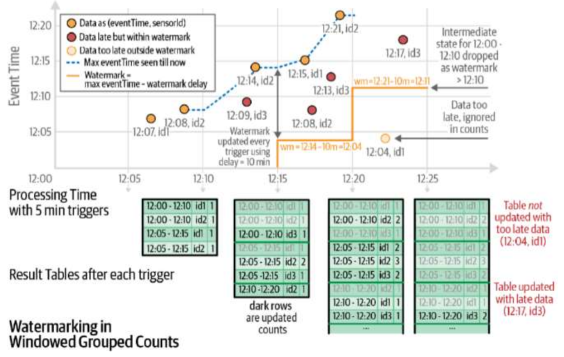

Processing Time
Input Stream Events
Event Time
Tumbling Windows
12:00 - 12:05
12:05 - 12:10
12:10 - 12:15
late, out-of-order event
Mapping of event time to 5-min tumbling windows
When event is being processed
Event-time the event maps to
Window the event is mapped to

  - Overlapping windows: Windows overlap. An event can be allocated to more than 1 window


Processing Time
Input Stream Events
Event-time
Overlapping Windows
12:00 12:10
12:05 - 12:15
12:10 - 12:20
late, out-of-order event
Mapping of event time to overlapping windows of length 10 mins and sliding interval 5 mins
When event is being processed
Event-time the event maps to
Window the event is mapped to

Watermarks

- Motivation: How do you know when all the data happening during the time window has arrived? E.g. From 12:00 to 12:05, there are 10 events, and 1 event got delayed and only arrived at 12:30. How do you know where or not to wait until 12:30?
- Watermark = max eventTime – watermark delay
  - Max eventTime = max eventTime seen till now
- Watermarks are calculated as results are triggered
- When a watermark is triggered, we can no longer consider rows from results table that have event time range **ending < watermark**, ie these rows are finalised
  - However, an event with eventTime < watermark may still be inserted into the results table


Event Time
Data as (eventTime, sensorId)
Data late but within watermark
Data too late outside watermark
Max eventTime seen till now
Watermark = max eventTime - watermark delay
Intermediate state for 12:00 - 12:10 dropped as watermark > 12:10
Data too late, ignored in counts
Watermark updated every trigger using delay = 10 min
Processing Time with 5 min triggers
Result Tables after each trigger
dark rows are updated counts
Table not updated with too late data (12:04, id1)
Table updated with late data (12:17, id3)
Watermarking in Windowed Grouped Counts

Streaming with Flink

- Event-driven; inputs are event logs and outputs event logs
  - Decouples senders and receivers
  - Provides asynchronous, non-blocking event transfer
- Real-time streaming processing with milliseconds latency

System Architecture

- Dispatcher starts and submits the app to the Job Manager, which requests slots from the Resource Manager.
- The Resource Manager allocates slots to Task Manager
- Each Task Manager can execute several tasks (slots) in parallel

Event-Time Processing

- Every record in Flink is accompanied by an event timestamp
- **Watermarks**
  - Watermarks (special records) are periodically injected into the stream
  - A watermark annotated with timestamp "X" means that all records that happen before event time "X" have already been received
- **Checkpointing**
  - Each task maintains its own state, and updates its state whenever a new input has been processes
  - **Checkpoint barriers** are periodically injected into the stream by Job Manager into each of the sources. Each source then checkpoints their current offset (up to which event it has already processed) and forwards the checkpoint barrier
  - Checkpoint barriers cannot be overtaken by other records
  - When a task receives a checkpoint barrier C_i from input stream S_i, it buffers all records from S_i that arrive later than C_i. Those that arrived earlier are regularly processed
  - Records that come from input streams whose barrier have not yet arrived are also regularly processsed
  - Once a task receives **all** the barriers from each of its input stream, it checkpoints the current state of the task
  - Checkpoint barriers are then forwarded, propagating the checkpointing; the task then resumes processing
  - Sinks acknowledge the reception of checkpoint barriers to Job Manager, indicating that a checkpoint is complete

Graph Processing

PageRank

- Page i is more important if it has more in-links j ($j \rightarrow i$)
- Naïve approach: Rank each page i based on its # of in-links
  - Problem: Malicious user can create many dummy pages pointing to i
  - Solution: An in-link from a page j is proportional to its importance r_j
- Weight of an in-link from page j = $\frac{r_j}{d}$, d = # out links from j
- Importance of a page:

$$r_i = \sum_{j \rightarrow i} \frac{r_j}{d_j}$$

Flow Model

- Flow equations include equations for the importance of each page, and the equation that states that the sum of all importance is equals to 1, i.e. $\boldsymbol{r_1 + r_2 + \cdots + r_k = 1}$. This equation enforces uniqueness

- Compute the equations using a **Stochastic Adjacency Matrix $\boldsymbol{M}$**
    - M is a $k \times k$ matrix where k = # pages
    - If $i \rightarrow j$, then $M_{ji} = \frac{1}{d_j}$ else $M_{ji} = 0$ (M_{ji} means row j, col i)
    - All columns must sum to 1 (see "teleports" if it does not)
- **Rank Vector $\boldsymbol{r}$** is a vector containing the importance of each page
    - The sum of all the values in r is 1
    - Initialised with random/equal importance values
- Flow equation: $r^{(t+1)} = Mr^{(t)}$
    - We use this to efficiently solve for r if there are many pages
    - Power iteration: Repeatedly applying the flow equation; r will eventually converge
    - Stop after a fixed # of iterations, or when $|r^{(t+1)} - r^{(t)}| < \varepsilon$

Random Walk

- Based on the probability of a random web surfer reaching a page
- At $t = 0$, start on a random page. At subsequent time steps, follow an out-link from the current page uniformly at random and repeat indefinitely
- Construct a $\boldsymbol{p(t)}$, a vector containing the probability distribution over the pages at time step t
    - $p(0)$ is initialised with equal probabilty values
- Stationary distribution: $p(t+1) = Mp(t)$
    - As t → ∞, $p(t)$ converges, which represents the long-term probability that a random surfer will be at each page
- This is actually exactly the same as flow formulation, just a different way of thinking

PageRank with Teleports

- r may never converge: Consider only 2 pages that link to each other, with $r^{(0)} = [1 \;\; 0]^T$
- r may never converge to what we want due to:
    - **Dead-ends**: A page with **no out-links**, causing importance to "leak out"
        - A dead-end will have its column in M contain all 0s
        - r will eventually sum to 0
    - **Spider traps**: All out-links are within the group, which eventually absorbs all the importance (and the importance of the other pages will approach to 0)
        - r will still sum to 1, but r_k for each page k in the spider trap will have all the importance
- At each time step, with probability $1 - \beta$, the random surfer teleports to some random page (including to itself)
- **Always teleport when arrived at dead-end** m.
    - Since it can teleport anywhere, at dead-end m, column m of M will **contain all** $\frac{1}{N}$ where N = total # of pages, instead of 0
- With teleports, it will never get stuck in a spider trap since it will eventually teleport out of it
- Importance of a page, with teleports:

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

- **Google Matrix $\boldsymbol{A}$** :
  $A = \beta M + (1-\beta)\left[\frac{1}{N}\right]_{N \times N}$, where $M_{*m} = \frac{1}{N}$ if m is a dead-end
- PageRank with teleports equation: $\boldsymbol{r = Ar}$

Topic-Specific PageRank

- During teleport, teleport to only a page from a topic-specific set S instead of from the set of all pages
- For each teleport set S, we get a different vector r_s
- Adjusted google matrix:

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

- Topic can be chosen using the context of the query or asking the user to pick from a menu

Pregel

- Computation consists of a series of supersteps
- Each superstep invokes compute() for each vertex (in parallel)
    - Compute() specifies behaviour at a single vertex v, at a superstep s
- Termination:
    - A vertex can choose to deactivate itself ("vote to halt")
    - A vertex is re-activated if new messages are received
    - Terminate when all vertices are inactive (all "vote to halt") or when predefined # of supersteps is reached

Pregel Architecture

- Vertices are hash partitioned and assigned to workers ("edge cut")
- Each worker maintains the state of its own portion of the graph in **memory**. Computations also happen in memory
- In each superstep, a worker runs compute() on each of its vertices
- Messages from a vertex are either sent to **same worker** (neighbour also belongs to the same partition), or to **another worker** (neighbour belongs to another partition).
- The latter is **buffered locally and sent as a batch**
- Fault tolerance:
    - Checkpointing after each superstep
    - Corrupted workers are reassigned and reloaded from checkpoint

<u>NoSQL</u>

- 3 types: Key-Value, Document, Wide Column

	Key-Value Stores	**Document**
What	Queried by Keys. Values can store complex objects and cannot be queried	DB consists of multiple collections, each containing multiple documents (JSON). Diff documents, diff fields
Complexity	Simple API (GET, PUT)	
Suited For	Small continuous read & writes/Data rarely Δs Store basic info with no clear schema Complex queries rarely required	Many queries based on fields of a document Flexible schema is needed (diff sets of fields)

- **Wide Column Stores**
    - Group related columns into column families
    - Cannot add/delete column families after initialisation, but can add new columns into a family
    - **Sparsity**: If a row does not use a column, that empty column doesn't occupy any space
    - Each row/column intersection can contain multiple cells, each containing a timestamped version of that data for that row and column
    - Pros:
        - Ideal for large dataset that require flexible schemas
        - Columns are stored separately → easily partitioned across distributed nodes
        - Similar data stored together + only retrieve necessary columns → faster I/O
    - Cons:
        - Slow if reading many fields

- **Graph Databases**
    - Nodes store data, edges store the relationship between data
    - Can be processed quickly using graph processing engines

- **Vector Databases**
    - Each row is a vector in d dimensions
    - Similar rows are close to one another in the vector space
    - Pros:
        - Fast similarity search
        - Efficient and scalable storage (just store numbers)

- **BASE Principle**
    - **B**asically **A**vailable: Basic read/write are available most of the time
    - **S**oft State: State of the data can change w/o application interactions due to eventual consistency
    - **E**ventually Consistent:
        - Strong consistency: All readers will reads same result if they read immediately after a write. Readers are blocked during write on the target object
        - Eventual consistency: No blocking; if we wait long enough, eventually all readers will read the same most recent value → favour availability over consistency
    - BASE > ACID if availability is more important than consistency (e.g. BASE social media, ACID for banking applications)
    - Availability/Consistency can be configured

- **Denormalization**
    - WHY: JOIN is difficult in NoSQL:
        - Key-Value Stores: Cannot join by value (values cannot be queried)
        - Document Stores: Documents can have different fields
    - SOLUTION: Design your tables around the queries we expect to receive, then duplicate data since <u>data storage is cheap</u>
    - ✓ Easy to query data as all necessary information is available in a single document
    - ✗ Changes to one field may need to be propagated to multiple tables

- **Data Partitioning**
  - **Table Partitioning**: Different tables/collections on different machines
    - ✓ Easy to partition
    - ✗ Limited scalability - each table cannot be split across multiple machines (esp. large tables)
  - **Horizontal Partitioning/Sharding**: Different tuples on different nodes, partitioned using the partition key
    - Use frequently used columns that are used for filtering tuples/"group by" as partition key
      - ✓ Partition Pruning: Queried data will be in the same few partitions → Efficient query
      - ✗ Data skewness can lead to inbalanced partitions, leading to poor parallelization
    - Range Partition: Partition by range of partition key values
      - ✓ Efficient range queries
      - ✗ Imbalanced shards, but the balancer tries to keep shards balanced
    - Hash Partition: Partition by range of hash values of partition key
      - ✓ Automatically balances the shards
      - ✗ Add/remove a node → need to redo partition → move data around → incurs I/O (Solution: Consistent hashing)
      - ✗ Range query may involve many partitions
  - **Horizontal Partitioning – Consistent Hashing**
    1. Arrange all the possible hash values on a circle
    2. Place each node ("marker") on the circle
    3. For each tuple, hash its partition key to get the hash value, then assign it to the node **clockwise** after it on the circle
    - Deleted node: simply re-assign all that node's tuples to the node clockwise after it
    - Added node: simply split largest node into two
    - ✓ Efficient node add/removal w/o incurring large I/O
    - ✓ Simple replication strategy: Replicate a tuple in the next k nodes clockwise after the primary node that stores it
    - ✓ Can have multiple markers per node – when a node is removed, its tuples will not all be reassigned to the same node → better load balancing

NoSQL Pros/Cons

Pros	Cons
Horizontal Partitioning: Enables high scalability. If data increases, simply partition it into more shards → improved speed from parallelization + cheaper	**No declarative query language**: Query logic (e.g. JOIN) may have to be handled on the application side → additional programming required
Relaxed consistency guarantees: Availability > consistency → High performance and availability	**Weaker consistency guarantees**: Stale data → may have to be handled on the application side
Flexible/Dynamic schemas: Adapts well to changing requirements – ideal for storing unstructured data	