Introduction

Challenges of Big Data

- Volume: Scale of data
  - Impacts performance, cost, reliability, and algorithm design complexity
- Velocity: Speed of streaming/real-time data
  - Impacts performance, cost, reliability, and algorithm design complexity
- Variety: Different formats of data
  - Each new data format → New system for handling
- Veracity: Uncertainty of data
  - E.g., Dirty and noisy data, Data provenance/source, Data uncertainty

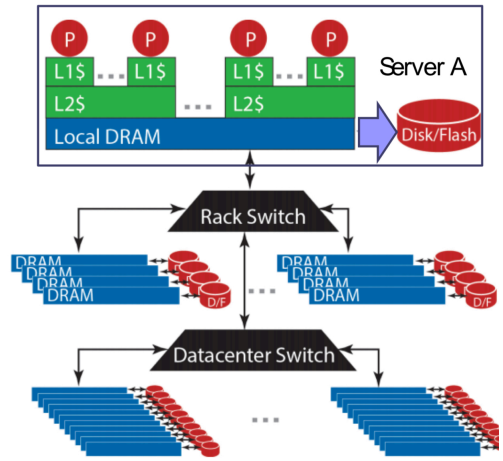
Infrastructure for Big Data

- Utility Computing: Computing resources as a metered service
  - Pros: Scalability (Infinite capacity), Elasticity (Scale up or down on demand)
- To enable utility computing:
  - Virtual Machines: Enable sharing of hardware resources by running each application in isolated virtual machine
    - High overhead: Each VM has its own OS
  - Containers: Lightweight sharing of resources by isolating applications, while sharing the same OS
- Infrastructure as a Service (IaaS): User rents VM and decides what to run (e.g., EC2)
- Platform as a Service (PaaS): Platform to host, while taking care of hardware for you (e.g., Google App Engine)
- Software as a Service (SaaS): Existing app that does everything for you (e.g., Gmail)

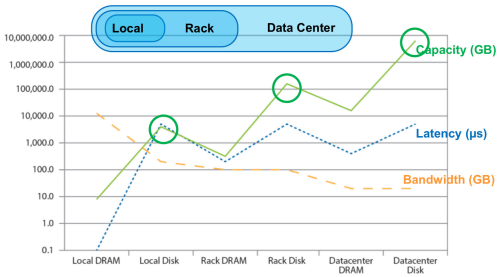
Bandwidth vs. Latency

- Bandwidth: Maximum amount of data that can be transmitted per unit time
  - Throughput: Amount of data *actually* transmitted per unit time
  - Bandwidth of whole path ≈ Minimum bandwidth along the path
- Latency: Time taken for 1 packet to go from A to B
  - Latency of whole path ≈ Sum of latency along the path

Storage Hierarchy in Data Center



- Local server → Rack → Datacenter
- Dynamic Random Access Memory (DRAM): Fast but limited capacity
  - Data in disk must first be loaded into DRAM for usage
- Disk: Slow but large capacity
- Flash: In between DRAM and disk



- Disk's capacity > DRAM's capacity
- Capacity increases as we go from local → rack → datacenter, since the capacity sums up
- Disk reads have higher latency and lower bandwidth than DRAM
- DRAM latency increases as we go from local → rack → datacenter, due to switch latency
  - Go out of local server → Slows by several magnitudes due to switch latency, but slows within same magnitude when going around multiple machines
- Disk latency similar because disk latency is much larger than switch latency
- Bandwidth decreases as we go from local → rack → datacenter, since bounded by switch

Big Ideas in Data Centers

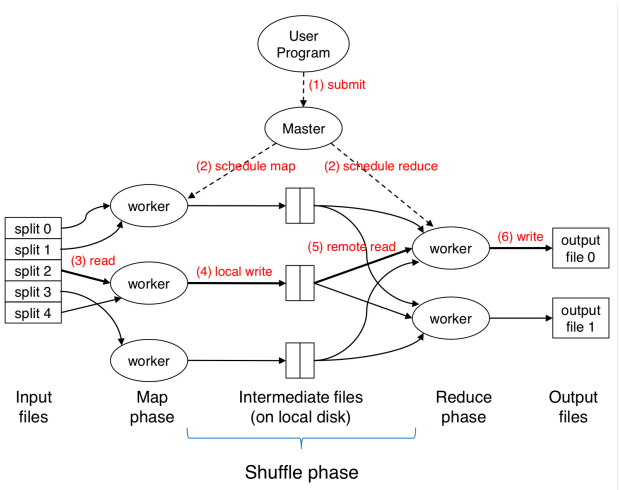
- Scale out, not up
  - Pros: Better reliability, Lower cost

- Seamless scalability: Ideally, if task scales linearly to number of machines, use more machines for better performance
- Move processing to the data, rather than data to the task, via task scheduling
- Process data sequentially and avoid random access

MapReduce

- Motivation: MapReduce provides functional abstraction for 2 operations to solve typical big data problems that include:
  1. Map: Iterating over a large number of records and extracting something from each record
  2. Shuffle: Shuffle and sort intermediate results
  3. Reduce: Aggregate intermediate results
- Interfaces for the user to specify:
  - $\text{map}(k_1, v_1) \rightarrow \text{List}(k_2, v_2)$
    - Input: Key-value pair representing a record
  - $\text{reduce}(k_2, \text{List}(v_2)) \rightarrow \text{List}(k_3, v_3)$
    - Input: All values with same key are grouped and sent to same reduce task

Implementation



1. Submit: User submits program (`map()`, `reduce()`, and configurations like number of workers)
2. Schedule: Master node schedules resources for tasks and does not handle any data
3. Read: Input files split into tasks of 128MB for workers to execute tasks 1 at a time
   - Can be local or remote read, but local preferred
   - Map phase: Worker iterates and computes over each key-value tuple
4. Local write: Map worker writes outputs of `map()` to intermediate files on local disk
   - Files can be partitioned into chunks depending on number of reduce tasks
   - Each chunk is sorted by key
   - Can be in DRAM, but it's a trade-off between capacity and performance
5. Remote read: Reduce worker responsible for 1 or more keys
   - Reducer processes keys in sorted order
   - For each key, worker reads the needed key-value pairs from corresponding partition of each mapper's local disk
   - Reduce phase: Reducer receives needed key-value pairs and computes reduce function on values of each key
6. Write: Output of `reduce()` written to HDFS

Details

- Choice of split size:
  - Too big: Limited parallelism
  - Too small: High overhead since master node can be overwhelmed by scheduling, Devolves into random access, instead of sequential
- Barrier between map and reduce phases, since we must finish map before starting reduce

Partitioner and Combiner

- Optional functions in local write step that optimize disk and network traffic
- Partitioner: Custom partition defined by user to better spread load among reducers
  - Motivation: By default, assignment of keys to reducers determined by hash function. What if some keys have more values than others?
- Combiner: Locally aggregate output from mappers to reduce disk writes
  - Motivation: Writing `map()` output without aggregating is expensive, since need to write and read intermediate results
  - "Mini-reducers": Can reuse reducer function
  - User must ensure correctness: Combiner should not affect correctness, whether the combiner runs **0, 1, or multiple times**
    - Generally, reduction function must be binary operation that is associative ($a + (b + c) = (a + b) + c$) and commutative ($a + b = b + a$)
      - E.g., `sum()`, `min()`, `max()`
      - Incorrect: Mean, Minus
  - Possible to maintain state within same map task (e.g., hash table to count words and counts across all lines in a task) $\rightarrow$ Trade-off between reducing disk/memory I/O and increasing the memory working set

Performance Guidelines for Algorithm Design

- Linear scalability: More nodes can do more work in same time
- Minimize disk I/O: Sequential vs. Random
- Minimize network I/O: Send in bulk vs. Send in small chunks
- Reduce memory working set (Portion of memory actively being used during execution): Reduce needed memory and out-of-memory errors

Hadoop Distributed File System

- Assumptions:
  - Commodity hardware, instead of "exotic" hardware
    - Commodity hardware: Nodes with both compute and storage
    - "Exotic" hardware: Rich in either compute or storage (e.g., supercomputers) $\rightarrow$ Requires high network traffic to move data to compute nodes $\rightarrow$ High I/O
  - High component failure rates
  - "Modest" number of huge files, rather than lots of small files
  - Large sequential reads, instead of random access
- Design decisions:
  - Files stored as fixed-size chunks
  - Reliability through replication: Each chunk is has 3 copies by default (2 in same rack and 1 in different rack to balance between reliability and performance)
  - Designed for write-once, read-many
  - Single node to coordinate access
- Same nodes are used for both HDFS and Hadoop $\rightarrow$ Hadoop tries to schedule map tasks to run on machines that already contain the needed data (Data locality; "Move processing to data")

Namenode

- Manages file system namespace
  - Holds directory structure, metadata, file-to-block mapping, access permissions
  - Directs clients to datanodes for reads and writes
  - No data goes through the namenode
- Maintains overall health
  - "Heartbeat": Periodic communication with datanodes
  - Block rebalancing and garbage collection
- What if namenode's data is lost?
  - All files on file system cannot be retrieved, since no way to reconstruct from raw data blocks
  - Hadoop provides resilience through backups and checkpointing

Writing Data

1. HDFS client sends request to write data
2. Namenode checks file namespace, allocates blocks, and directs request to datanode and replicas
3. HDFS client sends file blocks to datanode **without** going through namenode
4. Datanode communicates with next datanode for replication

Reading Data

1. HDFS client sends request to read data
2. Namenode searches file namespace and directs request to **available** datanode that is **closest to client machine**
3. HDFS client communicates with datanode to receive each file block

MapReduce and Relational Databases

- Motivation: Develop large-scale relational databases using MapReduce

Projection (π_c)

> `SELECT product_id, price FROM Sales`

- Map: Take in tuple with tuple ID as key and emit new tuple with specified attributes
- Reduce: No reducer needed
- No need to shuffle

Selection (σ_p)

> `SELECT * FROM Sales WHERE price > 10`

- Map: Take in tuple with tuple ID as key and emit only tuple that fulfills the predicate
- Reduce: No reducer needed
- No need to shuffle

Group By

> `SELECT product_id, AVG(price) FROM Sales GROUP BY product_id`

1. Map over tuples and emit `<product_id, price>`
2. Shuffle step will automatically group tuples by key
3. Compute average in reducer, and optimize with combiners

Inner Join

> `SELECT * FROM Sales INNER JOIN Products ON Sales.product_id = Products.product_id`

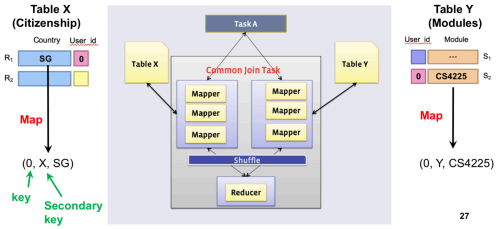
Broadcast Join

- Aka Map Join
- Requires 1 of the tables to fit in main memory of server
  - Rationale: To join faster than $O(RS)$, need random access $\rightarrow$ Conversion of small table into hash table $\rightarrow$ Fast if hash table is in main memory ($O(R+S)$), but slow if hash table is in hard disk due to I/O
- Steps:
  1. Convert small table into a hash table, with key as the join key
  2. Store a copy of hash table in memory of all mappers
  3. Mappers iterate over the big table and **hash join** the records with the hash table (i.e., $R_i \bowtie S$ where R_i is the i-th chunk of big table R)

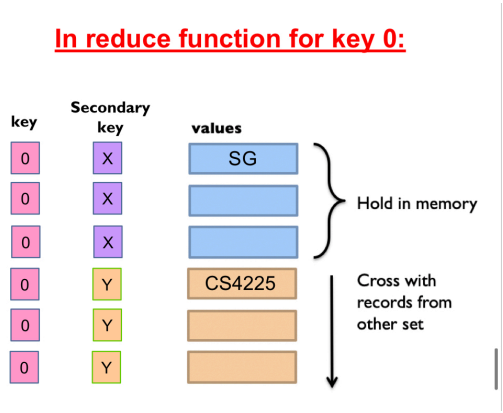
Reduce-side Join

- Aka Common Join

- Slower than broadcast join
- Steps:



1. Different mappers operate on each table and emit records with key as the join key
2. Reducer: Use secondary sort to ensure all keys from table X arrive before table Y
   - Secondary Sort: Sort such that both keys and values are sorted using a custom comparator and partitioner



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

MapReduce and Data Mining

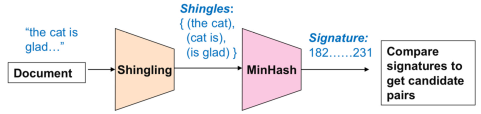
- Motivation: Perform large-scale data mining techniques using MapReduce

Similarity Search

- Motivation: Finding "similar" objects. For instance:
- Distance Metrics: Lower distance $\leftrightarrow$ Higher similarity
  - Euclidean distance: $d(a,b) = \sqrt{\sum_{i=1}^{D}(a_i - b_i)^2}$
  - Manhattan distance: $d(a,b) = \sum_{i=1}^{D}|a_i - b_i|$
  - Cosine similarity (For direction): $s(a,b) = \frac{a \cdot b}{||a|| \cdot ||b||}$
  - Jaccard similarity (Between sets A and B): $s_{\text{Jaccard}}(A,B) = \frac{|A \cap B|}{|A \cup B|}$
  - Jaccard distance: $d_{\text{Jaccard}}(A,B) = 1 - s_{\text{Jaccard}}(A,B)$

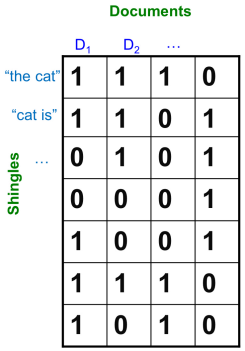
Finding Similar Documents

- Problems:
  - All pairs similarity: Given large number N of documents, find all similar pairs with Jaccard distance below a threshold
  - Similarity search: Given a query document D, find all documents which are similar with D



Shingling

> Convert documents to sets of short phrases ("shingles")



- K-shingle (or k-gram): Sequence of k tokens that appear in the document
  - E.g., Above example is set of 2-shingles for D_1
- Matrix representation: Columns $\rightarrow$ documents, Rows $\rightarrow$ shingles
  - Naive: Similarity between documents can be measured via Jaccard Similarity $\rightarrow$ Use logical AND for $\cap$ and logical OR for $\cup$

Min-Hashing

> Min-Hash each document set to short signature, while preserving similarity

- Motivation: Above method using matrix is slow $\rightarrow O(N^2)$ to find all similar pairs
- Key idea: Min-Hash each column C to small signature $h(C)$, such that:
  - $h(C)$ is small enough to fit into main memory
  - Highly similar documents have the same signature $h(C) \rightarrow$ Candidate pairs
- Therefore, need to find the hash function h such that:
  - If $\text{sim}(C_1, C_2)$ is high, then high probability that $h(C_1) = h(C_2)$
  - If $\text{sim}(C_1, C_2)$ is low, then high probability that $h(C_1) \neq h(C_2)$

- Min-Hashing:
  - Steps:
    1. Given a set of shingles for a document, hash each shingle to an integer
    2. Compute minimum of these integers
  - Key property: Given h is min-hash, $\Pr(h(C_1) = h(C_2)) = s_{\text{Jaccard}}(C_1, C_2)$
- Candidate pairs: Documents with the same signature
  - Can either directly output candidate pairs or compare them using above method using matrix
  - In practice: 1 hash function may not be reliable due to hash conflicts $\rightarrow$ Use multiple hash functions to generate N signatures for each document $\rightarrow$ Candidate pairs are document pairs that meet some sufficient number of matching signatures

MapReduce Implementation

- Assume only 1 hash function
- Map:
  - Input: `<document_id, document>`
  1. Read over document and extract shingles
  2. Hash each shingle and take the minimum to get min-hash signature
  3. Emit `<signature, document_id>`
- Reduce:
  - Input: `<signature, [doc_1, doc_3, ...]>`
  1. Receive all documents with given min-hash signature
  2. Generate all candidate pairs from these documents
  3. Optional: Compare each candidate pair if they are actually similar
- Other considerations:
  - To maximize parallelism, need to have many equal tasks
    - Maximum number of map tasks: Depends on number of chunks from documents, which is usually sufficient
    - Maximum number of reduce tasks: Depends on number of distinct signatures $\rightarrow$ Need uniform hash function
  - Highest usage of I/O occurs when reading over documents
  - If using N hash functions, naive way can be repeat this for N times then tally the candidate pairs afterwards

Clustering

- Clustering: Separate unlabelled data into groups of similar points

K-Means Algorithm

Algorithm

1. Pick K random points as cluster centers
2. Repeat until no more updates
   1. Assign each point to nearest cluster
   2. Move each cluster center to average of its assigned points

MapReduce Implementation v1

- For a single iteration of k-means:

```
class Mapper:
    def configure():
        c = loadClusters()

    def map(id i, point p):
        n = nearestClusterId(clusters c, point p)
        p = extendPoint(point p)
        emit(clusterId n, point p)

class Reducer:
    def reduce(clusterId n, points [p1, p2, ...]):
        s = initPointSum()
        for point p in points:
            s = s + p
        m = computeCentroid(point s)
        emit(clusterId n, centroid m)
```

- Problem: High network I/O traffic for `emit(clusterId n, point p)`
  - Since need to emit for all points and shuffle, which is random access
  - Disk I/O between mappers and reducers: $O(nmd)$ where n is number of points, m is number of iterations, and d is dimensionality
  - Slower than `map()` reading over all points, which is sequential and read from HDFS

MapReduce Implementation v2 with In-Mapper Combiner

```
class Mapper:
    def configure():
        c = loadClusters()
        h = {}

    def map(id i, point p):
        n = nearestClusterId(clusters c, point p)
        p = extendPoint(point p)
        h[n] = h[n] + p

    # Run after all map() function calls are done
    def close():
        for clusterId n in h:
            emit(clusterId n, point h[n])
```

```
class Reducer:
    def reduce(clusterId n, points [p1, p2, ...]):
        s = initPointSum()
        for point p in points:
            s = s + p
        m = computeCentroid(point s)
        emit(clusterId n, centroid m)
```

- Intuition: Each value of hash table contains "sum" of points belonging to cluster n
- Disk I/O between mappers and reducers: $O(kmd)$ where k is number of cluster centers, m is number of iterations, and d is dimensionality
  - Much better than v1, since $n >> k$
- In-Mapper Combiner: Accumulates aggregate results in-memory within the mapper without writing
  - User controlled
  - If implemented correctly, can be more efficient at the expense of memory
  - Normal combiners are different: Since they still write the aggregate results and are more system controlled

NoSQL

> "Not only SQL"

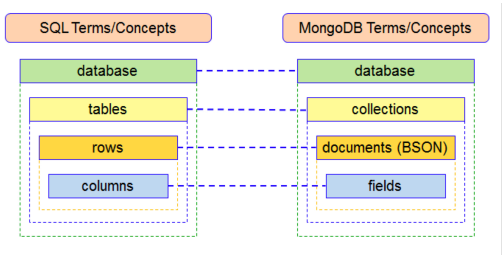
- NoSQL: Non-relational databases
- Misconception: NoSQL databases often still have SQL-like query languages
- Characteristics:
  - Volume and velocity: Horizontally scalable
  - Variety: Flexible schemas
  - Weaker concurrency model than RDBMS

Key-Value Stores

> Stores association between keys and values

- Keys usually primitive and can be queried
- Values can be primitive or complex and usually cannot be queried
- API: Get, Put
- Good for:
  - Small continuous reads and writes, since direct access with keys
  - Storing basic information
  - When complex queries are not required
- Use cases: Cache, User sessions
- Non-persistent: In-memory hash table (e.g., Redis)
- Persistent: Data stored to disk (e.g., Dynamo)

Document Stores



- E.g., MongoDB
- Document: JSON-like object with fields and values
- Allows more complex querying (e.g., CRUD)
- Good for:
  - Dynamic/evolving schema, since just need to adjust JSON document structure
  - Nested data, since JSON supports nesting

Graph Databases

> Store highly interconnected data, represented as nodes and edges

- Good for complex relationship queries
- Use cases: Social networks, Geospatial analysis

Vector Databases

> Store vectors (i.e., each row represents a point in d dimensions)

- Good for:
  - Storing high-dimensional vector embeddings
  - Fast similarity searches: Given a point, retrieve similar neighbors from database

Spark

- Motivation: Hadoop MapReduce has problems
  - Network and disk I/O: Intermediate data has to be written to local disks and shuffled across machines
  - Not suitable for iterative tasks: Since each iteration is 1 MapReduce job
  - Solution: Spark stores most of its intermediate results **in memory**
    - If too big, then spills into disk
- Architecture:
  - Driver process: Responds to user input/code and distributes work to executors
    - Local mode: All processes run on same machine (usually for debugging)
  - Executors: Run code assigned by driver process in parallel and send the results back to the driver process
  - Cluster manager: Allocates resources when application requests it
- Extensible for all storage solutions, since difficult to migrate data

Resilient Distributed Datasets (RDD)

> Resilient: Fault tolerance via lineage; Distributed datasets: Collection of Java/Scala objects that is distributed over machines

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

# RDD from text file, read on each executor in parallel
textFile = sc.textFile("File.txt")
```

- Immutable
- Transformation: Transforms RDDs into RDDs
  - **Lazy**: Transformation will not execute anything, until an action is called on it
    - Pros: Spark can optimize query plan to improve speed
  - E.g., `map`, `order`, `groupBy`, `filter`

```
nameLen = dataRDD.map(lambda s: len(s))
```

- Action: Triggers computation of result from series of transformations
  - E.g., `show`, `count`

```
nameLen.collect()
```

- Transformations and actions are executed **in parallel** on each machine $\rightarrow$ Results only sent to driver in final step

Caching

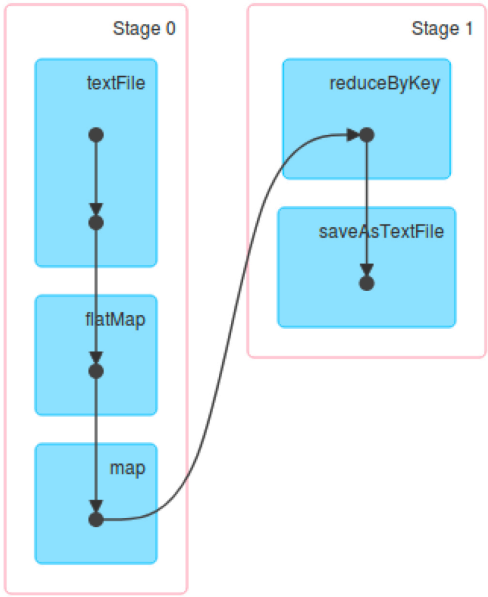
> Saves an RDD to memory of each executor, where each executor stores the partition that it computed

```
lines = sc.textFile("")
errors = lines.filer(lambda s: s.startsWith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache() # Only caches after Action #1

# Action #1: Computes and caches messages
messages.filter(lambda s: "mysql" in s).count()

# Action #2: Directly filter on cached messages
messages.filter(lambda s: "php" in s).count()
```

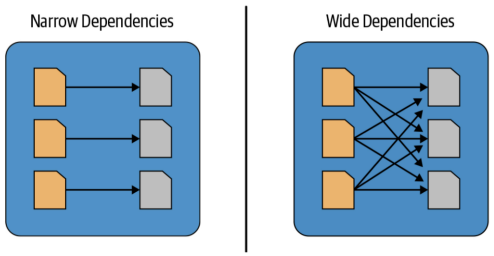
- `cache()`
- `persist(options)`: Used to save an RDD to memory or disk (if RDD too big to fit in memory)
- When to cache?
  - RDD is expensive to compute and needs to be re-used multiple times
  - Memory limitations: If executors do not have enough memory, they will evict least recently used RDDs

Lineage

> Directed Acyclic Graph (DAG) of all RDD objects and how they are transformed



- Transformations construct DAG; actions trigger computations
- Narrow dependencies: Each partition of parent RDD is used by ≤ 1 partition of child RDD
  - E.g., `map`, `flatMap`, `filter`
- Wide dependencies: Each partition of parent RDD is used by > 1 partitions of child RDD
  - E.g., `reduceByKey`, `groupBy`, `orderBy`



- Consecutive narrow dependencies are grouped together as stages

    - Within stages: Spark performs consecutive transformations on the same machine
    - Across stages: Spark needs to shuffle data across partitions by writing to disk (Similar to shuffling in MapReduce)
        1. Parent partitions write intermediate results to **local disk**
        2. Child partitions remotely read intermediate results and combine
        3. Child partitions write combined results into local disk
            - Costly! → Shuffling/wide dependencies should be minimized
- Fault tolerance through lineage: If a worker node goes down, replace by new worker node and use the DAG to recompute data in lost partition
    - For wide dependencies: Read combined results into local disk
    - Both driver and executors store lineage information
    - Spark does not use replication like in Hadoop, since duplicating in memory is expensive
        - Only incurs extra work if failure

DataFrames

> Table of data

- Higher-level interface, compared to RDD
    - All DataFrame operations compiled down to RDD operations
- Rows are partitioned across servers
- Can use SQL queries to transform DataFrames

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

Datasets

> Similar to DataFrames, but type-safe

- Provides best of RDD (customization) and DataFrame
- Type-safe: Each row has type (e.g., `Flight`)
    - DataFrame is special case of Dataset, where type of Dataset is fixed to `Row`

Spark SQL

> Module in Spark built on top of RDDs

- Significance:
    - Provides unified interface to other Spark components
    - Provides SQL-like queries on DataFrames and Datasets for Java, Scala, Python
- RDD vs. DataFrame:
    - RDD: Tells Spark *how* to do → Intention unknown to Spark → Optimization must be done manually by user
    - DataFrame: Tells Spark *what* to do → Spark can optimize
- Catalyst optimizer: Takes in query and converts it into RDD execution plan
    1. Analysis: Check with column information
    2. Logical optimization: Predicate pushdown before joins, column pruning, etc.
    3. Physical planning: Propose several physical plans and use most cost-effective plan
    4. Code generation
        - Via Project Tungsten: Focuses on improving hardware efficiency of Spark applications

Machine Learning with Spark ML

Data Preprocessing

- Suitable for Spark, since Spark can transform big data well
- Common problems and preprocessing steps:
    - Missing values
        - Delete rows
        - Impute values using mean/median/regression model (Optional: Add dummy variable to indicate variable was missing or not)
    - Categorical features
        - Categorical encoding: Convert categorical feature to numerical feature, where numerical values are usually assigned to represent an ordinal relationship
        - One-hot encoding: Convert discrete feature to series of binary features
            - Useful when you don't want to imply an ordinal relationship
    - Normalization: Clipping, log transformation, standard scaler

Stream Processing

- Stream: Data arrives over time
    - vs. batch processing: Operates on full dataset
    - Potentially infinite amount of data: System cannot store entire stream due to limited memory
    - Input ports: Data from sensor, TCP connection, file stream, message queue, etc.
    - Tuple: Element of a stream
    - Use case: Constantly arriving over time + high volume + requires immediate attention
        - E.g., financial transactions, stock trades
- Stateful stream processing: Ability to store and access intermediate data
    - vs. record-at-a-time transformations: Does not store intermediate data → If failure, no way to recover previous results
    - Allows "exact-once": Each record only processed once

Spark: Micro-batch Streaming

> Micro-batch processing model: Divides input data into micro-batches and processes each batch using Spark

- Pros:
    - Can use original Spark batch processing capabilities → As easy as writing batch pipelines with automatic optimizations

    - Failure recovery using lineage
    - Deterministic nature ensures "exact-once"
- Cons: Latencies between batches, depending on batch size
    - Too big: More delay to wait for batch, but higher throughput
    - Too small: Less delay, but lower throughput due to overhead
- Defining a streaming query:
    1. Input source
    2. Data transformation: Same as batch processing
    3. Output sink and output mode (e.g., append, complete mode)
    4. Processing details
        1. Triggering details: Batch size (i.e., when to trigger processing of new streaming data)
        2. Checkpoint location: Periodically store query process information and intermediate states in remote storage for failure recovery
- Considerations:
    - May need to run 24/7 → Need to allocate cluster resources properly
    - Number of partitions for shuffles should be set lower in stream processing than batch processing, since more overhead with mini-batches
    - Set source rate limits for stability
    - Have multiple streaming queries in same Spark application for stable amount of computation

Stop-the-world Checkpoint

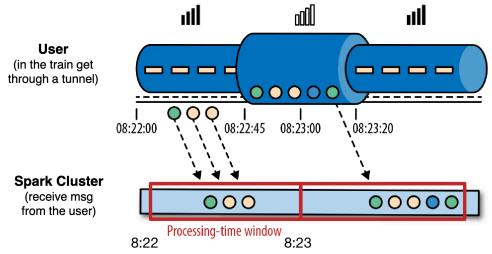
- Aka consistent checkpoints
- To save checkpoint:
    1. Stop ingestion of all input streams
    2. Wait for processing of all in-flight data
    3. Checkpoint by copying state of each task to remote, persistent storage
    4. Resume ingestion of all streams
- To recover from failure:
    1. Restart the whole application
    2. Reset the states of all stateful tasks to latest checkpoint
    3. Resume the processing of all tasks
- Pros: Simple
- Cons: Stopping entire application to save a checkpoint leads to delays
    - Fine for micro-batch streaming, since micro-batches cause more delays
    - Too slow for native streaming

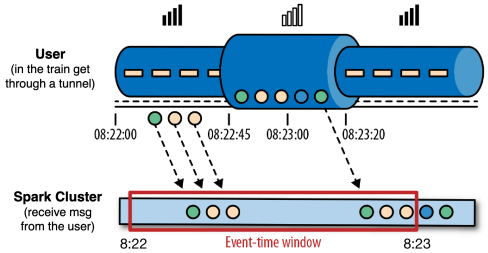
Data Transformation

- Stateless transformation: Processes each row individually without needing previous rows
    - E.g., `select`, `map`, `filter`
- Stateful transformation: Derives partial results from each row and aggregates it with previous results via state
    - E.g., `df.groupBy().count()`
    - Stateful streaming aggregations not based on time: E.g., `sum`, `mean`, `count`

Stateful Streaming Aggregations Based on Time

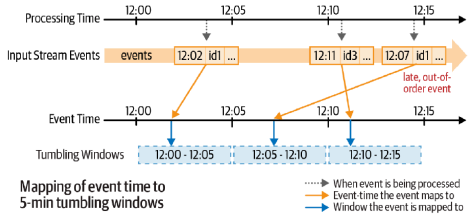
- Aggregate based on which time?
    - Processing time: Time when machine processes stream
    - Event time: Time when event happened
    - Depending on which time to use for aggregation, same time interval can include different events



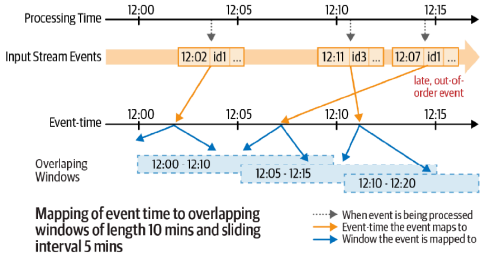


- **Event time preferred** for aggregation
    - Rationale: Operations based on event time are deterministic
    - Using processing time is easier, but indeterministic due to unpredictable network conditions → Difficult to recover during failure

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



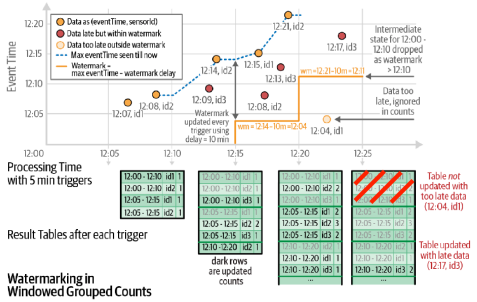

```
sensorReadings
    .groupBy("sensorId", window("eventTime", "10 minutes", "5 minutes"))
    .count()
```



Watermarking

- Motivation: Streams are infinite → How to determine when to start dropping records?
- Watermark: $\text{Max. event time} - \text{Watermark delay}$
  - Implies that user okay with events not being captured before then → Events before it may or may not be counted, and events after it are definitely counted

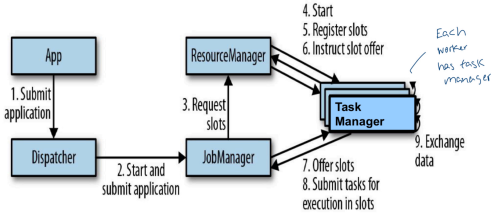
```
sensorReadings
    .withWatermark("eventTime", "10 minutes")
    .groupBy("sensorId", window("eventTime", "10 minute", "5 minute"))
    .count()
```



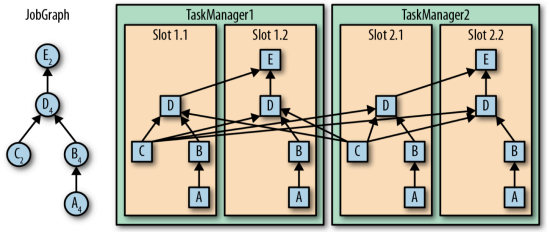
- At 12:25, state with 12:00-12:10 dropped since before watermark
- If data with event time 12:07 with id 1 arrives at 12:23: Still captured at 12:25, even though event time before watermark, since still got open windows

Flink: Native Streaming

> Flink: Distributed system for stateful data stream processing

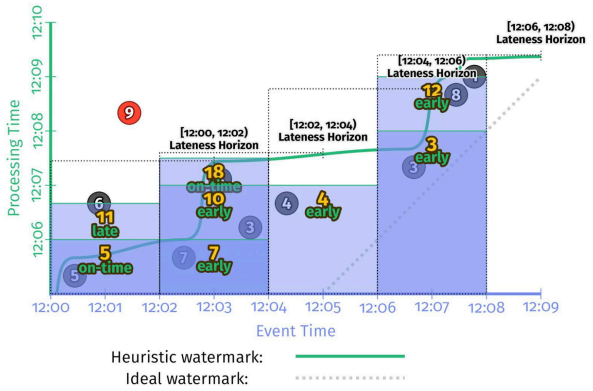


- Event-driven: Usually used with message queue service like Kafka
  - Pros:
    - Persistent messages even after read → Better failure recovery
    - Decoupling between sender and receiver
    - Asynchronous, non-blocking event transfer
- Task manager: Offers processing slots to control number of concurrent tasks it can execute
  - Processing slot: Executes 1 parallel task of application
  - Manages data transfer continuously from sender tasks to receiver tasks
    - Records buffered before transfer



Event-time Aggregation

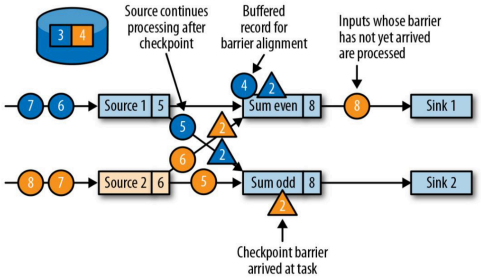
- Every record contains event timestamp
- Watermark: Special record injected by user to trigger calculation for event-time window
  - Contains timestamp representing user's estimate of progress of event time
  - Different from watermark in Spark



- Task: Aggregate records using event-time window
- Heuristic watermark: Injected by user
- At 12:06, Flink receives watermark with 12:02, which marks the "end" of this event-time window and triggers calculation
  - Less delay than Spark: Since Flink's calculation time is flexible, whereas Spark's calculation is on a fixed interval
- Early triggering possible (e.g., 7)
- Late triggering possible via separate lateness horizon function (Similar to Spark's watermark)

Distributed Checkpoint

- Motivation: Stop-the-world checkpoint causes too much delay for Flink
- Pros:
  - Does not pause the entire application → Some tasks continuing processing while others save their state as a checkpoint → Checkpointing decoupled from processing
- To save a checkpoint:
  1. Job manager initiates checkpoint by injecting checkpoint barrier to sources
     - Checkpoint barrier: Special record that contains a checkpoint ID and acts like a divider between records
  2. Sources checkpoint their state and emit checkpoint barrier to following tasks
  3. Tasks wait to receive barrier on each input partition
     1. If not yet arrived, process records regularly
     2. If arrived from 1 input partition, buffer subsequent records from partition (since these records are after the barrier and are included in the next checkpoint)
  4. Tasks checkpoint their state, once all barriers have been received
  5. Tasks forward the checkpoint barrier, and continue regular processing
  6. Sink acknowledges reception of checkpoint barrier to job manager



Stream Processing in Spark vs. Flink

- Spark:
  - Micro-batch streaming → Latency of few seconds
  - Watermark: Configuration to determine when to drop late events
  - Stop-the-world checkpoint: Synchronous → Processing resumes after checkpointing finishes
- Flink:
  - Real-time streaming → Latency of milliseconds
  - Watermark: Special record to determine when to trigger calculation of event-time window
    - Dropping of late events uses separate late handling function
  - Distributed checkpoint: Asynchronous → More efficient and lower latency

Graphs

PageRank

- Goal: Rank "importance" of pages on the web
  - Node: Web page
  - Directed edge: Hyperlink
- Defining "importance":
  - Try 1: Rank each page based on number of in-links
    - Rationale: In-links are harder to manipulate (i.e., spam lots of out-links on personal website)
    - Problem: Can create lots of dummy pages with links to 1 page
  - Try 2: Make number of "votes" that a page has proportional to its own "importance" (i.e., less importance → Less votes)
    - Recursive definition

Flow Formulation

- Importance r_j for page j:

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

- d_i: Number of out-links of node i
- Using above equation for all n nodes, we get n equations and n unknowns, but 1 redundant equation (linearly dependent due to cyclic nature of links) $\rightarrow$ No unique solution
  - Solution: Add additional condition $\sum_{i \in N} r_i = 1$, where N is set of all nodes
- Thus, we can solve this equations using substitution or Gaussian elimination
  - Problem: Need better method for huge graphs

Matrix Formulation

- Define matrix M:
  - If $i \rightarrow j$, then $M_{ji} = \frac{1}{d_i}$
  - Else, $M_{ji} = 0$
  - M is a column stochastic matrix: Columns sum to 1
- Define rank vector r, where r_i is importance score of page i
  - $\sum_i r_i = 1$
- Flow equation in matrix form:

$$M \times r = r$$

  - Equivalent to flow equation for all nodes
  - Intuition: Each row j in M represents "vote weightage" flowing into node j, which is then multiplied with r

Power Iteration

1. Suppose there are n web pages
2. Initialize: $r^{(0)} = [1/n, \dots, 1/n]^T$
3. Iterate: $r^{(t+1)} = M \times r^{(t)}$
4. Stop when $|r^{(t+1)} - r^{(t)}|_1 < \epsilon$
   - L_1 norm: $|x|_1 = \sum_i |x_i|$

Random Walk Formulation

- Imagine a random web surfer:
  1. At time $t = 0$, surfer starts on a random page
  2. At any time t, surfer is on some page i
  3. At time $t + 1$, surfer follows an out-link from i uniformly at random
  4. Repeat
- Define probability distribution vector $p(t)$, where $p(t)_i$ is the probability that the surfer is at page i at time t
  - $p(0) = [1/n, \dots, 1/n]^T$
  - $p(t + 1) = M \times p(t)$
  - As $t \rightarrow \infty$, $p(t)$ approaches steady state, which are the page importance scores
- Equivalent to flow formulation!
  - Intuition: Each row j in M represents probability of random walks from all nodes into node j, which is then multiplied with $p(t)$

PageRank with Teleports

- Above PageRank does not always converge correctly

Spider Trap

- Spider trap: Subset of nodes with in-links into the group, but no out-links out of the group
  - 1 node pointing to itself is also a spider trap
  - Random surfer stuck in group $\rightarrow$ Spider trap absorbs all the "importance"
- Solution: At each time step, random surfer has 2 options:
  1. With probability β, follow an out-link at random
  2. With probability $1 - \beta$, teleport to some random page
     - β usually around 0.8 to 0.9

Dead-end

- Dead-end: Single page node with in-links, but no out-links
  - Random surfer cannot go anywhere $\rightarrow$ Turns all importance scores to 0, since it takes in "importance", but does not give out any "importance"
  - M not column stochastic (i.e., column i, which corresponds to out-links of dead-end node i, are all 0)
- Solution: If run into dead-end, teleport to some random page
  - Graph equivalent: For each dead-end m, add edge from m to every node (including itself)
  - Matrix equivalent: For each dead-end m, preprocess random walk matrix M by setting entire column m to $1/n \rightarrow$ Make matrix column stochastic
    - Intuition: Each dead-end has equal probability to teleport to each node

Formulation with Teleport

- Assume dead-ends resolved with preprocessing
- PageRank equation:

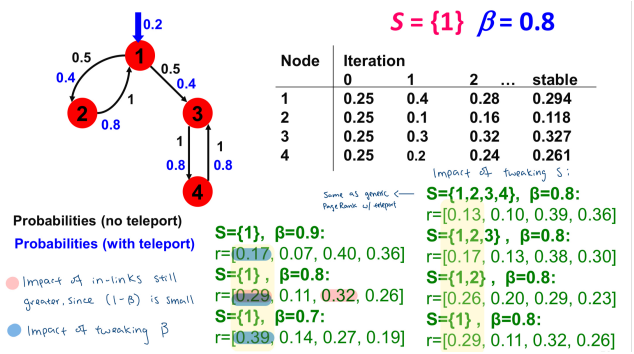
$$r_j = \sum_{i \rightarrow j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{n}$$

  - Recall: r_i is probability that surfer is at page i
  - Left term intuition: For each in-link into node j, there is β probability of surfer coming in
  - Right term intuition: From every node, there is $(1 - \beta)$ probability of surfer teleporting in
    - $(1 - \beta) \sum_i \frac{r_i}{n} = (1 - \beta) \frac{1}{n}$
- In matrix form: $r = A \times r$, where

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

Topic-sensitive PageRank

- Goal: Evaluate pages by their popularity and by how close they are to a particular topic
- Idea: Teleport to only topic-specific set of "relevant" pages
- Matrix formulation: Given set S of topic-specific pages
  - If $j \in S$, $A_{ji} = \beta M_{ji} + (1-\beta)\frac{1}{|S|}$
    - Big picture: Each row j adds additional probability from teleport
    - Intuition: Only nodes in S have probability of surfer teleporting in
  - Else, $A_{ji} = \beta M_{ji}$
  - A is column stochastic
  - Intuition: Add more "importance" to pages in S (i.e., more related)



- Application: Usage of fixed set S sounds rigid, but efficient to handle lots of pages

PageRank Implementation

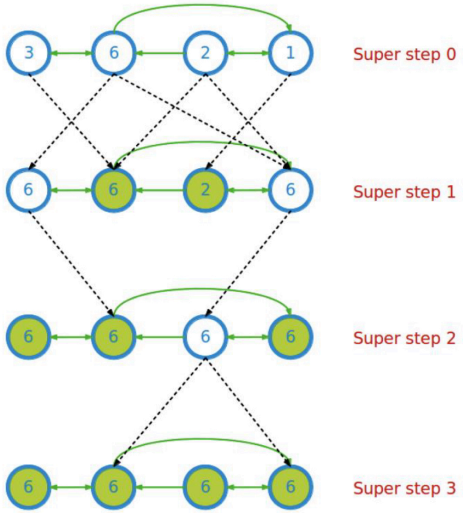
> Think like a vertex

- Idea: Graph algorithms implemented from **view of a single vertex**
  - Computation consists many supersteps
  - In each superstep, framework invokes user-defined function `compute()` for each vertex
  - Given `compute()` at vertex v and superstep s:
    - Can read messages sent to v in superstep $s-1$
    - Can send messages to other vertices that will read in superstep $s+1$
    - Can read/write value of v and value of its outgoing edges
  - Termination:
    - Vertex can choose to deactivate itself (e.g., when value does not change)
    - Inactive vertex can be "woken up" if new messages received
    - Computation terminates when all vertices are inactive

Example: Computing Max Value

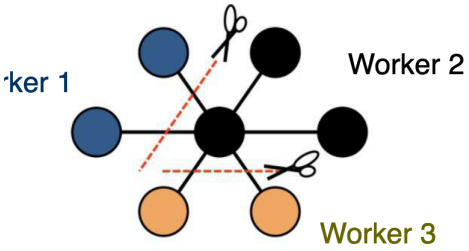
- Goal: Compute maximum of vertex values in graph
- Idea:
  1. Each vertex sends current value to neighbors
  2. Each vertex updates its value by comparing incoming values with its own value
  3. Process continues until all vertices stop changing

```
compute(v, messages):
    changed = False
    for m in messages:
        if v.getValue() < m:
            v.setValue(m)
            changed = True

    if changed:
        for each outNeighbor w:
            sendMessage(w, v.getValue())
    else:
        voteToHalt()
```



Pregel

> Closed-source graph processing model developed by Google



- Vertices are partitioned via hash function and "edge cut", and assigned to workers (i.e., stored in multiple servers)
- Each worker maintains state of its portion of graph **in memory**
  - Includes: Assigned vertices and relevant edges
  - Cut edges stored in both involved workers
- To send messages to vertices on different workers: Buffer locally and send as a batch to reduce network traffic
- Fault tolerance:
  - Checkpoint regularly to persistent storage
  - Detect failure through heartbeats

PageRank in Pregel

- Each superstep aggregates messages from neighbors to compute PageRank update
  - Use $r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{n}$
- Send $\frac{r_j}{d_j}$ as outgoing messages, effectively obtaining $\frac{r_i}{d_i}$ for the next superstep
- Stop after 30 supersteps for efficiency

```
compute(v, messages):
    if superstep() == 0:
        v.setValue(1 / numOfVertices())

    if superstep() >= 1:
        sum = 0
        for m in messages:
            sum += m
        v.setValue(0.8 * sum + 0.2 / numOfVertices())

    if superstep() < 30:
        d = numberOfOutNeighbors()
        for each outNeighbor w:
            sendMessage(w, v.getValue() / d)
    else:
        voteToHalt()
```