**Index-based triple stores** are a type of database management system that use indexes to store RDF triples, which consist of subject, predicate, & object. These systems use six indexes, which are all permutations of S, P, & O, allowing for fast single index scans. One of the main advantages of index-based triple stores is their high compression rates, which allow for efficient storage & retrieval of large amounts of RDF data. However, these systems can have complex query execution engines, making it difficult to optimize queries & achieve high performance.

**RAID**: **Redundant Array of Independent Disks**

• **RAID 0 (Striping):** Data is striped across multiple disks, which improves read & write performance. RAID 0 does not provide any redundancy or fault tolerance

**• RAID 1 (Mirroring):** Data is duplicated across two or more disks, providing fault tolerance through redundancy. also improves read performance but has a higher cost due to the need for additional storage.

• **RAID 5 (Striping with Parity):** Data & parity information are striped across three or more disks. Parity is an error detection & correction techniques. RAID 5 provides fault tolerance & improves read performance but requires extra storage for parity data. A min of 3 disks, & can handle 1 disk failure.

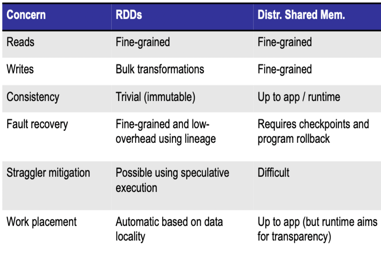
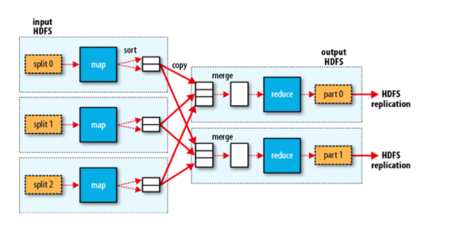
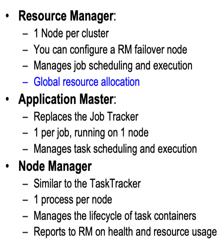
**Parallel File Systems (PFS)** are designed to address the needs of (HPC) environments, where large amounts of data need to be processed quickly & efficiently. PFS employ a distributed architecture in which multiple servers work together to store, manage, & access data. By distributing the workload across multiple servers, PFS can enhance performance, making it possible to handle massive data sets & complex computational tasks.One of the key features of PFS is **data chunking or striping**, in which data is divided into smaller chunks or stripes & distributed across multiple storage devices. This approach enables concurrent reading & writing of file data, which can significantly improve performance compared to traditional file systems. In addition, file data can be stored on multiple servers, further distributing the workload & improving performance. **Metadata management** is another critical feature of PFS & GFS. Metadata is stored separately from the data itself, allowing for more efficient searching, indexing, & data organization. This can be especially important in HPC environments, where data sets can be extremely large & complex.**Scalability** is another key feature of PFS. Because they are designed to handle large-scale storage & computing environments, PFS can be easily expanded to accommodate growing data sets & increasing performance requirements.To ensure **fault tolerance & data redundancy**, PFS use built-in mechanisms such as replication, erasure coding, & RAID techniques. These approaches help protect against data loss & hardware failures, ensuring that critical data remains available even in the event of a failure.Finally, **Load balancing** is an essential feature of PFS, which ensures optimal distribution of data & workload across available resources. By minimizing bottlenecks & maximizing performance, it also helps to ensure that HPC env can operate at peak efficiency.

**HDFS- NameNode** manages the file system namespace & coordinates file operations by directing clients to DataNodes for reads & writes. It periodically communicates with the DataNodes to maintain overall health & execute file system namespace operations like opening, closing, & renaming files & directories.

**The DataNode** provides actual storage & management of data blocks on a single host & performs block creation, deletion, & replication as instructed by the NameNode. It also sends heartbeat & block report messages to the NameNode.

**Secondary NameNode** not a failover NameNode but plays a role in checkpointing. It periodically merges the fsimage & edits the NameNode to creat new checkpoint image.

HDFS **Replication** to maximize reliability, availability, & bandwidth. Replicas are spread across machines & racks, & the master determines replica placement based on a set of rules (Same rack & different). HDFS also has a **rebalancer** that can be used to redistribute blocks across hosts.Threshold (X% set as target for balance state) HDFS allows the implementation of **custom block placement policies**, such as using lower utilization servers first or moving more blocks to newer generation hardware.

Hadoop YARN — 
Yet Another Resource Neqotiator 
ag 
Container 
MapReduce Status 
Job Submission 
Node Status 
Resource Request 
Cc..tain* HDFS Architecture 
Secondary Name Node 
NameNode 
/user/sales.csv 
C2 
Client 
Application 
Heartbe 
DN2 
DNI 
Manages the 
file system 
metadata 
DN3 
DN3 
vo 
DataNode 1 
DataNode 2 
ion, Balancing 
DataN0de 3 
Store the data 
36 Write Data Flow 
2. 
3. 
4. 
5. 
6. 
7. 
Lease acquisition 
• Chunk Version Increment 
Primary selection 
Data Forwarding 
Data buffering and Ack 
Write Command 
• Primary Mutation 
Secondary Mutation 
Primary Ack 
Client Ack 
4 
Client 
3 
Secondary 
Replica A 
Master 
2 
6 
7 
Primary 
5 
Replica 
Legend: 
Control 
6 
Secondary 
Replica B 
28 **MapReduce Implementation**- basic idea is to divide the input data into smaller chunks (splits), process each chunk independently in parallel across many nodes, & then combine the results.The **Map** phase is responsible for transforming the input data into key-value pairs. Each map task processes a single split of the input data & generates intermediate key-value pairs as output. The **shuffle** phase sorts & groups these intermediate pairs by key, & sends all values associated with a given key to a single reducer node. The **Reduce** phase takes the sorted & grouped intermediate data from the shuffle phase & performs some aggregation or computation on it. Each reduce task is responsible for processing all values associated with a particular key, & produces one or more output key-value pairs. The **overall framework handles** task scheduling, data distribution, synchronization, error handling, & fault tolerance. HDFS can store I/O data & intermediate data during the shuffle phase.

**Combiners (k,v) -> (k,v)** are local reducers that run in mem on the output of the map phase, before the data is sent over the network to the reducers. They can help reduce network traffic & improve performance by reducing the amount of data that needs to be transferred between nodes.(Pre reduce/pre computation like sum/average)

**Erasure coding** is used to mitigate hardware failures by storing codes (rather than full copies) that can be used to recover lost blocks of data in case of failure. Compared to replication, erasure coding has less storage overhead & can be a prime choice to balance cost, fault tolerance, & availability, especially when combined with more reliable hardware. **Bitwise XOR** is a type of erasure coding technique. It has, two blocks A & B are encoded using the XOR operation, resulting in a third block A⊕B, which is stored in a different location. If either block A or block B is lost due to hardware failure, the missing block can be recovered by taking the XOR of the remaining block & the encoded block.**Reed-Solomon** allows for recovery of any one block from any N other blocks, & can withstand up to K concurrent failures, making it applicable for any values of N & K. Reed-Solomon uses Galois Field arithmetic with GF(2W), where W is a positive integer, to encode & decode data. The notation RS(N,K) is used to represent the technique, where N is the number of data blocks & K is the number of coding blocks. Eg, (6,3) represents 6 data blocks & 3 coding blocks.

**RDD lineage** is the sequence of opp or transformations that were applied to create an RDD (Resilient Distributed Dataset) in a Spark app. It is essentially a record of the parent RDDs, & the transformation functions that were used to generate the current RDD. It is the history of how an RDD was created from its initial input data. This info is important for Spark's fault tolerance mechanism, it allows lost partitions or to be recomputed by re-executing the transfor which led the creation of the lost RDD.

**Benefits of RDD Model -** Consistency is easy due to immutability **-** Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data => [caching]) **-** Locality-aware scheduling of tasks on partitions**. –** Applicable for variety of applicatio

**Spark Lineage** - **narrow dependencies,** where each partition of the parent RDD is used by at most one partition of the child RDD

**- wide dependencies**, where multiple child partitions may depend on it.

**DAG Scheduling** 1. Spark creates an operator graph on the user code (RDD lineage)

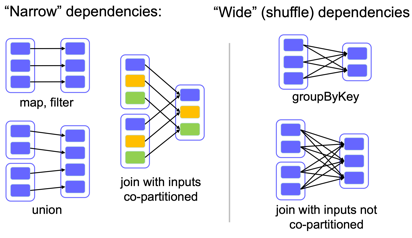
2. When an Action, the operator graph is submitted to the DAG Scheduler.

3. The DAG Scheduler breaks the lineage into stages based on the presence of wide dependencies.– Each stage consists of a set of tasks that can be executed together on the same set of input data. Spark optimizes the execution plan to minimize the number of shuffle operations required.4. The stages are then passed on to the Task Scheduler, which launches tasks through the cluster manager.5. The workers execute the tasks on the worker node. Spark coordinates the execution of tasks across the executors to ensure fault-tolerance & efficient resource utilization

• **HiveQL (HQL)** provides the basic SQL-like operations

– Column projection using SELECT – Filter rows using WHERE – JOIN between tables – Aggregate using GROUP BY – Manage tables & queries with CREATE,DROP& ALTER• Missing large parts of full SQL specification:

– HAVING clause in SELECT – Correlated sub-queries – Sub-queries outside FROM clauses– Updatable or materialized views– Stored procedures– Because it is only a wrapper for map-reduce execution, complex queries can be hard to optimize

Misc. Features 
Hadoop V2 
Hadoop V3 
YARN 
Timeline Service 
Balancing 
Erasure Coding 
Name Node 
HA (High Availability) 
YARN introduced in Had-OOP v2 has YARN Timeline service has been enhanced With ATS 
scalability issues. 
Nodes might have multiple disks added and 
replaced over time. HDFS Balancer In v2 
caused skew within a DataNode. 
Factor Of 3 data leading to 
200% storage overhead. If a file has 6 data blocks 
then a total of 18 Nocks win occupy the 
Hadoop v2 can support an additional passive 
Namenode as standby. 
v2 Which improves the scalability and reliability. 
Intra DataNOde Balancing has been introduced in 
Hadcnp v3 to address the intra-DataNode skews 
Which When disks are added 
Storage overhead Hadoop v3 is to 
With support for Erasure Coding. If a file has 6 data 
only 9 data are required in 
Hadoop v3 supports 2 more standby NameNodes 
25 Spark Components 
Driver 
Your code 
= n. SparkCmtut 
Spark Context 
ROD "Graph" 
Task Scheduler 
Block tracker 
Shuffle tracker 
Cluster 
manager 
Spark worker 
Executor 
Storage 
(HDFS, Hbase, Cassandra etc) Apache Avro & Apache Orc are both data serialization systems. While **Avro** is primarily a container file format for compact, fast & binary data storage, it also supports remote procedure call (RPC) with code generation being optional. On the other hand, **Orc** (Optimized Row Columnar) is a **column-oriented storage** format designed to improve performance for read-heavy workloads & features better compression, analytics query performance, vectorized query execution, & predicate pushdown. In terms of schema definition, **Avro schemas** are defined using JSON & contain information about the structure of the data being stored/processed, which is stored within the data file, while Orc organizes data by columns for better compression & analytics query performance.

**Predicate pushdown** is a feature in database systems, including columnar storage formats like Apache Orc, that optimizes query performance by pushing filtering operations (predicates) down to the storage layer instead of applying them at the query processing layer. This involves analyzing the predicates in a query & determining which ones can be applied (by checking metadata) at the storage level before retrieving the data. By applying predicates directly at the storage level, unnecessary data can be filtered out early, resulting in faster query execution times.

One unique characteristic of **ORC files** is that the file footer & postscript are located at the **end of the file**. This is because ORC files are designed to be written & read in an append-only manner in HDFS. By placing the file footer & postscript at the end of the file, new data can be added to the file without having to rewrite the entire file. Thus, ORC files an efficient & scalable for storing & processing large volumes data.

**Apache Parquet** is an open-source, columnar storage file format that was designed for efficient data storage & retrieval. It offers several features such as support for complex nested data structures, highly efficient columnar compression, & flexible encoding schemes. Parquet files are composed of row groups, column chunks, & pages, with metadata that includes file schema & structure. Additionally, Parquet is splittable & extensible, as it supports file footer metadata for efficient reads & parallel processing, as well as automatic schema merging for schema evolution. All these characteristics make Parquet an ideal choice for analytics (OLAP) use cases.

DBMS vs DSMS

• Persistent relations • Transient streams (and persistent relations)

• One-time queries • Continuous queries

• Random access • Sequential access

• Access plan determined by • Unpredictable data characteristics & arrival query processor & physical DB design patterns

• Resource limited (memory, per-tuple computation) • Resource rich

• Reasonably complex, near real time, query processing • Extremely sophisticated, • Query Operator: One pass • Query Operator: Arbitrary

• Query Plan: Adaptive • Query Plan: Fixed

**Approximate Query Evaluation** (AQE) is a technique used in data processing to handle high data volumes & avoid unbounded storage & computation. AQE is useful for ad hoc queries that require approximate results & follows the principle of eventual consistency. sliding windows, synopsis, samples, & load-shedding.

**Spark Streaming** • Extends Spark to support stream processing (Block replication- 2 )

• Efficient & fault-tolerant stream processing syste • Scales to 100s of nodes & achieves sub-second latencies• Batch-like API for implementing complex algo

**DStream** – seq of RDDs representing a stream of data DStream containing RDDs for each time window – Data sources such as: Twitter, HDFS, Kafka, Flume, TCP socket

• **Transformations** – modify data from on DStream to another – Standard RDD operations – map, countByValue, reduce, join. – Stateful operations – window,

• **Output Operations** – send data to external entity – saveAsHadoopFiles – saves to HDFS– foreach – do anything with each batch of results

**Fault Tolerance** • Executor with a receiver crashes– Restart receiver on a new executor– Use replicated data blocks

• **RDD Checkpointing** – Stateful stream processing can lead to long RDD lineages – Long lineage = bad for fault-tolerance, too much re-computation– RDD checkpointing saves RDD data to the fault-tolerant storage to limit lineage & re-computation• **Driver crashes** – DStream checkpointing: save DAG periodically to storage (HDFS)– How about the lost data blocks? Use Write Ahead Logs (WAL) i.e., write the data blocks to hdfs & read from it upon restart.• If Kafka used, replay log

**Kafka**- A message broker (distributor) between the consumers & producers is functionally independent from the cons & prod but gives an API to interact.

It is a fast, scalable, durable, & fault-tolerant publish-subscribe messaging system

– It offers high throughput for both publishing & subscribing. – It supports multi-subscribers & automatically balances the consumers during failure.• Strongly consistent– 1 Leader replica– Wait for "all” the replicas to complete (commit) – Tolerate N-1 failures with N replicas- Tradeoff if cluster in same data center low delay

**Kafka Operations** • *Write*: Messages sent by a producer to a particular topic partition will be appended in the order they are sent. – That is, if a producer sends 2 records (M1 then M2): M1 will have a lower offset than M2 & appear earlier in the log.

• *Read*: A consumer instance sees records in the order they are stored in the log.

• Log Cleanup: – Deletion: This can be set based on time / based on the size of the log.

Scheduling Process 
SparkContext 
RDD Graph (DAG) 
User Code 
rddl .join(rdd2) 
.groupBy(...) 
TaskScheduler 
Tasks 
Cluster 
Task 
1. split graph 
into stages of 
tasks 
2. launch tasks 
via cluster 
manager 
Worker 
Threads 
Block 
manager 
3. execute 
tasks 
35 **Zookeeper** is a highly available (replication) service that is used for coordinating processes in distributed systems. It uses a collection of servers -> **Ensembles** where one server acts as the leader & others act as replicas. If leader fails, another replica takes over. The ensemble remains available as long as majority of servers are running.

Election 
Increment current term 
• Change to Candidate state 
Vote for self 
Send RequestVote RPCs to all other servers, retry until 
either: 
1. 
2. 
3. 
Receive votes from majority of servers: 
Become leader 
Send AppendEntries RPC heartbeats to all other servers 
Receive RPC from valid leader: 
Return to follower state 
No-one wins election (election timeout elapses, split vote): 
Increment term, start new election 
14 Ensembles should have an odd number of servers to ensure that a majority decision can be made. For example, if there are 3 servers, then 2 servers must be running for the ensemble to remain available. Similarly, 5 servers can support up to 2 server failures, & 7 servers can support up to 3 server failures. **Read** requests can be directed to any server (local replica), & they succeed if at least one server responds. **Write** requests, on the other hand, go to the elected leader only & require more than half of the servers to be running (quorum) to be successful. This ensures that all writes are consistet across the ensemble. **Design** • API is wait-free – No blocking primitives in ZooKeeper– Blocking can be implemented by a client – Nodeadlocks

• Guarantees– Client requests are processed in FIFO order – Writes are linearizable

Data cached on client side (ID of current leader). Optimized for read-mostly operation.

**znodes** are organised in a hierarchical namespace - znodes can be manipulated by clients through the ZooKeeper API - znodes are referred to by UNIX style filesys path

**Ephemeral flag** is used to indicate that a znode should be deleted automatically when the client session ends. This is useful in scenarios where you want to create a temp node that is only needed for the duration of a session, to represent a lock/ lease.

**Sequential flag** is used to ensure that each znode created under a parent has a unique name that is sortable based on creation time. When a sequential flag is specified, a monotonically increasing counter is appended to the end of the znode's path name. The counter value of a new znode created under a parent with the sequential flag is always larger than the value of existing children of the same parent.

**Zookeeper manages Kafka** brokers (keeps a list of them) it helps in performing leader election for partitions•Also sends notifications to Kafka in case of changes (new topic)

**Consensus**: Types of Issues • **Concurrency**: Handling simultaneous requests & ensuring that they are consistently ordered across all nodes.• **Network partitions**: Nodes may become unreachable due to network issues, • **Fault tolerance:** The system must tolerate a certain number of node failures while maintaining consensus. •**Byzantine faults**: Malicious or arbitrary node behavior can disrupt consensus.

**Election Properties • Safety:** allow at most 1 winner per term – Each server gives out only 1 vote per term– 2 different candidates can’t accumulate majorities in same term

• **Liveness**: some candidate must eventually win– Choose election timeouts randomly in [T, 2T]– One server usually times out & wins election before others wake up

**Security**: By ensuring that a log entry is only committed when it has been written to the logs of all future leaders, Raft guarantees that the state machines will always apply the same value for that log entry. This prevents conflicts & inconsistencies that could arise if different state machines applied different values for the same log entry. Additionally, by only allowing the leader to append entries to its own log, Raft ensures that there can be no conflicting updates from multiple leaders at the same time, further improving system safety & consistency.

**Neutralizing Old Leaders**• Deposed leader may not be dead:– Temporarily disconnected from network– Other servers elect a new leader– Old leader becomes reconnected, attempts to commit log entries• Terms used to detect stale leaders (and candidates)– Every RPC contains term of sender– If sender’s term is older, RPC is rejected, sender reverts to follower & updates its term– If receiver’s term is older, it reverts to follower, updates its term, then processes RPC normally • Election updates terms of majority of servers – Deposed server cannot commit new log entries

**Client Protocol -** exactly-once semantics as long as client doesn’t crash

• if leader crashes after executing command– Must not execute command twice

• Solution: client embeds a unique id in each command– Server includes id in log entry

– Before accepting command, leader checks its log for entry with that id

– If id found in log, ignore new command, return response from old command

**PageRank** Update Rule: Each page divides its current PageRank (“flow”) equally across its outgoing links & passes these equal shares to the pages it points to. Notice that the total PageRank (“flow”) in the network is unchanged.

**Bulk Synchronous Parallel (BSP)** model is a programming & computation framework used for parallel computing. The computation in BSP is divided into a sequence of supersteps, where processes run concurrently, execute the same code, & create messages for other processes during a superstep. Barrier synchronization ensures that all messages have been transmitted before the next superstep begins, & messages are delivered at the start of the next superstep. Additionally, restrictions on sending & receiving messages within a superstep ensure deadlock-free execution

Map Reduce **shortcomings in Graph:**– Passes the entire state of the graph from one stage to next – Needs to coordinate the steps of a chained MapReduce• Pregel– Keeps vertices & edges on the machine that performs computation

– Uses network transfers only for messages

**Optimizations** • **Vertex Caching**: To reduce the overhead of data movement, GraphX caches vertex properties that are frequently accessed in memory, enabling faster access in subsequent iterations.• **Graph Pruning**: GraphX can remove unnecessary vertices and edges that do not contribute to the computation, reducing the overall size of the graph and improving processing efficiency.• **Incremental Computation:** GraphX supports incremental computation, where only the changes to the graph structure or properties are processed, avoiding redundant computation.

-Y New Election Rule 
Pickina the Best Leader 
committed? 
unavailable during 
leader transition 
• During elections, choose candidate with log most likely to 
contain all committed entries 
— Candidates include log info in RequestVote RPCs 
(index & term of last log entry) 
— Voting server V denies vote if its log is "more complete": 
• (lastTermv > lastTermc) OR 
• (lastTermv == lastTermc) AND (lastlndexv > lastlndexc) 
Leader will have "most complete" log among electing majority 23 Property graph split into smaller graphs by spliiting across a Vertex cut. A Vertex table and routing table is constructed to map it to an Edge table. Mirror caches are created within in the edge table and any updates are made to the cache first before sending it to the edge.

A picture containing text, screenshot, font, number

Description automatically generatedA picture containing text, screenshot, font

Description automatically generated