Deep Science Publishing, 2026
https://doi.org/10.70593/978-93-7185-126-8_4

DeepScience
Open Access Books

Chapter 4: Distributed Data Processing and Storage Architectures

4.1. Introduction

The rapid proliferation of cheap storage and high-speed networks has rendered the construction of distributed data processing and storage systems not only feasible but also essential. Such systems are key not just for data sharing or fault-tolerance purposes but also to cope with the immense quantities of data produced and stored during the normal functioning of Internet services. Currently, few applications can afford to use a single machine for data processing, and for those that cannot, the problems of redundancy in processing are much less critical than the problems of latency and availability. For even fewer applications is a single machine remotely sensible for data storage, bandwidth and maintenance costs being measured in life's hours. Consequently, distributed data systems are evolving according to the principles of decoupling storage and compute decoupling being so obvious that any group developing and scaling its workloads naturally reaches for such an architecture. At some point in the development of a workload, data locality no longer can be taken for granted, and when it does and task scheduling becomes a consideration a new cluster-based batch-processing framework emerges.

Distributed data processing architectures for batch data and supporting the locate-data strategy are a public good, and many such adaptations have been made: the development focus in the research and engineering groups indicated earlier; various Hadoop-derived batch-processing frameworks extending the design principles of Google MapReduce for external use by the open-source community; other languages and libraries that ease use of many such systems (e.g., Apache Pig, Apache Hive); and Sweeping, which utterly eliminates storage management from the data-management problem class by doing so simply and efficiently for every compute cycle.

4.1.1. Research design

Distributed data processing and storage architectures are examined and compared from a computer science perspective to answer three research questions:

- What are the fundamental components of distributed data systems?

- What are the major architectures for distributed data storage?

- What are the major architectures for distributed data processing?

The analysis draws on the concepts of architectural separation and coupling versus decoupling. Data systems invariably separate compute and storage layers for resource and workload decoupling, but closely coupled compute and storage elements may also be present.

Distributed Data Processing and Storage Architectures are examined and compared from a computer science perspective to answer three research questions:

What are the fundamental components of distributed data systems?

What are the major architectures for distributed data storage?

What are the major architectures for distributed data processing?

The analysis draws on the concepts of architectural separation and coupling versus decoupling. Data systems invariably separate compute and storage layers for resource and workload decoupling, but closely coupled compute and storage elements may also be present.

Fig 4.1: Distributed data processing architecture overview

4.1.2. Scope and Objective

Distributed data processing and storage architectures enable commodity-scale applications that require vast quantities of data to be processed within reasonable time frames. The dominant operating model divides these functions into distinct data-storing and data-processing systems. Aggregation of data and processing resources creates large clusters with a large number of nodes working together. Key technology trends shaping the architecture of these systems include increased number of cores and memory per node, large global memory hierarchies, and commodity Ethernet networking.

The purpose of this section is to provide an overview of the architectural space of data processing and storage systems. It reviews a range of specialized architectures that are tailored to specific workloads such as distributed databases or batching processing frameworks. Many distributed data systems allow for optimization of their design for their specific workloads. The wide variety of architectures arises from the CAP theorem, which states that it is impossible to provide consistency, availability, and partition tolerance at the same time. There trade-offs derived from this theorem thus require system architects to specialize systems in the context of their target workloads.

4.2. Fundamentals of Distributed Data Systems

Data systems can be understood as a combination of different data models and storage paradigms. Data models dictate how data is structured, while the storage paradigm determines how data is stored and accessed. The file model is an approach for structures without a system-associated query language. Storage is arranged as a logical hierarchy but services the underlying files through a sequential access mechanism. The object model provides the same functionality as a collection of files, but often in a manner more convenient for services that cater to objects or Binary Large Objects (BLOBs). The object model has a flat namespace, which has implications for performance and for the ability to access objects in bulk. The consistency guarantees offered by storage systems also play a significant role in the architecture choice. The Stored Program Computer Janus approach often offers the strongest guarantees at the expense of severe latency penalties. CAP theorem states that data structure availability, data structure consistency, and data structure partition tolerance cannot all be guaranteed at the same time. For practical systems, the reliability characteristics of the underlying distributed storage service will also have to be taken into account.

Distributed data systems may also be characterized by the (de)coupling of storage from compute as an architectural principle. A separated architecture offers scalability benefits and greater fault isolation. However, for very large data sets, the overhead of moving the data towards the computations becomes substantial. Furthermore, in some applications,

high-speed access to the data can be obtained only through (partial) co-location and replication; for those cases, a coupled architecture is preferable and appropriate care should be taken to exploit it efficiently. Assuming that the storage service meets the reliability requirements of the data processing service, actual data processing systems must provide fault tolerance against hardware failures, which are common in cloud clusters. Such fault tolerance is often provided through the use of load balancing; thus, the underlying load balancing technique of the processing framework plays an essential role in data processing reliability.

Equation 1: Speedup (Parallel Processing Efficiency)

Measures how much faster a distributed system performs compared to a single-node system.

$$S = T1TpS = \frac{T_1}{T_p}S = TpT1$$

- SSS: Speedup
- T1T_1T1: Execution time on a single node
- TpT_pTp: Execution time on ppp distributed nodes

4.2.1. Data Models and Storage Paradigms

Data models and storage paradigms represent fundamental aspects of a distributed data system's design. Three primary models exist: a traditional file-based representation, an object storage approach, and a database model. In file-based systems, data is organized into files, and these files into directories or folders. Object storage extends the file model by exposing a flat namespace that contains unique identifiers for accessing data; data units are typically large and rarely traverse the network in their entirety. Database systems are used for structured data. Regardless of the specific data organization method, all systems must provide durable storage and distribution of data across a number of instances.

All types of storage system provide a consistency guarantee about the high-level above data organization; consistency levels can be grouped into three categories: strict consistency, eventual consistency, and no guarantee beyond what the application can implement. Storage and database systems may also provide more than one type of consistency, exposing a subset of guarantees at any time; in these cases, the availability of the service is usually determined by the semantic "data models" and "storage paradigms"

4.2.2. Compute and Storage Separation

Separation of compute and storage in distributed data systems offers distinct scaling benefits with regard to the two resources, allowing them to be scaled independently based on their load, especially with regard to storage. However, as with other forms of decoupling within data systems, separate scaling creates other complications that need to be addressed, and in some cases, close coupling may actually be more desirable. The core of concept of separation of compute and storage is that a distributed storage system provides a service to various clusters of compute nodes. Storage nodes are optimized to provide storage and retrieval of data in bulk, while the compute clusters may be either short-term clusters performing high-performance analytical tasks or long-running clusters carrying out batch or near-realtime processing. In such an architecture, memory usage across the storage services is minimized through central management, and tasks having high memory requirements can be off-loaded to the storage service.

The first important consideration in this context is the realization that much of the data flow in a distributed data system is in the form of data being written rather than read. As a result, the scaling of storage in terms of throughput must take precedence over its latency characteristics. Decoupling the two forces the storage layer into specialized node-management strategies, often increasing reliability and placing a premium on replication and redundancy mechanisms. Data reconstruction schemes become important, especially in large storage clusters where the sheer volume of data means that even minor data loss can have larger impacts on analysis, and lost data need not be reconstructed immediately—as long as the data is reconstructible within the next few days or weeks, the task of regeneration can be done in a low-priority nonspecialized compute cluster.

4.2.3. Consistency, Availability, and Partition Tolerance

Any distributed system that tolerates network partitions cannot guarantee availability and consistency simultaneously. This trade-off, known as the CAP theorem, can manifest in two ways. First, a system can reduce availability during a network partition: when a partition occurs between two regions, requests to actors in one of the regions may be rejected. Second, a system can be designed to return stale data when a partition occurs, even to normal requests: operators that do not tolerate delays can return responses that do not reflect the state of the actors involved. There are additional ways to achieve a weaker model than the one that is shown, but these are less common.

Achieving high availability is certainly desirable, especially for applications that must provide a service to end users. Therefore, many systems either reject requests during a partition or accept requests and tolerate stale data. However, business requirements may

dictate that neither is acceptable. In that case, the delta queues constructed for replication can be used to define a consistency protocol for local requests. If all the actors in the application are written with the consistency requirements in mind and if the underlying layer enforces those requirements, the result is an application-specific consistency model. In these cases, the CAP theorem may be satisfied for the application as a whole, even though it is violated in the underlying system.

4.3. Architectures for Distributed Storage

Architectures for Distributed Storage assess the data storage components of distributed data systems, focusing on object storage and flat namespaces, distributed file systems, and distributed databases. Object storage stores unstructured data in a flat namespace and provides a limited set of operations. Metadata management is critical to scalability; directory-based, metadata-server-based, and complete-table approaches trade off scalability, consistency, and access pattern support. Distributed file systems generalize the UNIX file system for large-scale data storage. The metadata server is a key performance bottleneck, and caching, hierarchy, and failover strategies mitigate the issue. Distributed databases mimic a centralized DBMS, ensure consistency during transactions, and often provide sharding to address scale concerns. A two-tier approach, NewSQL, sacrifices some traditional consistency guarantees for speed and scalability.

Object storage and flat namespaces form the backbone of cloud services: the massive amount of unstructured data generated by users and Internet services needs to be efficiently stored and accessed. Object storage systems divide data into fixed-sized units called objects. Each object is stored in a flat address space, and data can only be retrieved, inserted, and deleted. The restricted interface provides wide-area network accessibility and enables key-value-like style data access at a low cost. A fundamental challenge in object storage systems lies in the meta-data management. Traditional file systems keep meta-data in a tree structure (e.g., the UNIX file system), but this design has significant scalability limitations. Meta-data can also be kept in a centralized server (e.g., Google File System), which can provide consistent meta-data for the entire system. However, the performance of the centralized meta-data server becomes a bottleneck in systems with large-scale data. Furthermore, some object storage systems (e.g., Amazon S3) completely sacrifice consistency and provide full eventual consistency instead.

4.3.1. Object Storage and Flat Namespace

Object storage is founded on a flat namespace and original design goals that include massive scalability, capacity, and availability. New object storage systems add performance as a deployment consideration after ensuring the original properties. Object

storage systems can be understood as functionally similar to distributed file systems minus a conventional hierarchical namespace. Files are accessible by a unique object ID but may also be referenced via a hypermedia representation or key/value interface. The object ID serves as a pointer to data and metadata in a flat namespace, typically stored on different servers. The flat namespace concept is an important enabler for horizontal scalability because object notation routing tables can be exclusively maintained on metadata servers. All other components can be dedicated to data IO. Other design considerations that can be distinct from a distributed file system include placement of the object data and metadata parts, method of propagating metadata updates, data storage format (often in a "blobs in racks" style), use of background garbage collection, and the level of concern for application access performance.

The flat namespace degrades performance in a manner usually associated with the global primary path in conventional DBMS, namely that access costs to individual objects must be managed on the application level. Nonetheless, many applications are bandwidth-driven and the coarseness of use is complemented by a high level of concurrent access. Distributed object storage systems have therefore found a natural domain. Different scalability quality dimensions such as total IO volume against individual IO response time can be decoupled. Object storage systems have moved into the cloud-enabling space in addition to being deployed for large archives. The former case is driven by multi-tenancy support, resource-over-commitment, and very low cost per GByte per year. Indeed profit models based on these are at the core of successful players like AWS, Microsoft, and Google.

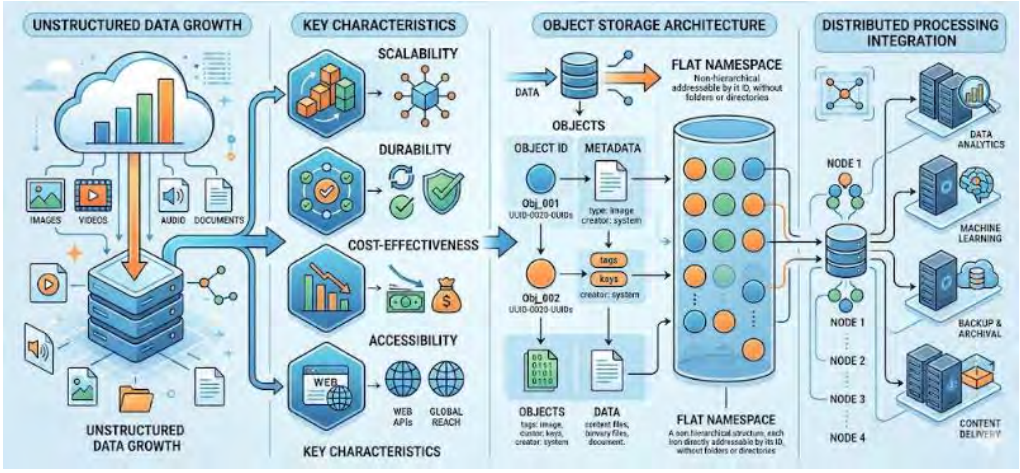

Fig 4.2: Object Storage and Flat Namespace

4.3.2. Distributed File Systems

Distributed file systems provide applications with a single file view, yet data are partitioned across nodes. The separation of file name space and data enhances scalability, but complicates metadata handling and introduces a single point of failure. A distributed metadata manager improves availability through redundancy and failover support. Metadata caching increases access speed, but requires consistency maintenance. Implementing a data-locality scheduling strategy accelerates reads. Explicitly storing data in local file systems can enhance performance, but increases complexity.

Distributed file systems share the object storage concept of a flat namespace, supplemented with a small, rapidly accessible metadata service. References to the name-to-locator mapping facilitate preliminary lookup, and the system accepts user-defined attributes for further information inclusion. Partitioning the data along with the workload minimizes data movement, and locality-aware task scheduling strives for co-location of data and processes. The exposed complexity permits these systems to adapt to varying requirements by tuning redundancy and cache policy.

Metadata management aims to prevent this single point of failure from becoming a bottleneck. Reducing the overhead requires a common workload characterization or a strategy for replicating the service. Caching serves as a complementary technique, since most reads are from recently accessed items, though it also introduces inconsistency. Resilience can be addressed using an active-passive design with seamless fail-over. However, enhancements usually follow an organic growth path rather than being intrinsically supported.

Caching is commonly advised for metadata-intensive operations, with focus on management rather than architecture. Performance can be improved by storing read-intensive data in local file systems or by adjusting the block size relative to the workload. Further, auxiliary disk I/O can be leveraged for reduced network congestion.

4.3.3. Distributed Databases and NewSQL Systems

Distributed databases provide a scalable storage solution for systems that require strong consistency without the costs associated with traditional relational databases. By offering the ACID guarantees that these provide while allowing storage resources to be elastically scaled, they enable a wider range of systems to provide an experience closer to that of classic RDBMS with the requisite throughput or latency. NewSQL systems achieve the scalability of NoSQL solutions while also allowing transactional properties more aligned with traditional databases, even though certain trade-offs exist—in terms of latency for sharded databases, for example.

Distributed databases can be implemented with a highly decoupled computation-storage architecture and sharded consistent key-value storage systems. In a sharded system, the distribution of records is determined by the key of the record, which is sent to a hash function and the result is taken mod the number of shards to determine to which shard it belongs. These systems can provide strong consistency guarantees, supporting the same kinds of applications as traditional RDBMS. However, latency is an issue as they cannot take advantage of the fact that a single copy of the shard is local. NewSQL systems configure a two-stage architecture in which a small number of servers can support the decoupled structure.

4.4. Architectures for Distributed Data Processing

The section compares batch processing frameworks and examines stream processing and real-time analytics, data locality, placement, and scheduling. It further reviews architectural considerations for in-memory computing and caching, focusing on data structures, eviction policies, and speed–price trade-offs.

Architecture design for large-scale distributed data processing encompasses three main paradigms: batch processing applied to archival data, near real-time processing of data streams, and low-latency response to user queries. A large volume of incoming data is usually stored in an underlying distributed storage system for later use. Dedicated processing clusters can execute analysis workloads in batch mode with high throughput. In batch processing frameworks, data randomly spread across many storage nodes is replicated to optimize access and fault tolerance. Scheduling approaches typically focus on throughput for best-prepared defined batch workloads.

Real-time analysis of data streams attempts to provide new insights as quickly as possible through near real-time analytics with low-latency response time—at the expense of throughput. For certain types of workloads with predefined response time, batch analysis of streaming data can never achieve the freshness of results made possible by event-driven stream processing solutions. In contrast, analytical queries often require ad hoc flexibility with variable latency. But latency requirements may be more relaxed than in traditional transaction-processing systems. Parameters such as data locality and task scheduling can significantly affect the achieved latency of an analytics workload. Data-management systems designed to support these considerations are normally organized as distributed shared memory for fast access.

A large amount of frequently accessed data can also be cached in fast-access storage, such as DRAM, to speed up query response time. Caching systems often exploit redundancy in the data access patterns of workloads and employ highly optimized cache-eviction policies to further improve performance. In certain applications such as

recommendation engines, customized in-memory data structures are built to accelerate specific types of queries. Specialized in-memory computing architectures combine the advantages of caching for certain workloads with high performance on queries that do not benefit from caching, achieving better price-performance ratios.

Equation 2: Amdahl's Law (Limits of Parallelization)

Determines the theoretical maximum speedup considering the non-parallelizable portion.

$$S(p) = 1(1-\alpha) + \alpha p S(p) = \frac{1}{(1-\alpha) + \frac{\alpha}{p}} S(p) = (1-\alpha) + p\alpha 1$$

- α\alphaα: Fraction of workload that can be parallelized
- ppp: Number of processors/nodes

4.4.1. Batch Processing Frameworks

The principles underlying batch processing frameworks are compared to elucidate their similarities and differences, concentrating on the scheduling model, fault tolerance mechanisms, and throughput calculations. Batch processing systems operate on large volumes of data stored in distributed file systems or HDFS-compatible storage that can perform multiple read and write operations. Examples include Apache Hadoop, Apache Spark, Apache Flink, and Chlorine, which is an internal system at Google. Batch processing frameworks employ a divide-and-conquer strategy, splitting jobs into smaller chunks for individual computation on different clusters. Nevertheless, they differ in scheduling methods and fault tolerance mechanisms. Batch processing workload throughput can be expressed as

Tbat = Nfiles ÷ Tfile + Ntasks ÷ Nrunning * Trask,

where Nfiles is the number of input files, Tfile is the time to read and write a file, Ntasks is the total number of tasks, Nrunning is the number of concurrently running tasks, and Ttask is the total time of a task.

In the basic MapReduce model, jobs are divided into map and reduce functions preceded by an optional prepare phase and succeeded by an optional cleanup phase. The master node creates map and reduce tasks and endeavors to schedule map and reduce tasks on nodes with local input data, queuing them in two separate FIFO queues. Upon completion of the map phase, a merge task can save the map outputs on HDFS, enabling the starter node to start scheduling reduce tasks even before the map phase is concluded. Other frameworks based on the same principles apply different scheduling algorithms;

for instance, YARN enables a shared cluster scheduling approach and supports other types of workloads, such as real-time, interactive, and streaming analytics. More sophisticated process scheduling and monitoring are possible with systems like Apache Tez or Apache Flink, which likewise support distributed data streaming.

4.4.2. Stream Processing and Real-Time Analytics

Popular stream-processing frameworks are Apache Storm, Apache Samza, Apache Spark Streaming, Amazon Kinesis, and Apache Flink. They share the idea of a data pipeline whose input is an unbounded sequence of events and whose output has the same nature. A pipeline is typically structured as a directed graph, where the vertices correspond to stages that implement specific operations and edges denote data communication.

Much like batch-processing frameworks, stream-processing systems separate the orchestration logic from the main data flow. An external controller takes care of assigning processing tasks to worker nodes, monitoring their health, and restarting them as necessary. Task-level fault tolerance guarantees are the responsibility of the individual stages—that is, they must ensure that the global operation remains correct even if individual workers fail. In a naive implementation, ensuring this could incur a large performance overhead: if, for instance, one of the stages stores copies of its state in a remote persistent storage service, such as Amazon S3, the throughput of the entire pipeline would suffer. To minimize this overhead, some frameworks—Apache Storm, for example—combine the full redundancy of tuple replication to mask crash failures with a more relaxed reliability requirement in terms of result consistency.

Real-time analytics contrast with simple transformation operations. They typically process, summarize, or aggregate input data and maintain state between consecutive firing of the pipeline. Windowing capabilities and techniques for managing state are thus key features of these systems. Another vital consideration when implementing real-time analytics is latency: unlike traditional batch-processing applications, the user expects intermediate results to be made available quickly. This aspect surfaces primarily in state-maintaining stages, the longer execution time of which has a large impact on response times. Various techniques are employed to mitigate the overhead incurred by preserving state, ranging from keeping state in-memory to limiting the size of the preserved information; evade disk bottlenecks during state accesses; and bound the delay introduced by data processing.

4.4.3. In-Memory Computing and Caching

In-memory computing and caching represent the high-performance corner of the distributed data processing and storage space. By trading physical storage capacity for speed, these systems offer a maximum performance experience at a relative cost that is always decreasing. However, careful use is often required to ensure that the speed/price ratio remains appropriate for the chosen solution.

In-memory computing utilizes structures such as hash tables or key-value stores to provide super-fast reads and writes of small-size chunks of data. They often serve as building blocks for more complex functions that inevitably need to create indexes that do not fit in-memory or take part in executing joins. Because these structures are in RAM, the choice of eviction policy is crucial: whenever a new entry is added and the structure is full, one entry needs to go, and this choice dramatically affects the performance of the whole application. Popular policies include least frequently used, least recently used, and random. Caching is a superset of this concept that stores a larger but slower data structure. Speed is gained from leveraging temporal locality and recomputing values that are less frequently used. A less-used but important example is pre-computing frequently-needed joins; this takes a lot of data space, but when the same join needs to be calculated thousands of times, it is often a better option than joining the tables every time.

4.5. Data Locality, Placement, and Networking

The considerations for data locality, placement, and networking are imperative to any distributed data processing and storage architecture. Locations of data and compute influence latency and throughput characteristics for both data retrieval and processing. Answers for data replication, task scheduling, network topology, and bandwidth impact overall architecture performance and scalability.

Locality and scheduling of compute relative to data represent trade-offs with direct impact on performance and cost. Altering task locations can reduce or eliminate data movement for certain operations; the previously uncertain costs of this movement can be explicitly captured if the physical location of data is known at task scheduling time. Provided sufficient data locality, appropriate scheduling can eliminate data transfer between nodes. Nevertheless, scheduling decisions remain challenging in distributed systems: even when exact task execution times are known, the centralized computation and communication overhead implicitly affect overall time; it often becomes critical to balance efficiency next to both task and communications time. Algorithms for scheduling thus usually incorporate a workload model to assist in cost predictions.

Design decisions regarding replication, costing, and task scheduling directly affect communication requirements, yet these are further shaped by the underlying network architecture. The routing and bandwidth characteristics from the latter determine the total communication time of the job and throughout the life of the system. Bandwidth, latency, and congestion also interact with placement and replication choices to produce additional scalability and performance trade-offs.

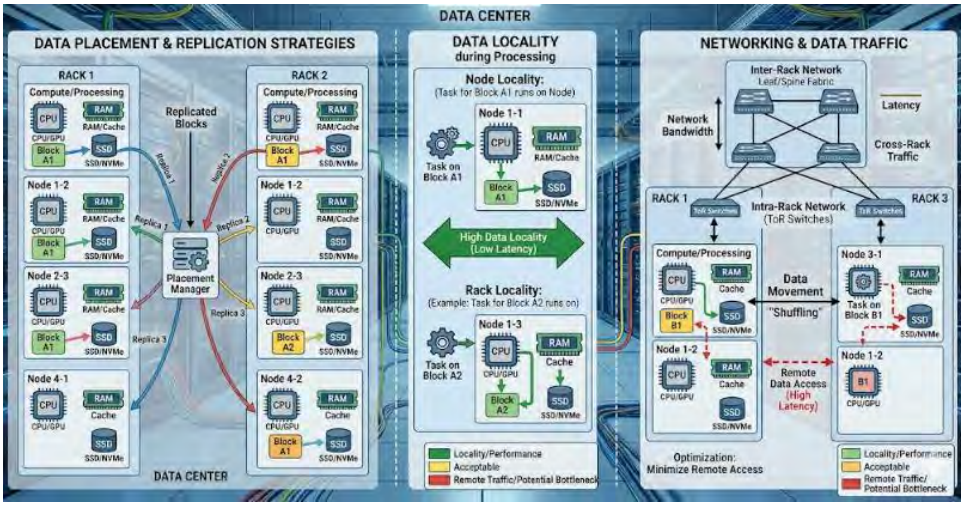

Fig 4.3: Data Locality, Placement, and Networking

4.5.1. Data Replication Strategies

The trade-offs between data availability and read and write latencies can be addressed by replicating data across multiple servers. Replicated data can be updated with a consistency model that balances these concerns—for example, allowing clients to read from the nearest copy even when it is temporarily out of date—and enables fault-tolerant access when a server, rack, or datacenter is unavailable. Replication can occur at the storage layer or the application layer, with the latter usually involving two-phase commits for strong consistency. As access patterns typically favor reads, it is important that time spent accessing the metadata that indicates where data copies live is minimized.

Storage layer replication is apparent in distributed databases, where a shard is replicated across multiple nodes. For NewSQL systems, data is typically replicated synchronously or quasi-synchronously across datacenter regions to support resilience against catastrophic failures. In Cadence, clusters are divided into leader-followers, with long-distance followers catching up during quiet time. In Dynamo-style systems, storage nodes have a local quorum for reads and a separate quorum for writes, leveraging eventual consistency to enable low-latency access to stale data while abstracting it away

from clients that require the freshest result. In HDFS, replication is used primarily for fault tolerance, with clients always reading from the copy on the chosen datanode. In Ceph, the replicated object store, clients bear the burden of selecting the best copy for read access.

4.5.2. Data Locality and Task Scheduling

Data locality improves performance by reducing the cost of data access. It can be achieved by executing tasks near the data or by co-locating data and compute. In the following discussion, data locality is enabled by local task scheduling. A comprehensive survey of the algorithms and approaches is presented in .

Data locality can be exploited in different ways, depending on the system architecture and data storage approach used. File systems and object storage systems store files or objects in a flat namespace; therefore, file placement alone does not affect data access times. Scheduling such tasks needs to consider the load on each worker. Most distributed databases support complex queries and allow the execution of subqueries, which can be evaluated locally, thus potentially reducing data access delays. Stream processing systems provide operators with data locality hints, while some stream databases have a push-based architecture, allowing local processing.

MapReduce is designed to execute map tasks near the data in distributed file systems. Local task scheduling tries to assign map tasks to the worker that is storing the data split, to workers in the same rack as the storing worker, or, if those are not available, to workers in the same data center. Local task scheduling is thus both bandwidth- and latency-aware. Besides map tasks, reduce tasks are also scheduled with an awareness of data locality; the entire Shuffle phase can be accelerated if the right assignments are made.

4.5.3. Network Topologies and Throughput Considerations

Network topologies and throughput capacity are important architectural facets of distributed data systems because they influence the duration and speed of the data flow across the network. The bipartite nature of data processing and storage systems entails that all tasks must exchange data amongst various servers, hence, throughput—bandwidth, latency, and congestion—is a key determinant of their efficiency. Due to the trade-off between cost and speed, designing a network that achieves effective throughput without straining the budget is a nontrivial task.

The total throughput capacity of a network can be affected by its topology. For example, a star topology has a higher capacity than a ring—multiple devices may launch data flows in a star topology, while in a ring they must take turns. High-throughput networks,

therefore, tend to have a star-like structure or more complex hierarchical structures that incorporate a star structure at their core. For instance, the organization of the Internet can be represented by a four-layer hierarchical star topology, where most routers are at the third layer directly connected to the fourth-layer routers. By aggregating lower-layer access speed and concentrating outgoing data traffic, a hierarchical topology can support high bandwidth. However, it incurs longer delays than a flat topology, primarily because each data flow must be forwarded by multiple routers before reaching its destination.

4.6. Conclusion

Distributed data processing and storage architectures support operations on vast amounts of data generated every day and enable new types of applications. Such processing operations can be categorized as either data analytics or transaction processing, while the data architectures can be categorized as distributed storage systems, supporting direct data access, or distributed databases, supporting concurrent queries involving multiple data records.

The analysis of these architectures offers the following insights. First, batch processing requires the management of very large data sets while maximizing throughput in a cost-effective manner. Distributed batch processing frameworks, such as Hadoop, combine a fault-tolerant, evolutionary approach with explicit data locality. Second, stream processing introduces additional requirements for low latency and state management. Real-time analytics frameworks combine these requirements with the use of complex queries. Third, distributed file systems support the efficient access of large-scale data sets by considering the specific requirements of the underlying storage systems. Fourth, when the working set fits into memory, in-memory computing offers a significant speed advantage by storing data in RAM rather than in secondary storage. Fifth, caching relies on cheaper components in the infrastructure to store frequently accessed data close to the application.Another trend gaining prominence is DataOps: the practice of improving the cycle time of moving data from production to analytical usage for training or retraining machine-learning models. Kappa and Lambda Architectures can be viewed in the context of DataOps. DataOps can benefit from work done in these areas, while also positioning them within a broader analytic context and involving new disciplines such as Compliance (ensuring the correctness of produced data) and Test Data Management (ensuring that proper test data is available to properly validate downstream models). DataOps can also encompass the flow and quality of data used within the enterprise to make day-to-day operational decisions.

Equation 3: Storage Replication Overhead

Represents total storage required when data is replicated across nodes for fault tolerance.

$$Stotal = R \times SdataS_{\text{total}} = R \times S_{\text{data}}Stotal = R \times Sdata$$

- StotalS_{\text{total}}Stotal: Total storage used
- RRR: Replication factor (e.g., 3 in systems like Hadoop Distributed File System)
- SdataS_{\text{data}}Sdata: Original dataset size

4.6.1. Future Trends

Distributed data-processing systems are in continuous evolution, driven by the generation of ever-increasing volumes of data and new technologies, such as the Internet of Things and machine learning. Cloud computing and serverless architectures have gained traction in recent years due to their attractive economics and elasticity. In parallel, the need for high-quality real-time streaming data has introduced architectures capable of handling continuous data feeds, which are increasingly required for machine learning or other data-driven systems. These trends are often encapsulated in the Marketing and Analytics tags Kappa Architecture and Lambda Architecture..

References

Zaharia, M., et al. (2023). Apache Spark: Unified analytics engine. *Communications of the ACM*.

Avinash Pamisetty, Vijaya Rama Raju Gottimukkala. (2024). Agentic AI-Driven Multi-Cloud Big Data Architecture For Predictive Demand, Credit Risk, And Inventory Financing In National Food Service Supply Chains. Metallurgical and Materials Engineering, 30(4), 959–975. Retrieved from https://metall-mater-eng.com/index.php/home/article/view/1933

Kreps, J., Narkhede, N., & Rao, J. (2023). Kafka: A distributed streaming platform. *LinkedIn Engineering*.

Aitha, A. R. (2021). Optimizing Data Warehousing for Large Scale Policy Management Using Advanced ETL Frameworks.

Armbrust, M., et al. (2023). Delta Lake: Reliable data lakes. *VLDB Journal*.

Karau, H., & Warren, R. (2023). *High performance Spark*. O'Reilly.

Thutari, R. T., Garapati, R. S., B M, Manjula., R K, Supriya., & M, Senbagan. (2025). Adaptive Access Control and Authentication Management for IoT Using Attention-GRU and Reinforcement Learning. In 2025 2nd International Conference on Software, Systems and Information Technology (SSITCON) (pp. 1–6). https://doi.org/10.1109/ssitcon66133.2025.11342003.

Vassiliadis, P. (2023). A survey of ETL and data pipeline systems. *ACM Computing Surveys*.

Nagabhyru, K. C., & Babu, A. J. Human In The Loop Generative AI: Redefining Collaborative Data Engineering For High Stakes Industries.

Vartak, M., Madden, S., Parameswaran, A., & Polyzotis, N. (2018). Model management systems: The next frontier of machine learning. IEEE Data Engineering Bulletin, 41(2), 13–26.

Amistapuram, K. (2024). Generative AI in Insurance: Automating Claims Documentation and Customer Communication. Turkish Journal of Computer and Mathematics Education (TURCOMAT), 15(3), 461-475.

Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., et al. (2018). Accelerating the machine learning lifecycle with MLflow. IEEE Data Engineering Bulletin, 41(4), 39–45.

Sudhakar, A. V. V., Inala, R., Verma, A. K., Nag, K., Pandey, V., & Anand, P. S. (2025). Hybrid Rule-Based and Machine Learning Framework for Embedding Anti-Discrimination Law in Automated Decision Systems. In 2025 International Conference on Intelligent Communication Networks and Computational Techniques (ICICNCT) (pp. 1–6). https://doi.org/10.1109/icicnct66124.2025.11232861.

Karau, H., & Warren, R. (2017). High performance Spark: Best practices for scaling and optimizing Apache Spark. O'Reilly Media.

Nandan, B. P., & Chitta, S. S. (2023). Machine Learning Driven Metrology and Defect Detection in Extreme Ultraviolet (EUV) Lithography: A Paradigm Shift in Semiconductor Manufacturing. Educational Administration: Theory and Practice, 29(4), 4555-4568.

Januschowski, T., Bohlke-Schneider, M., Wang, S., Salinas, D., et al. (2021). Criteria for classifying forecasting methods. International Journal of Forecasting, 36(1), 167–177.

Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., & Bouchachia, A. (2014). A survey on concept drift adaptation. ACM Computing Surveys, 46(4), Article 44.

Lu, J., Liu, A., Dong, F., Gu, F., et al. (2018). Learning under concept drift: A review. IEEE Transactions on Knowledge and Data Engineering, 31(12), 2346–2363.

Webb, G. I., Hyde, R., Cao, H., Nguyen, H. L., & Petitjean, F. (2016). Characterizing concept drift. Data Mining and Knowledge Discovery, 30(4), 964–994.

Nagubandi, A. R. (2025). Cryptocurrency Market Spillovers: Risk Contagion Across Global Financial Systems.

Baier, L., Kühl, N., & Satzger, G. (2020). How to cope with change? Preserving validity of predictive services over time. In Proceedings of the 53rd Hawaii International Conference on System Sciences (pp. 1381–1390).

Yandamuri, U. S. (2026). Scalable Cloud-Based Intelligent Decision Systems Leveraging AI and Big Data for Industry-Specific Optimization. Minnesota Journal of Business Law and Entrepreneurship, (1), 584-601.

Rabanser, S., Günnemann, S., & Lipton, Z. (2019). Failing loudly: An empirical study of methods for detecting dataset shift. In Advances in Neural Information Processing Systems, 32, 1396–1408.

Velangani Divya Vardhan Kumar Bandi. (2024). Intelligent Data Platforms For Personalized Retail Analytics At Scale. Metallurgical and Materials Engineering, 30(4), 1011–1027. Retrieved from https://metall-mater-eng.com/index.php/home/article/view/1011-1027

Lipton, Z. C., Wang, Y.-X., & Smola, A. J. (2018). Detecting and correcting for label shift with black box predictors. In Proceedings of the 35th International Conference on Machine Learning (pp. 3122–3130).

Suresh, H., & Guttag, J. V. (2021). A framework for understanding unintended consequences of machine learning. arXiv preprint arXiv:1901.10002.

Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., et al. (2019). Model cards for model reporting. In Proceedings of the Conference on Fairness, Accountability, and Transparency (pp. 220–229).

Enterprise-Scale Gen AI Orchestration Using Small LMs and LLM Agents for Intelligent ITSM and HRSD Automation in Enterprise Ecosystems. (2025). MSW Management Journal, 35(2), 1889-1897.

Gebru, T., Morgenstern, J., Vecchione, B., Vaughan, J. W., et al. (2021). Datasheets for datasets. Communications of the ACM, 64(12), 86–92.

Davuluri, P. N. Integrating Artificial Intelligence into Event-Driven Financial Crime Compliance Platforms.