

Architectural Foundations for Migrating Legacy Warehouses to Hadoop-Based Hybrid Data Platforms: A Review

Sudha Donepudi
Independent Researcher, USA.

Abstract - Organizations that move analytical workloads from relational warehouses to Hadoop-based platforms must decide how data is stored, queried, replicated, and verified in the new environment. This review examines ten peer-reviewed publications that describe the building blocks of such platforms. These cover distributed computation and structured storage, SQL access to distributed data, columnar file organization, warehouse data pipelines, massively parallel query engines, and unified in-memory processing. The review is organized in five sections, each drawing on two sources. The sources show that SQL-like interfaces lower the barrier for analysts, that block replication and checkpointing protect stored data, and that columnar formats and native execution engines reduce scan cost. They also show that declarative optimizers let one engine serve many workloads. The synthesis offers practical guidance for planning migrations. It stresses matching file formats to access patterns, choosing engines by latency need, and preserving metadata and lineage so that source-to-target parity can be demonstrated.

Keywords - Data Migration, Hadoop Ecosystem, SQL-On-Hadoop, Columnar File Formats, Apache Spark.

1. Distributed Processing and Storage Foundations

1.1. The MapReduce Computation Model

Migration from relational warehouses to Hadoop-style platforms builds on two ideas that originated in large-scale web infrastructure: a simplified computation model and a scalable store for structured records. The computation model is MapReduce. A developer writes a map function that converts each input record into intermediate key-value pairs, and a reduced function that merges all values sharing a key. The runtime system partitions the input, schedules tasks across a cluster of commodity machines, manages communication among nodes, and handles machine failures, so programmers need little experience with distributed systems (1). The authors show that many data-intensive jobs fit this two-function pattern, including distributed grep, word counting, reverse web-link graphs, inverted indexes, and distributed sorting.

Several design choices explain the model's practicality. A single master process assigns maps and reduce tasks to workers and tracks their progress. Failed tasks are executed again, which preserves correct results when the functions are deterministic. Slow workers, known as stragglers, are mitigated by launching backup executions of the remaining tasks near the end of a job. The scheduler also places tasks on or near machines that already hold the relevant input blocks, which reduces network traffic.

1.2. Bigtable as a Structured Storage Model

Computation alone does not replace a warehouse; structured data also needs a scalable home. Bigtable is a distributed storage system for structured data, described as a sparse, distributed, persistent, multidimensional sorted map. Each value is addressed by a row key, a column key, and a timestamp, so several versions of a cell can coexist (2). Columns are grouped into column families, which serve as the unit for access control and storage accounting. Rows are kept in lexicographic order and divided into contiguous ranges called tablets, which a master assigns to tablet servers.

The design relies on supporting services. The underlying distributed file system stores logs and data files, a lock service maintains membership and coordination, and an immutable sorted file format holds the persisted data. The authors report that applications with very different demands used the system, from throughput-oriented batch processing to latency-sensitive serving of end-user requests.

1.3. Implications for Legacy Warehouse Transition

Taken together, the two sources describe a division of labor that a migration team must reproduce. Batch transformation is expressed through the map and reduced pattern, while keyed, versioned access is provided by a sorted-map store. Relational tables with fixed schemas and joints do not map directly to either model. Planners therefore need to decide, for each legacy workload, whether it belongs to scan-oriented batch computation or to row-key-driven access, and they must account for the access control and versioning behavior that column families and timestamps provide.

Table 1: Core Components of the Foundational Processing and Storage Models [1, 2].

Component	Role	Migration-Relevant Characteristic
Map function	Converts input records to intermediate pairs	Expresses record-level transformation logic
Reduce function	Merges values that share a key	Expresses aggregation logic
Runtime master	Schedules tasks and tracks progress	Hides failure handling from developers
Row key ordering	Sorts rows lexicographically	Determines locality of related records
Column family	Groups columns for access control	Shapes schema design and permissions
Tablet server	Serves a contiguous range of rows	Distributes load across the cluster

2. SQL Warehousing and Distributed Storage on Hadoop

2.1. Hive as a Declarative Warehousing Layer

Writing low-level Map Reduce programs for routine analysis proved laborious, which motivated a warehousing layer over Hadoop. Hive provides a SQL-like language, HiveQL that supports familiar constructs and allows developers to plug in custom map-reduce scripts when the declarative syntax is insufficient (3). Data is organized into tables with typed columns, including complex types such as lists, maps, and structures. Tables may be divided into partitions according to column values and further into buckets, so queries can skip data that is irrelevant to a predicate.

The architecture contains several cooperating parts. A metastore serves as the system catalog and records schemas, partitions, and storage locations. A compiler parses a query, produces a logical plan, and translates it into a directed acyclic graph of MapReduce jobs. An execution engine runs those jobs, and serializer-deserializer interfaces let the system read data in different record formats. Command-line, web, and Thrift interfaces provide access for users and external programs (4).

Table 2: Principal Hive Constructs and Their Purposes [3, 4].

Construct	Purpose	Practical Consequence
HiveQL	Declarative SQL-like querying	Lowers the barrier for analysts
Partition	Divides a table by column value	Enables data skipping
Bucket	Subdivides data within a partition	Supports sampling and efficient joins
Metastore	Stores schema and location information	Centralizes metadata
Query compiler	Builds MapReduce job graphs	Converts SQL into distributed work
Serializer-deserializer	Reads and writes record formats	Accommodates varied file layouts

2.2. HDFS Architecture and Reliability

The storage beneath Hive is the Hadoop Distributed File System. A single NameNode maintains the namespace tree and the mapping of file blocks to nodes, whereas many DataNodes hold block replicas in their local file systems. Files are split into large blocks, and each block is replicated, with three copies as the default. Placement takes network topology into account so that a rack failure does not remove every copy (3). DataNodes send periodic heartbeats and block reports, which let the NameNode detect failures and schedule new replicas.

Metadata durability relies on a namespace image and a journal of changes. A checkpoint node can merge these records periodically, a backup node can maintain an in-memory copy of the namespace, and snapshots allow state to be preserved before software upgrades. Additional utilities include a balancer for even disk utilization and block verification through checksums (4).

Table 3: Principal HDFS components and responsibilities [3, 4].

Component	Responsibility	Reliability Contribution
NameNode	Holds namespace and block map	Central coordination of metadata
DataNode	Stores block replicas	Distributed data holding
Replication policy	Places copies across racks	Tolerates node and rack failure
Checkpoint node	Merges image and journal	Protects metadata durability
Snapshot	Preserves file system state	Supports safe upgrades
Balancer	Redistributes blocks	Prevents uneven disk use

2.3. Combined Implications

Hive supplies the relational vocabulary that legacy warehouse users expect, and HDFS supplies the durable, replicated substrate. A migration design must therefore treat metastore contents and HDFS replication and checkpoint settings as part of the data, not as an afterthought.

3. Columnar Layouts and Warehouse Data Pipelines

3.1. A Production of Warehouse Pipeline

A large social-media company documented how its analytics infrastructure evolved from a conventional relational warehouse to a Hadoop-based design as data volume outgrew its earlier systems. Data originates from web-server logs and from operational databases. Log events are collected through a distributed logging service into intermediate Hadoop clusters, then copied into a Hive-managed cluster where scheduled and ad hoc jobs run. Dimension data from operational MySQL databases is loaded periodically. Results flow onward to relational serving systems, including Oracle and MySQL, where reporting tools consume them (5).

The authors describe operational challenges alongside the architecture. These include resource sharing among many users with different workloads, scheduling of periodic jobs, and the need to keep the warehouse usable by analysts who rely on SQL rather than programming. The pipeline therefore functions as a hybrid, with legacy relational systems retained at the edges while Hadoop performs bulk storage and transformation (6).

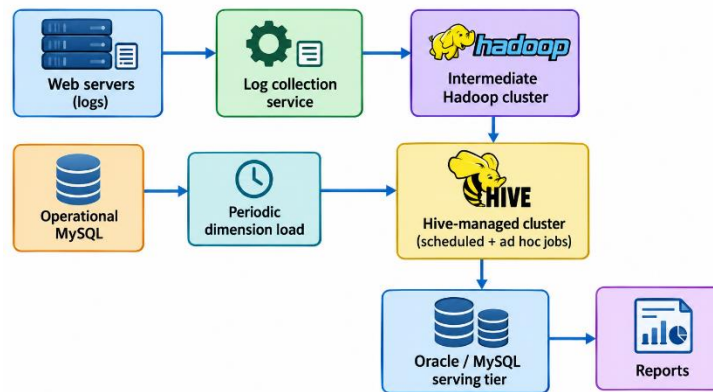


Fig 1: Hybrid Warehouse Data Flow [3, 4]

3.2. Columnar Storage and Tree-Based Execution

A second source addresses interactive analysis of very large datasets. Dremel stores nest records in a columnar layout, so a query reads only the columns it references. Repetition and definition levels attached to each value allow the original nested records to be reassembled without loss. Queries execute through a multi-level serving tree: the root receives a query, rewrites it, and passes pieces down to intermediate nodes and finally to leaf servers that scan data. Partial results are aggregated on the way back up [5]. Data is read where it lives, without a loading step, and the system complements MapReduce by analyzing its outputs.

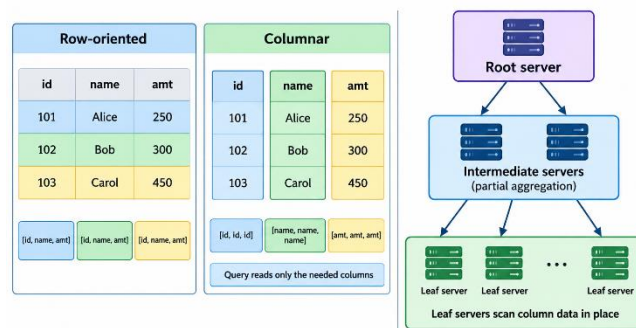


Fig 2: Columnar Organization and Serving-Tree Execution [5, 6].

3.3. Combined Implications

The pipeline description shows where legacy and modern stores meet, and the columnar design shows why analytic reads on the modern side are efficient. Migration planning should record every hand-off in the pipeline, because each is a point where source-to-target parity must be confirmed [6].

Table 4: Pipeline Stages and Storage Choices [5, 6].

Stage	Function	Design Feature
Log collection	Gathers event data	Distributed logging service

Intermediate cluster	Buffers incoming data	Decouples producers from the warehouse
Hive-managed cluster	Running scheduled and ad hoc jobs	SQL access over distributed storage
Relational serving tier	Delivers results to reports	Retains legacy consumers
Columnar layout	Limits read to needed columns	Repetition and definition levels
Serving tree	Parallelizes interactive queries	Aggregation across tree levels

4. Optimized File Formats and Parallel SQL Engines

4.1. Hive Advancements

Hive's early design produced inefficiencies that later work addressed. A new columnar format, ORC, organizes data into stripes, applies type-aware encodings and general-purpose compression, and keeps lightweight indexes that let readers skip unneeded row groups (Huai et al., 2014). The same work describes an optimizer that detects correlations among operations, so shared scans and repeated data shuffles are eliminated. It also introduces vectorized execution, which processes batches of rows at once, and adopts a runtime that avoids writing intermediate results to the distributed file system between stages [7].

Table 5: Hive Advancements and Their Purposes [7, 8].

Advancement	Purpose	Effect on Queries
ORC format	Columnar storage with indexes	Reduces data read
Type-aware encoding	Compact column representation	Lowers storage cost
Correlation optimizer	Merges related operations	Removes redundant scans
Vectorized execution	Processes row batches	Improves processor efficiency
Tez-style runtime	Avoids intermediate materialization	Shortens multi-stage plans

4.2. Impala as a Native Parallel Engine

Impala takes a different path by bypassing MapReduce entirely. It is an open-source, massively parallel SQL engine that runs on Hadoop clusters and is written primarily in C++ for the execution backend, with a Java frontend that performs query planning. A daemon runs on each node, and any daemon can coordinate a query. A state-store service distributes cluster membership information, and a catalog service propagates metadata, drawing on the Hive metastore [7]. Plans are divided into fragments that execute on many nodes, and runtime code generation tailors machine code to each query.

The engine reads several formats from HDFS, including Parquet, Avro, Sequence File, and text, and can also query HBase tables. Parquet is presented as the preferred columnar choice. The authors note that mid-query failure recovery is not provided, so a failed query must be resubmitted [8].

Table 6: Impala Components and Roles [7, 8].

Component	Role	Design Note
Daemon on each node	Plans and executes queries	Any node can coordinate
Statestore	Distributes membership information	Supports cluster awareness
Catalog service	Propagates metadata	Reuses the Hive metastore
Java frontend	Compiles and plans queries	Builds distributed fragments
C++ backend	Executes plan fragments	Uses runtime code generation
File format support	Reads columnar and row formats	Parquet is the preferred columnar format

4.3. Format and Engine Selection

The two sources frame file format and engine as linked decisions. ORC pairs with Hive, whereas Parquet pairs naturally with Impala, and both shorten scan time by reading fewer columns. Because Impala uses the shared metastore, tables defined once can be queried from either system, which reduces duplicated metadata work during migration.

5. Unified Processing with Spark and Spark SQL

5.1. Spark as a Unified Engine

Spark was designed to run diverse workloads in one engine rather than moving data among specialized systems. Its central abstraction is the resilient distributed dataset, an immutable, partitioned collection manipulated through transformations that are evaluated lazily. Fault recovery relies on lineage: each dataset records the operations that produced it, so lost partitions can be recomputed without replicating data. Datasets may also be caught in memory, which benefits iterative algorithms and interactive use [9]. Libraries for SQL, streaming, machine learning, and graph processing share this core, so applications can combine them in a single program [9].

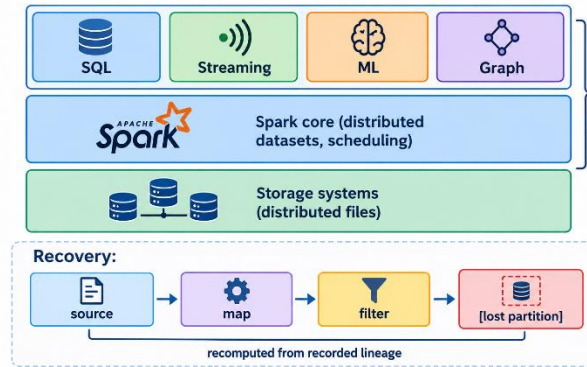


Fig 3: Unified Spark Stack and Lineage-Based Recovery [9, 10].

5.2. Spark SQL and Catalyst

Spark SQL joins relational and procedural processing. Its Data Frame API lets users apply relational operations to distributed collections, while allowing ordinary code to be mixed with declarative queries. Catalyst, the extensible optimizer, represents plans as trees and transforms them with rules. Optimization proceeds through analysis, logical optimization, physical planning, and code generation (Armbrust et al., 2015). A data sources interface supports pushing column selection and filters into external systems, and the engine can infer schemas from semi-structured data such as JSON. User-defined functions and types extend the system, and Hive tables can be accessed through the same interface [10].

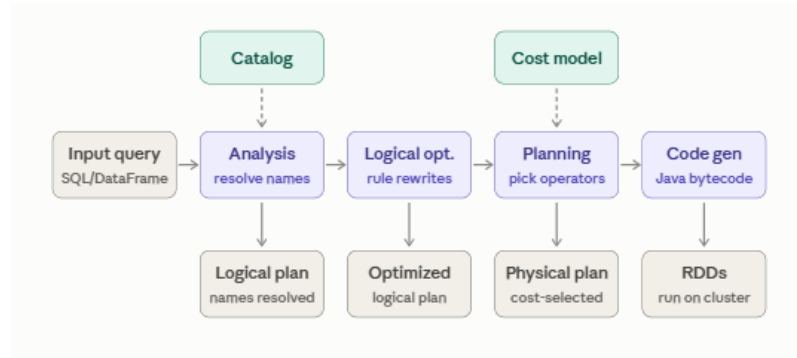


Fig 4: Catalyst optimization phases [9, 10].

5.3. Combined Implications

The two sources together suggest that one engine can serve migration validation, transformation, and downstream analytics. Declarative optimization means the same logic benefits from improvements without rewriting, and pushdown into external sources limits unnecessary data transfer [10].

Table 7: Spark components and migration relevance [9, 10].

Component	Function	Relevance
Resilient distributed dataset	Immutable partitioned collection	Fault-tolerant processing
Lineage	Records derivation steps	Recovery without replication
DataFrame API	Relational operations on collections	Familiar to SQL users
Catalyst optimizer	Rule-based plan transformation	Automatic query improvement
Data sources interface	Connects external stores	Filter and column pushdown
Library integration	Combines SQL, streaming, learning	One engine for many tasks

6. Conclusion

The ten sources describe a consistent architecture for moving analytical workloads from relational warehouses to distributed platforms. Foundational work established a simple computation model and a sorted, versioned store. Hive and the Hadoop file system then added familiar SQL access, durable replicated storage, and a shared metadata catalog. Pipeline and columnar-storage research showed how legacy and modern systems coexist, with relational serving tiers retained at the edges and bulk work done on distributed storage. Later engines improved performance through columnar formats, native parallel execution, and optimizers, and Spark unified batch, interactive, and library-based workloads under declarative optimization.

Several practical implications follow. First, file format and query engine should be chosen together, based on access patterns and latency needs. Second, metadata, partitioning, and replication settings should be documented and migrated alongside the data itself. Third, each hand-off between legacy and modern systems should be treated as a checkpoint where row-level and aggregate agreement is confirmed before consumer's switch. Fourth, a declarative interface such as SQL or DataFrames keeps transformation logic portable across engines, which limits rework when platforms change. Finally, teams should plan for the limits of individual engines, such as the absence of mid-query recovery in some systems, by designing retry and monitoring around them. Following these principles gives organizations a defensible path to platform modernization.

References

- [1] Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107–113.
- [2] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), Article 4.
- [3] Thusoo, A., Sen Sarma, J., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., & Murthy, R. (2009). Hive: A warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2), 1626–1629.
- [4] Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010). The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies* (pp. 1–10). IEEE.
- [5] Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R., & Liu, H. (2010). Data warehousing and analytics infrastructure at Facebook. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 1013–1020). ACM.
- [6] Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., & Vassilakis, T. (2010). Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1–2), 330–339.
- [7] Huai, Y., Chauhan, A., Gates, A., Hagleitner, G., Hanson, E. N., O'Malley, O., Pandey, J., Yuan, Y., Lee, R., & Zhang, X. (2014). Major technical advancements in Apache Hive. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 1235–1246). ACM.
- [8] Kornacker, M., Behm, A., Bittorf, V., Bobrovsky, T., Ching, C., Choi, A., Erickson, J., Grund, M., Hecht, D., Jacobs, M., Joshi, I., Kuff, L., Kumar, D., Leblang, A., Li, N., Pandis, I., Robinson, H., Rorke, D., Rus, S., ... Yoder, M. (2015). Impala: A modern, open-source SQL engine for Hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*.
- [9] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., & Zaharia, M. (2015). Spark SQL: Relational data processing in Spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 1383–1394). ACM.
- [10] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., & Stoica, I. (2016). Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11), 56–65.