EDUSCHOOL JOURNAL OF COMPUTER SCIENCE RESEARCH STUDIES (EJCSRS)
(Open Access, Double-Blind Peer Reviewed Journal)
ISSN Online:

Column-Oriented Database Internals: Vectorized Execution, Late Materialization, and Compression in Clickhouse and Duckdb

Win Mathew John

Associate professor, PG Department of Computer Applications, Marian College Kuttikkanam (Autonomous), Kerala, India

Article information

Received: 17th February 2026
Received in revised form: 23rd March 2026
Accepted: 25th April 2026
Available online: 12th June 2026

Volume: 1
Issue: 2
DOI: https://doi.org/10.5281/zenodo.20657522

Abstract

The database systems that power analytics look almost nothing inside like the ones that power transactions, even when they speak the same SQL. The difference begins with a single decision about how a table is laid out on storage. A transactional system stores each row contiguously, which suits writing and reading whole records; an analytical system stores each column contiguously, which suits scanning a few columns across millions of rows. That one choice cascades into a distinct set of internal techniques that together make modern analytical engines an order of magnitude faster than row stores on the queries they target. This paper examines three of those techniques as realised in two influential open-source systems, ClickHouse and DuckDB. Vectorized execution processes data in batches of thousands of values rather than one tuple at a time, amortising interpreter overhead and unlocking the data-parallel instructions of modern CPUs. Late materialization keeps data in compact column form for as long as possible, delaying the expensive reconstruction of rows until it is unavoidable. And because a column holds values of one type with much local redundancy, lightweight compression schemes shrink storage and, critically, let the engine operate directly on compressed data. We explain the mechanism behind each, show how they reinforce one another, and discuss why the column store has become the default architecture for analytical workloads.

Keywords:- Column-Oriented Database, OLAP, Vectorized Execution, Late Materialization, Lightweight Compression, ClickHouse, DuckDB, Query Processing, Analytical Databases.

I. INTRODUCTION

Two kinds of work pull a database in opposite directions. Transaction processing inserts, updates, and retrieves individual records, touching every column of a few rows. Analytics scans enormous numbers of rows but only a handful of columns, computing aggregates and grouping results [3]. A storage layout optimal for one is poor for the other, and for decades the same row-oriented engines tried to serve both, to the detriment of analytics.

The column store resolves the tension by inverting the layout. Instead of storing each row's fields together, it stores each column's values together, so that scanning one attribute across the whole table reads a single contiguous run and skips the columns a query does not mention entirely [1], [2]. Figure 1 contrasts the two layouts. This idea, proven at research scale by C-Store and at production scale by a generation of systems that followed, is now the consensus architecture for analytical databases [2], [3]. But the layout is only the beginning. Its real power comes from the execution techniques it enables, because once data is stored by column, the engine can process it in ways a row store cannot.

This paper focuses on three such techniques, using ClickHouse and DuckDB as concrete reference points. ClickHouse is a distributed column store built for high-throughput analytics over very large datasets; DuckDB is an embeddable analytical engine that brings the same internals into a single process for interactive analysis [8], [9]. Despite their different deployment models, both rest on vectorized execution, late materialization, and column compression. We treat each in turn and show how, combined, they account for the performance gap illustrated in Figure 2.

II. THE STORAGE LAYOUT AND WHY IT MATTERS

The case for column storage on analytical workloads rests on the memory hierarchy. A query that averages one column over a billion rows reads, in a column store, exactly that column's data and nothing else, so no bandwidth is wasted moving fields the query ignores [1], [3]. In a row store the same query drags every column of every row through the cache to reach the one it needs. Because analytical queries are typically bandwidth-bound, eliminating that waste is a large and direct win. The layout also concentrates values of a single type and similar magnitude next to one another, which, as Section V explains, is what makes aggressive compression possible [7]. The foundational comparison of column and row stores quantified these effects and established that the advantage comes not from one trick but from the layout enabling a whole family of them [1]. The same columnar principle underlies the open storage formats and in-memory standards of the modern analytical ecosystem, from the nested columnar representation pioneered by Dremel [13] to the cross-language Apache Arrow layout that lets engines share columnar data without serialisation [14].

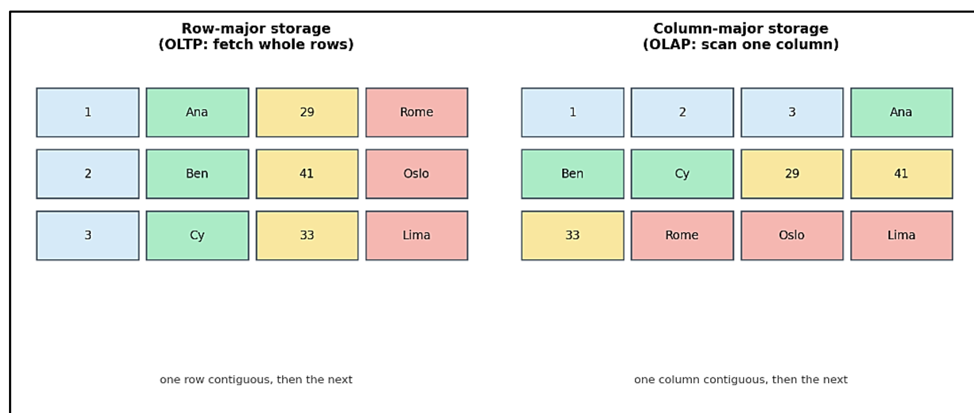

Fig 1: Row-major versus column-major storage of the same table. Scanning one attribute across all rows reads a contiguous run in the column layout and touches no irrelevant data.

III. VECTORIZED EXECUTION

A. Past the Tuple-at-a-Time Bottleneck

Classical query engines use an iterator model in which each operator produces one tuple at a time, pulling from its child on demand [16]. The model is elegant and composable, but at the scale of analytics it is ruinously slow, because every operator invocation carries a fixed overhead of function calls and interpretation that is paid once per row across billions of rows [4], [11]. Vectorized execution, pioneered in the MonetDB/X100 system, replaces the single tuple with a vector, a batch of typically a few thousand values from one column, as the unit that flows between operators [4], [5]. The per-batch overhead is now amortised over thousands of values, and the inner loop that does the real work becomes a tight pass over a contiguous array.

B. Friendly to the Hardware

Tight loops over typed arrays are exactly what modern CPUs reward. They keep the instruction pipeline full, make memory access predictable so the prefetcher succeeds, and let the compiler emit single-instruction multiple-data operations that apply one instruction to many values at once [5], [12]. This is why a column store's columnar layout and vectorized execution are natural partners: the data is already a contiguous typed array, which is the ideal input for a vectorized, SIMD-friendly kernel [3], [12]. An alternative school compiles each query to native code rather than interpreting vectors, and a careful study of the two approaches found they reach broadly comparable performance by different routes, with vectorization favouring simplicity and compilation favouring the very hottest paths [10], [11]. Both ClickHouse and DuckDB are built around vectorized execution [8], [9].

IV. LATE MATERIALIZATION

A query eventually has to produce rows, but the longer an engine can postpone assembling them, the faster it runs, and this is the insight behind late materialization [6]. In the naive, early-materialization strategy, the engine reconstructs full tuples from the relevant columns at the start of execution and then processes rows; in the late strategy, it carries the columns separately and operates on them in column form, stitching values into rows only at the latest possible moment, often after filtering and aggregation have already discarded most of the data [6]. The payoff is twofold. The engine does its work on compact, type-uniform column data that vectorizes well, and it never pays to reconstruct rows that a predicate later eliminates. Position lists and selection vectors track which rows survive each operator, so a filter can mark qualifying positions without materialising anything, and only the columns actually needed in the result are ever assembled [3], [6]. Late materialization is therefore not an isolated trick but the discipline that lets vectorized execution and compression keep paying off deep into a query plan.

V. COMPRESSION THAT EXECUTION CAN SEE THROUGH

A. Lightweight, Type-Aware Schemes

Because a column holds values of one type, often sorted or slowly changing, it compresses far better than a mixed row, and analytical engines exploit this with lightweight schemes chosen per column [7]. Run-length encoding collapses long stretches of a repeated value; dictionary encoding replaces a small set of distinct strings with compact integer codes; delta and frame-of-reference encodings store small differences instead of full values for sorted or clustered numeric columns; bit-packing uses only as many bits as a column's range requires [3], [7]. These are deliberately cheaper than general-purpose compressors, because the goal is not maximal ratio but the best balance of size against decompression speed.

B. Operating on Compressed Data

The decisive advantage is that some of these encodings can be processed without being decoded. A count or a filter over a run-length-encoded column can work on the runs directly; an equality predicate over a dictionary-encoded column can compare integer codes instead of strings; a sum can sometimes be computed from compressed representations [7]. This means compression reduces not only storage and the bytes read from disk but also the work the CPU performs, turning what is usually a space-time trade-off into a win on both axes [7], [12]. The integration of compression with execution, rather than treating it as a separate layer below the engine, is one of the defining internal characteristics of a modern column store, and both ClickHouse and DuckDB implement a palette of such encodings selected to keep vectorized kernels fed [8], [9].

Table 1. Lightweight Column Encodings and Their Properties

Encoding	Best for	Operate on compressed?	Typical use
Run-length (RLE)	Long repeated runs	Yes (count, filter)	Sorted / low-cardinality columns
Dictionary	Few distinct values	Yes (compare codes)	Strings, categoricals
Delta / frame-of-reference	Sorted or clustered numerics	Partially	Timestamps, sequential keys
Bit-packing	Small value ranges	Partially	Bounded integers, codes

VI. HOW THE PIECES COMPOUND

The three techniques are not independent optimisations stacked on a shared base; they reinforce one another, which is why their combined effect exceeds the sum of the parts shown in Figure 2. The columnar layout makes data a contiguous typed array, which is what vectorized execution needs and what compresses well [1], [3]. Compression keeps that array small enough to stay in cache and lets the engine compute over encoded values, feeding the vectorized kernels with less data and less work [7], [12]. Late materialization ensures the engine stays in this efficient column-and-vector regime for as long as possible, reconstructing rows only after most data has been filtered away [6]. Remove any one and the others lose force: vectorization over decompressed early-materialized rows would surrender much of the benefit. The architecture is best understood as a single coherent design in which storage layout, execution model, materialization strategy, and compression were chosen together, a lesson reinforced by commercial engines whose column-store acceleration integrates the same techniques [3], [15].

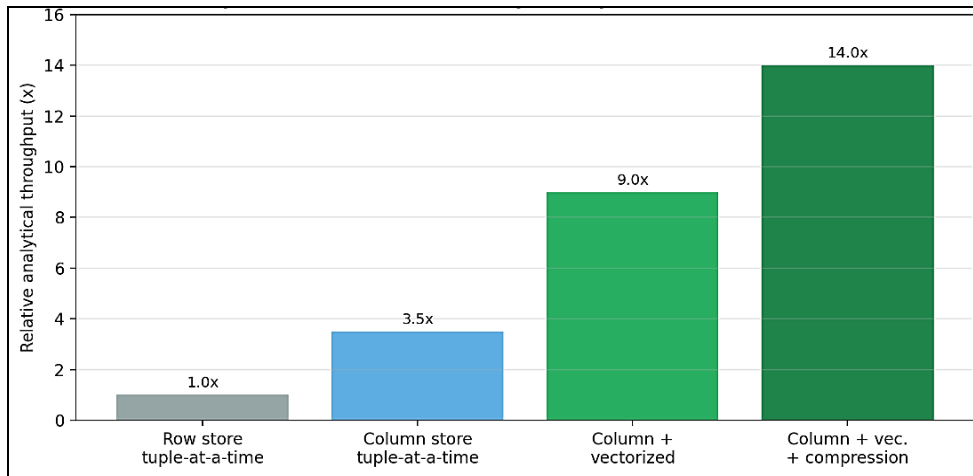

Fig 2: Representative analytical throughput as the column-store techniques are added in turn. The gains compound because the layout, vectorization, and compression reinforce one another rather than acting independently.

VII. INDEXING, ZONE MAPS, AND DATA SKIPPING

The fastest data to process is the data never read, and column stores invest heavily in avoiding irrelevant input. Rather than the fine-grained secondary indexes of transactional systems, which would be ill-suited to scan-heavy analytics, they rely on lightweight metadata that lets the engine skip large blocks of data whose values cannot match a query. The basic device is the zone map, a small summary, typically the minimum and maximum value, kept for each block of a column; when a query filters on that column, the engine consults the zone maps and reads only the blocks whose value range overlaps the predicate, skipping the rest without touching them [19]. Figure 3 illustrates the mechanism. This is especially effective on columns that are sorted or naturally clustered, such as timestamps in an append-mostly table, where a time-range query can eliminate the overwhelming majority of blocks. ClickHouse builds on this idea with a sparse primary index that marks the boundaries of granules along the table's sorting key, and with additional data-skipping indexes including bloom filters for membership tests on high-cardinality columns [9], [20], [22]. DuckDB maintains comparable zone-map statistics to prune row groups during a scan [8]. The common principle is that a little precomputed metadata converts a full scan into a partial one, compounding with the vectorized, compressed execution of the earlier sections.

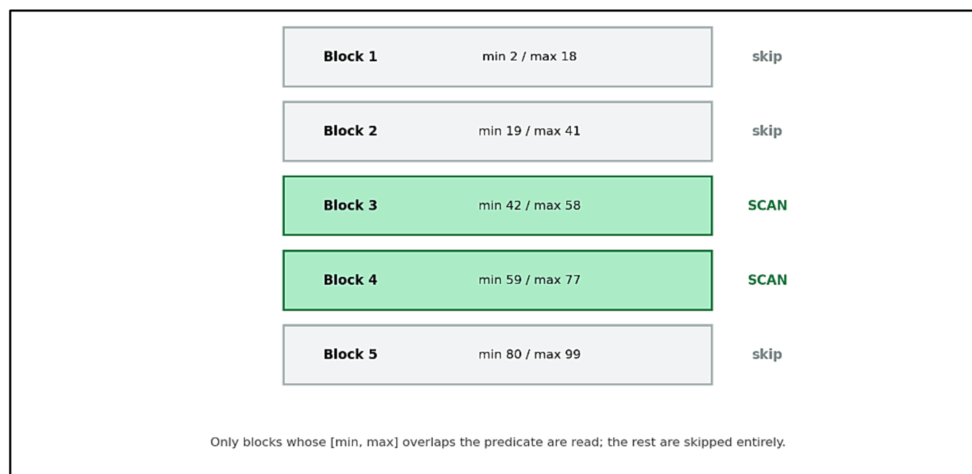

Fig 3: Zone maps turn a full scan into a partial one. Each block carries its value range, and only blocks whose range overlaps the query predicate are read, which is highly effective on sorted or clustered columns.

VIII. MERGE-BASED STORAGE AND HANDLING WRITES

Column stores optimise for reading, which raises the question of how they handle writes, since inserting a single row would otherwise require touching every column file. The dominant answer borrows the log-structured merge discipline from write-optimised key-value stores [18]. Incoming data is written in self-contained sorted parts rather than updated in place, and a background process periodically merges smaller parts into larger ones, keeping data ordered by the sorting key and the per-block metadata accurate [18], [22]. ClickHouse names its principal engine family for exactly this merge behaviour, and the choice of sorting key is the single most

consequential physical-design decision, because it determines both compression effectiveness and the power of data skipping [9], [22]. Updates and deletes, awkward in an append-oriented design, are handled by writing markers or replacement parts that the merge process later reconciles, so the system favours high-throughput appends and bulk loads over frequent small mutations [22]. This is a deliberate trade: analytical engines accept weaker support for in-place updates in exchange for the read performance that sorted, compressed, immutable parts make possible, which is appropriate for the append-mostly nature of analytical data [21].

IX. PARALLELISM, DISTRIBUTION, AND HARDWARE

The techniques described so far make a single core fast; modern engines must also exploit many cores and, for the largest data, many machines. Within a node, analytical engines parallelise a query by partitioning its input into chunks processed concurrently across cores, with a scheduler balancing the work so that no core sits idle, an approach exemplified by morsel-driven execution that has become the template for in-memory analytical parallelism [17]. The columnar, vectorized design is well matched to this, because vectors of values are natural units to distribute and recombine. DuckDB applies exactly this model to extract full performance from a single multi-core machine, which is often sufficient given how much modern hardware a laptop or server now provides [8], [17]. For data beyond one machine, ClickHouse shards tables across nodes and executes queries in a distributed fashion, pushing filtering and partial aggregation to where the data lives so that only compact intermediate results travel over the network [9]. Underlying both is a sensitivity to the hardware reality that motivated the whole architecture: keeping working sets in cache, generating instruction sequences the CPU can vectorize, and spilling gracefully to storage when data exceeds memory [3], [12]. Table 2 contrasts how the distributed ClickHouse and the embeddable DuckDB realise the shared internals.

Table 2. Two Column Stores, Shared Internals, Different Deployment

Aspect	ClickHouse	DuckDB
Deployment model	Distributed, sharded server	Embedded, in-process library
Storage engine	MergeTree family (merge-based parts)	Single-file, row groups
Data skipping	Sparse primary index + skip indexes	Zone-map statistics per row group
Parallelism	Multi-core and multi-node	Multi-core, single node
Primary use	Large-scale interactive analytics	Interactive / embedded analytics

X. CONCLUSION

The column store is the clearest example in database systems of an architecture in which one decision, storing data by column, propagates into a coherent set of mutually reinforcing techniques. Vectorized execution processes batches that suit modern CPUs, late materialization defers row reconstruction until the data has been thinned, and lightweight compression shrinks storage while letting the engine compute over encoded values [4], [6], [7]. Systems as different in deployment as the distributed ClickHouse and the embeddable DuckDB share these internals precisely because the techniques are what make analytical queries fast, independent of where the engine runs [8], [9]. For workloads that scan many rows over few columns, this design now delivers performance that row stores cannot approach, which is why the column-oriented architecture has become the default foundation for analytics. The continuing research questions concern the boundaries, hybrid systems that must serve transactions and analytics together, and how far the same compressed, vectorized, late-materialized discipline can be pushed as hardware evolves [10], [11].

REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really?," in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2008, pp. 967–980.

[2] M. Stonebraker et al., "C-Store: A column-oriented DBMS," in Proc. 31st Int. Conf. Very Large Data Bases (VLDB), 2005, pp. 553–564.

[3] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The design and implementation of modern column-oriented database systems," Found. Trends Databases, vol. 5, no. 3, pp. 197–280, 2013.

[4] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-pipelining query execution," in Proc. 2nd Biennial Conf. Innov. Data Syst. Res. (CIDR), 2005, pp. 225–237.

[5] M. Zukowski, M. van de Wiel, and P. Boncz, "Vectorwise: A vectorized analytical DBMS," in Proc. IEEE 28th Int. Conf. Data Eng. (ICDE), 2012, pp. 1349–1350.

[6] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden, "Materialization strategies in a column-oriented DBMS," in Proc. IEEE 23rd Int. Conf. Data Eng. (ICDE), 2007, pp. 466–475.

[7] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2006, pp. 671–682.

[8] M. Raasveldt and H. Mühleisen, “DuckDB: An embeddable analytical database,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2019, pp. 1981–1984.
[9] ClickHouse, Inc., “ClickHouse: An open-source column-oriented OLAP database management system,” ClickHouse Documentation, 2023.
[10] T. Neumann, “Efficiently compiling efficient query plans for modern hardware,” Proc. VLDB Endowment, vol. 4, no. 9, pp. 539–550, 2011.
[11] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, “Everything you always wanted to know about compiled and vectorized queries but were afraid to ask,” Proc. VLDB Endowment, vol. 11, no. 13, pp. 2209–2222, 2018.
[12] D. Lemire and L. Boytsov, “Decoding billions of integers per second through vectorization,” Softw. Pract. Exper., vol. 45, no. 1, pp. 1–29, 2015.
[13] S. Melnik et al., “Dremel: Interactive analysis of web-scale datasets,” Proc. VLDB Endowment, vol. 3, nos. 1–2, pp. 330–339, 2010.
[14] Apache Software Foundation, “Apache Arrow: A cross-language development platform for in-memory data,” Apache Arrow Documentation, 2023.
[15] V. Raman et al., “DB2 with BLU Acceleration: So much more than just a column store,” Proc. VLDB Endowment, vol. 6, no. 11, pp. 1080–1091, 2013.
[16] G. Graefe, “Volcano—An extensible and parallel query evaluation system,” IEEE Trans. Knowl. Data Eng., vol. 6, no. 1, pp. 120–135, 1994.
[17] V. Leis, P. Boncz, A. Kemper, and T. Neumann, “Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2014, pp. 743–754.
[18] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (LSM-tree),” Acta Informatica, vol. 33, no. 4, pp. 351–385, 1996.
[19] G. Moerkotte, “Small materialized aggregates: A light weight index structure for data warehousing,” in Proc. 24th Int. Conf. Very Large Data Bases (VLDB), 1998, pp. 476–487.
[20] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Commun. ACM, vol. 13, no. 7, pp. 422–426, 1970.
[21] M. Stonebraker et al., “The end of an architectural era (it’s time for a complete rewrite),” in Proc. 33rd Int. Conf. Very Large Data Bases (VLDB), 2007, pp. 1150–1160.
[22] ClickHouse, Inc., “MergeTree engine family,” ClickHouse Documentation, 2023.