



WHITE PAPER · DATA STORAGE ENGINES

Columnar and Vectorized Analytical Engines

Storage layout, execution and the modern CPU

White Paper 3 · Data Storage Engines series · Version 1.0 · 26 October 2026

Bassirah Technology — Innovation & R&D

Columnar and Vectorized Analytical Engines: Storage Layout, Execution and the Modern CPU

White Paper 3 · Data Storage Engines series · Bassirah Technology — Innovation & R&D · 26 October 2026

Document control

Field	Value
Series	Data Storage Engines, paper 3 of 6
Author	Bassirah Technology — Innovation & R&D
Version	1.0 · 26 October 2026
Scope	A technology survey. It describes published research and publicly documented systems; it does not report Bassirah benchmarks or customer results.

Contents

1. Abstract	4
2. Introduction	4
2.1 Why analytical workloads favor columnar layout	4
2.2 Scope	4
2.3 How this paper builds on papers 1 and 2	4
3. Columnar storage fundamentals	5
3.1 Row, column and hybrid layouts	5
3.2 Late materialization	5
3.3 Data skipping and zone maps	5
3.4 Sorting and clustering keys	5
3.5 Parts, granules and background merges	6
4. Encoding and compression	6
4.1 Lightweight encodings	6
4.2 Decompression speed versus compression ratio	6
4.3 SIMD-friendly layouts	6
4.4 Strings and floating-point data	7
4.5 Encoding-first formats	7
5. Vectorized execution	7
5.1 From tuple-at-a-time to vector-at-a-time	7
5.2 Batch sizing and cache residency	7
5.3 Selection vectors and SIMD kernels	8
5.4 Vectorization, code generation and adaptivity	8
6. Operators on modern hardware	8
6.1 Scans and filters	8
6.2 Hash joins	8

6.3 Aggregation and sorting.....	8
6.4 Out-of-core execution.....	9
6.5 Multi-core scheduling and NUMA.....	9
7. Open file formats and the composable engine stack.....	9
7.1 Parquet, ORC and Arrow	9
7.2 Newer formats	9
7.3 Composable and embedded engines	10
8. Cloud and distributed analytical engines.....	10
8.1 Separating storage from compute.....	10
8.2 Object storage as primary storage.....	10
8.3 ClickHouse in distributed and cloud deployments.....	10
8.4 Freshness and hybrid designs	11
9. Comparative analysis.....	11
10. Open challenges and future opportunities.....	12
10.1 Updates, deletes and freshness	12
10.2 Format fragmentation and interoperability.....	12
10.3 Compression versus CPU on new hardware.....	12
10.4 GPU and accelerator offload	13
10.5 Memory capacity, bandwidth and CXL.....	13
10.6 Cost and energy efficiency in the cloud	13
10.7 Robust optimization and execution.....	13
10.8 Semi-structured, nested and AI data	14
11. What this means for platform decisions.....	14
References	14

1. Abstract

Analytical queries read many rows but few columns, aggregate heavily and rarely change data in place. Columnar engines are built around this pattern. This paper examines how they turn it into efficient use of the modern CPU: by storing each column separately, encoding it with lightweight schemes that decode quickly, skipping blocks with small summaries, and processing data in cache-sized vectors with SIMD instructions. It reviews storage layout, encoding and compression, vectorized and compiled execution, and the design of scans, joins, aggregation, sorting and out-of-core processing. It then covers open file formats, composable engines such as Apache DataFusion and Velox, embedded engines such as DuckDB, and cloud designs that separate storage from compute. The paper closes with open challenges, framed as research directions and opportunities, and with notes for platform decisions.

2. Introduction

2.1 Why analytical workloads favor columnar layout

A typical analytical query scans a large table, filters on a few attributes, joins with smaller tables and aggregates the result. A row store reads every attribute of every row it touches. A column store reads only the columns the query names, and values of one type stored together compress well [1]. C-Store made the case for this read-optimized design, storing data by column and packing values carefully both on disk and in memory during query processing [2]. In the same year, MonetDB/X100 showed that database engines used modern CPUs poorly, achieving few instructions per cycle on analytical work, and proposed processing data a vector at a time instead of a tuple at a time [3].

Both ideas now underpin most analytical engines. The ClickHouse authors describe its executor as following the X100 vectorization model [4], and DuckDB was designed as an embedded engine for analytical SQL [5]. The shared aim is to keep the CPU busy on useful work, with data arriving in a form it can process in tight loops.

2.2 Scope

This paper covers engines that store or read columnar data and execute queries with vectorized or compiled code, on one node or across a cluster. It concentrates on peer-reviewed work from 2021 to 2026 and on official documentation, citing earlier papers where they introduced an idea still in use. It does not report benchmark figures. Where a paper reports results, we describe them qualitatively and attribute them to the authors.

2.3 How this paper builds on papers 1 and 2

Paper 1 placed DuckDB and ClickHouse in Class B of a five-class taxonomy, analytical and read-optimized engines, and named memory bandwidth and CPU cache efficiency as their bottleneck. Paper 2 examined LSM-trees, whose immutable sorted runs and background merges reappear here: the ClickHouse authors describe the storage layer of its MergeTree engine as based on log-structured merge trees [4]. This paper opens Class B.

3. Columnar storage fundamentals

3.1 Row, column and hybrid layouts

The row layout, or N-ary storage model, places all attributes of a record together. It suits transactions that read or write whole records. The column layout places all values of one attribute together. It suits scans that touch a few attributes of many records [1].

Most practical formats sit between the two. PAX (Partition Attributes Across) keeps a page's records together but groups each attribute's values inside the page, improving cache use without changing which pages are read [6]. Apache Parquet applies a similar idea at file scale: a file is split into row groups, each holding one column chunk per column, divided into pages, with metadata in a footer [7]. The ClickHouse authors note that Snowflake persists data in hybrid PAX pages, whereas ClickHouse's own format is strictly columnar [4].

3.2 Late materialization

A column store must eventually stitch columns back into rows. Late materialization delays this: the engine filters and joins on the columns it needs, tracks the positions of surviving rows, and fetches other columns only for those rows [1]. Liu et al. argue that the advantages of applying this uniformly have narrowed in vectorized engines, and let each attribute choose its own materialization point by cost [8]. In DuckDB, their approach outperformed both pure early and pure late strategies on the authors' benchmarks.

3.3 Data skipping and zone maps

Small per-block summaries let a scan skip blocks that cannot satisfy a predicate. The simplest is a zone map holding a column's minimum and maximum in a block. Amazon Redshift evaluates predicates over zone maps before scanning [9]. Parquet offers an optional page index with minimum and maximum values per page, plus an offset index for jumping to the matching pages of other columns [7]. Data Blocks added Positional SMAs, which narrow the scan range inside a block that cannot be excluded as a whole [10].

ClickHouse provides skipping indexes, including min-max, set and Bloom-filter types, each defined over a block of consecutive granules [11]. Its documentation is explicit about the limits: a skipping index helps only when the indexed column correlates with the sort order, and otherwise adds cost at insert and query time while skipping little [11].

3.4 Sorting and clustering keys

Skipping works only if similar values sit together, so sort order is a central design decision. In ClickHouse, each data part is sorted by the primary key, and a sparse primary index stores the key of the first row of each granule. Filters on a key prefix are evaluated over key ranges with three-valued logic, and ranges with mixed outcomes are split recursively [4]. Projections store the same rows under another sort order, at extra insert, merge and storage cost [4], an idea that C-Store's overlapping projections anticipated [2].

Cloud warehouses that ingest continuously must recluster to keep pruning effective. WAIR reclusters only the micro-partitions on the boundaries of query ranges; its authors report query performance close to a fully sorted layout at a lower, bounded reclustering cost [12].

3.5 Parts, granules and background merges

ClickHouse's MergeTree engine shows how these elements combine. Each insert creates an immutable part sorted by the primary key, and background merges combine parts [4]. A part is a directory with one file per column. Rows are divided into granules, by default of 8,192 rows, the smallest unit that scan and index operators handle, while I/O works on compressed blocks of neighboring granules [4]. Merges can also collapse or pre-aggregate rows, and time-to-live rules can recompress parts or move them to object storage [4].

The parallel with paper 2 is direct: appending immutable sorted files and merging them later turns random writes into sequential ones, at the cost of merge work and a delay before deleted rows physically disappear.

4. Encoding and compression

4.1 Lightweight encodings

Lightweight encodings exploit a column's structure with simple arithmetic. Dictionary encoding replaces values with small integer codes; run-length encoding (RLE) stores a value once with its repeat count; delta encoding stores differences between consecutive values; frame of reference (FOR) stores values relative to a block minimum, and bit-packing then uses only as many bits per value as the block needs. Parquet defines dictionary, RLE and bit-packing hybrid, several delta encodings and byte stream split, among others [7]. Because decoding is cheap, and some operators can work on encoded values directly, these schemes save bandwidth without costing much CPU [1].

4.2 Decompression speed versus compression ratio

General-purpose codecs such as LZ4 and Zstandard treat data as bytes and must decompress a whole block before any value is used. Zeng et al. evaluated Parquet and ORC on real-world data distributions and modern hardware [13]. They recommend dictionary encoding by default, favoring decoding speed over compression ratio for integers, making block compression optional, and embedding finer-grained auxiliary structures. BtrBlocks argues that with high-throughput networks, inefficient decompression makes data-lake scans CPU-bound, raising query time and cost [14]. It cascades lightweight encodings, chosen by sampling, to combine fast decompression with a good ratio.

Engines resolve the trade-off differently. ClickHouse compresses blocks with LZ4 by default and allows specialized codecs, such as delta and Gorilla, to be chained [4]. Its authors reason that with aggressive pruning, I/O costs outweigh decompression costs for the remaining data, and contrast this with DuckDB, which chose the lightweight methods of Data Blocks for workloads that mix analytics with occasional updates [4], [10]. The two choices rest on different premises about the workload.

4.3 SIMD-friendly layouts

Classic bit-packing stores values one after another, which limits how many a SIMD instruction can decode at once. FastLanes interleaves bit-packed values to suit a virtual 1,024-bit SIMD register and reorders tuples in all columns into a common transposed order, exposing independent work for lane widths of 8 to 64 bits [15]. Its scalar code can be auto-vectorized by compilers, which avoids maintaining platform-specific SIMD code. A vector of 1,024 decoded values fits in L1 or L2 cache, so decompression happens between memory and the CPU [15].

The layout carries over to GPUs. Afroozeh et al. report that FastLanes decompression could make GPU queries faster than on uncompressed data, but found the 1,024-value granularity too large for a single GPU warp because of register pressure, and used smaller mini-vectors [16].

4.4 Strings and floating-point data

Strings are often the largest and slowest columns, and dictionaries help only when whole strings repeat. FSST compresses strings with a static table of frequent substrings [17]. Its authors report speeds comparable to speed-optimized codecs such as LZ4 with better compression factors on text. Because the table is static, individual strings can be decompressed independently, which allows query processing on compressed data.

Floating-point data resists delta and FOR encoding because IEEE 754 doubles do not represent most decimal values exactly. ALP encodes doubles that originated as decimals as integers, and otherwise compresses their leading bits [18]. It is designed for vectorized execution, and its authors report better speed and compression ratio than earlier floating-point schemes on their datasets.

4.5 Encoding-first formats

These results are shaping new formats. The FastLanes file format avoids general-purpose compression in favor of data-parallel lightweight encodings, cascades encodings, compresses correlated columns together, and decodes in small batches that fit CPU and GPU caches [19].

5. Vectorized execution

5.1 From tuple-at-a-time to vector-at-a-time

In the classic iterator model, each operator returns one tuple per call, so interpretation overhead is paid per tuple. X100 kept the iterator structure but passed vectors between operators [3]. Each primitive is a tight loop over arrays that compilers can pipeline and auto-vectorize, and per-call overhead is spread over the vector [15].

DuckDB processes DataChunks, collections of vectors with a default size of 2,048 tuples [20]. A vector can be flat, constant, dictionary or sequence encoded, so compressed data can stay compressed during execution. Velox's Arrow-compatible vectors support similar encodings plus frame of reference, and lazy vectors that are populated only on first use [21].

5.2 Batch sizing and cache residency

Vectors must be large enough to amortize interpretation and small enough for a pipeline's working set to stay in cache. Kersten et al. compared vectorized and compiled execution inside one test system [22]. They found that vectorization is better at hiding cache-miss latency, while compiled code needs fewer instructions, which favors cache-resident work.

Selective operators complicate this. Qiao and Zhang show that hash joins can leave pipelines full of small, sparse chunks that lose most of the benefit of vectorization [23]. They model when to compact chunks, tune the threshold at run time with a learning-based method, and propose logical compaction to limit data movement.

5.3 Selection vectors and SIMD kernels

Vectorized filters rarely copy data. They record surviving rows in a selection vector that later operators read through; Photon's column batch, for example, holds a position list of active rows [24]. ClickHouse compiles hot loops into several kernels, such as scalar, auto-vectorized AVX2 and hand-written AVX-512 versions, and chooses one at run time from the CPU's reported features [4]. One binary then runs on older processors while using wider instructions where they exist.

5.4 Vectorization, code generation and adaptivity

The alternative is data-centric code generation, which compiles each pipeline into machine code that keeps a tuple in registers from scan to pipeline breaker. Kersten et al. concluded that both paradigms are efficient, with different strengths [22]. Compilation latency is the main obstacle for short queries; Umbra reduces it with a single-pass code generator and a backend tuned for low compile time [25].

Production engines increasingly mix the two. Photon chose vectorized interpretation, citing runtime adaptivity, and its authors report that development, debugging and observability were easier than with a code-generating prototype [24]. Photon runs specialized code paths for batch properties such as the absence of NULL values [24]. Redshift blends operators through code generation and adds prefetching and vectorized execution [9]. ClickHouse is vectorized but compiles recurring expressions, aggregate functions and multi-key sort comparisons with LLVM, and its operators can switch to external aggregation or join algorithms at run time when memory runs short [4]. Data Blocks combined a vectorized scan with compiled pipelines [10].

6. Operators on modern hardware

6.1 Scans and filters

A modern scan prunes blocks with zone maps or skipping indexes, decodes the rest into vectors, evaluates predicates into selection vectors and fetches further columns only for surviving rows. Redshift's scans also use SIMD to decompress its lightweight encodings [9]. Encodings with random access, such as FSST, let predicates run without decompressing a whole block [17].

6.2 Hash joins

Research long compared radix-partitioned hash joins, which partition inputs to fit cache before building, with simpler non-partitioned joins. Bandle et al. integrated a radix join into Umbra, with a Bloom-filter semi-join reducer for selective queries [26]. On TPC-H, partitioning paid off in very few joins, and the benefit faded quickly away from ideal parameters, so they advise against adding it to a code-generating engine for real-world workloads. ClickHouse uses a parallel hash join whose threads build a shared, partitioned hash table [4]. Join order matters as much as join algorithm (Section 10.7).

6.3 Aggregation and sorting

ClickHouse instantiates many hash-table variants from one template and chooses among them per operator by the types of the grouping columns, the estimated cardinality and other

factors [4]. Parallel aggregation runs in two phases: threads build local tables of partial states, and a final step merges them [4].

Kuiper and Mühleisen found that sorting rows is almost always more efficient than sorting columnar data, even when columns must be converted to rows and back, and built DuckDB's sort on that result [27].

6.4 Out-of-core execution

When intermediates exceed memory, many engines switch to an external algorithm and performance drops sharply. Kuiper et al. aim for graceful degradation instead. For aggregation, they unify memory management for temporary and persistent data and use a page layout that is efficient in memory but can be spilled [28]. For joins, they add an adaptive external hash join that compresses materialized intermediates at run time and shares memory dynamically among concurrent operators [29]. In DuckDB, the authors report that performance degrades gradually as data exceeds the memory limit. Umbra's buffer manager, with variable-size pages, takes a related approach to data on SSDs [30].

6.5 Multi-core scheduling and NUMA

Morsel-driven parallelism splits input into small fragments, or morsels, that worker threads take from a dispatcher and push through a whole pipeline [31]. The degree of parallelism can change during execution, and the dispatcher keeps work on memory local to each NUMA node. ClickHouse instead fixes the maximum degree of parallelism in the plan, partitions input into larger ranges, and rebalances work between stages with exchange operators; its authors acknowledge that this can stall when lanes progress unevenly [4].

7. Open file formats and the composable engine stack

7.1 Parquet, ORC and Arrow

Parquet and ORC were designed in the early 2010s for the Hadoop ecosystem and became the standard formats of data lakes [13]. Apache Arrow plays the complementary role in memory: a language-independent columnar layout with contiguous buffers, validity bitmaps and aligned, padded buffers suited to SIMD, shareable between processes without copying [32]. Parquet or ORC at rest and Arrow in memory is now a common division of labor.

7.2 Newer formats

Several new formats respond to the limits Zeng et al. identified [13] and to new workloads.

- **Lance** targets AI workloads that need both scans and random access. A 2025 preprint argues that structural encoding, how repetition and validity information are stored, governs random-access performance [33].
- **Nimble**, from Meta, targets very wide tables such as machine-learning feature tables and separates encodings from the physical layout; its repository describes it as a work in progress [34].
- **Vortex** combines a file format with an extensible compression framework, cascading encodings and kernels that compute on encoded data; it is an incubation project of LF AI & Data [35].
- **FastLanes** builds its format on data-parallel encodings (Section 4.5) [19].

Two research proposals address format evolution directly. F3 embeds WebAssembly decoders in each file, so a reader without a native decoder can still decode the data [36]. AnyBlox bundles WebAssembly decoders with datasets, decoupling decoders from engines and format specifications [37].

7.3 Composable and embedded engines

Execution engines are becoming reusable components. Velox is a C++ library of vectorized, dialect-independent components; at Meta it is integrated with Presto, Spark and machine-learning preprocessing, among others [21]. Prestissimo replaced Presto's Java workers with Velox-based C++ workers in a phased, in-place migration, and its authors report gains in latency, hardware efficiency and reliability [38]. Apache DataFusion is an embeddable Rust engine that uses Arrow as its memory model and exposes many extension points; its authors argue, with comparisons against DuckDB, that open standards and modular design need not cost performance [39]. Standard plan formats make engines swappable: Sirius accepts Substrait plans so that a GPU engine can replace a CPU engine behind an existing interface [40]. Trino, a distributed SQL query engine, queries heterogeneous sources, including Iceberg tables stored as Parquet, ORC or Avro on object storage [41].

DuckDB showed that analytical SQL can run inside the application process, as SQLite does for transactional work, avoiding data transfer between database and application [5]. ClickHouse has followed with an in-process mode, chDB, which its authors describe as inspired by DuckDB [4].

8. Cloud and distributed analytical engines

8.1 Separating storage from compute

Cloud warehouses keep data in shared storage and scale compute separately. Snowflake's engineers describe a design built for elasticity, multi-tenancy and performance, and use production data to show how new hardware and fine-grained billing have changed its design assumptions [42]. Redshift disaggregated storage and compute and added tiered storage, multi-cluster scaling, data sharing, the AQUA acceleration layer and a serverless option [9]. The lakehouse pattern extends separation to open formats read directly from object storage [43], the setting Photon was built for [24].

8.2 Object storage as primary storage

Object storage has high bandwidth, high latency and a cost per request [13]. Durner et al. found the gap between network and local NVMe bandwidth closing [44]. They derive cost- and performance-optimal retrieval configurations and present AnyBlob, a download manager that reduces CPU use; integrated into Umbra without local caching, it performed comparably to cloud warehouses that cache data on local SSDs. When data arrives this fast, decompression can become the bottleneck, which strengthens the case for lightweight encodings [14].

8.3 ClickHouse in distributed and cloud deployments

Self-managed ClickHouse scales out by sharding tables and replicating each shard, coordinated by Keeper, a C++ coordination service based on Raft consensus [4]. For sharded

tables, the initiating node pushes as much work as possible to other nodes, which can return raw columns, filtered rows, partial aggregation states or complete results [4].

ClickHouse Cloud uses the SharedMergeTree engine family [45]. Data lives in shared object storage, metadata is coordinated through Keeper, and replicas do not communicate directly, so adding or removing compute does not require copying metadata. The ClickHouse authors plan to describe the cloud architecture in a later publication [4], so this paper relies on the official documentation.

8.4 Freshness and hybrid designs

Analytical engines are often asked to serve fresh operational data. TiDB adds Raft learner replicas that convert rows to columns, giving analytical queries a fresh, consistent column store isolated from transactions [46]. F1 Lightning adds HTAP capability as a loosely coupled service over existing transactional systems [47]. ClickHouse, optimized for appends, supports lightweight deletes that mask rows at query time until merges remove them physically [4], [11].

9. Comparative analysis

Table 1 compares engine families and design choices discussed in this paper. The ratings are qualitative and relative to each other. Actual behavior depends on data, schema, query mix and configuration.

Engine family or design choice	Scan and aggregation efficiency	Update and delete handling	Elasticity	Open-format interoperability	Best-fit workloads
Sorted parts with sparse index and background merges (ClickHouse MergeTree)	High	Medium	Medium; High with shared object storage	Medium	Real-time analytics on append-heavy events, logs and time series with many concurrent queries
Embedded vectorized engine (DuckDB)	High	Medium to High	Low (single process)	High	Interactive and local analytics, data science, querying files in place
Data-centric compiled engine (HyPer, Umbra)	High	Medium to High	Low to Medium	Medium	Low-latency ad hoc queries where compilation cost is controlled
Composable vectorized library (Velox, DataFusion, Photon)	High	Depends on host system	Depends on host system	High	Building or modernizing engines; consistent semantics across systems

Engine family or design choice	Scan and aggregation efficiency	Update and delete handling	Elasticity	Open-format interoperability	Best-fit workloads
Federated distributed SQL over open formats (Trino)	Medium	Depends on table format	High	High	Ad hoc SQL across lakehouse tables and multiple sources
Disaggregated cloud warehouse (Snowflake, Redshift)	Medium to High (cache-dependent)	Medium	High	Medium	Variable, multi-tenant workloads that benefit from elastic compute
HTAP with a columnar replica (TiDB, F1 Lightning)	Medium	High	Medium	Low	Analytics on fresh transactional data
GPU-native SQL engine (Sirius)	High when the working set fits accelerator memory	Depends on host system	Low to Medium	Medium	Scan- and join-heavy queries on bounded working sets; early stage

Table 1: Qualitative comparison of analytical engine families and design choices.

10. Open challenges and future opportunities

10.1 Updates, deletes and freshness

Challenge. Columnar layouts and heavy compression favor appends. Updates and deletes are usually handled with masks, delta structures or rewrites, so deleted data lingers and reads pay for filtering until merges catch up [4], [11]. **Research direction.** HTAP designs replicate changes into column stores with consistent reads [46], [47], and DuckDB pairs lightweight compression with multi-version concurrency control to support occasional updates [4].

Opportunity. Engines that expose freshness and physical-deletion lag as explicit, bounded properties would help teams serve operational analytics and meet deletion obligations, a theme paper 2 raised for LSM-trees.

10.2 Format fragmentation and interoperability

Challenge. Parquet and ORC carry design decisions from the early 2010s [13], and new formats such as Lance, Nimble, Vortex and FastLanes each address different gaps [33], [34], [35], [19]. Every new format adds reader and writer work to every engine. **Research direction.** F3 and AnyBlox ship WebAssembly decoders with the data, decoupling encodings from engines [36], [37]. **Opportunity.** A stable extension mechanism inside the established formats, rather than a succession of replacements, would let research encodings reach production without breaking the ecosystem. Paper 5 will examine the table formats layered above these files.

10.3 Compression versus CPU on new hardware

Challenge. As network and storage bandwidth grow, decompression can become the bottleneck, and the right balance between compression ratio and decoding speed shifts with

each hardware generation [14], [44]. **Research direction.** Data-parallel layouts and adaptive, sampled encoding choices aim for both a good ratio and fast decoding [15], [18], [14].

Opportunity. Engines could choose encodings per block from measured cost models of the target hardware, and revisit those choices as data ages or moves between tiers.

10.4 GPU and accelerator offload

Challenge. GPUs offer high memory bandwidth and parallelism, but limited memory capacity, data movement over the host interconnect and cost have kept GPU databases niche [40].

Research direction. Sirius argues that larger GPU memory, faster interconnects and mature libraries such as libcudf have removed these barriers; its authors report better cost efficiency than CPU-only DuckDB on their benchmarks [40]. Compressed execution on GPUs is advancing [16], and Redshift's AQUA layer pushes scans and aggregations to an FPGA-based, off-cluster cache [9]. **Opportunity.** Composable engines make offload a per-operator decision; the open questions are when data movement pays for itself and how to share accelerators with machine-learning work.

10.5 Memory capacity, bandwidth and CXL

Challenge. Analytical engines are often bound by memory bandwidth, and large intermediates exceed memory capacity. **Research direction.** Ahn et al. evaluated commercial CXL memory devices with SAP HANA [48]. They found that the effect of CXL's higher latency and lower bandwidth depends on each data structure's access pattern, with analytical workloads affected more than transactional ones, and that CXL shared memory can shorten restart times. Graceful spilling to SSD is a complementary path [28], [29], [30].

Opportunity. Placing data structures across DRAM, CXL memory and SSD by access pattern is a natural next step for vectorized engines.

10.6 Cost and energy efficiency in the cloud

Challenge. In the cloud, every request, byte scanned and core-second has a price, and caching, clustering and scaling choices interact. **Research direction.** Work on object-storage retrieval [44], workload-aware reclustering [12] and operational studies of elastic warehouses [42] treats cost as a design target. Bachras and Jacobsen argue that energy, carbon emissions and water use should become first-class design criteria alongside performance [49].

Opportunity. Optimizers that weigh monetary cost and energy along with latency, and report them per query, would make trade-offs visible to the teams who pay for them.

10.7 Robust optimization and execution

Challenge. A poor join order or a misestimated cardinality can make an analytical query much slower than necessary, and sparse batches after selective operators erode vectorization [23]. **Research direction.** Diamond hardened joins split joins into lookup and expand steps that the optimizer can reorder to avoid oversized intermediates [50]. Robust Predicate Transfer makes acyclic queries provably robust to join order [51]. Zhang et al. found that simple adaptive query processing can compete with reinforcement-learning optimizers, without training and with interpretable behavior [52]. **Opportunity.** Robust operators combined with runtime adaptation could make ad hoc performance more predictable with less manual tuning.

10.8 Semi-structured, nested and AI data

Challenge. JSON, nested types, embeddings and wide feature tables do not fit fixed relational schemas, and storing them as opaque blobs forfeits columnar benefits. **Research direction.** JSON Tiles extracts frequent keys into columnar form automatically and collects statistics for the optimizer [53]. ClickHouse's JSON type stores paths as subcolumns up to a limit and keeps rarer paths in a shared structure [11]. Lance and Nimble target random access and wide tables for machine-learning pipelines [33], [34]. **Opportunity.** Engines that adapt physical layout to the observed structure of the data, and serve both scans and point lookups, would narrow the gap between analytical and AI storage. Paper 4 examines vector storage engines.

11. What this means for platform decisions

No single analytical engine design fits every workload, so the layout, encoding and execution model should follow from each workload's scan, freshness, concurrency and cost profile. In Bassirah's platform work, ClickHouse, a Strategic Technology Partner, serves real-time analytics, either as partner-led ClickHouse or as ClickHouse OSS; federated SQL over Apache Iceberg tables stored as Parquet uses Trino OSS or Starburst, a Strategic Technology Partner; and Apache Superset or Strategy BI (from Strategy, a Strategic Technology Partner) provides business intelligence on top. In each case, sort keys, encodings and engine settings are validated on a representative workload before adoption, and no outcome is assumed from published results.

References

1. Abadi, D., Boncz, P., Harizopoulos, S., Idreos, S. and Madden, S. (2013). The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3), 197–280. (<https://doi.org/10.1561/19000000024>)
2. Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N. and Zdonik, S. (2005). C-Store: A Column-oriented DBMS. *VLDB*, 553–564. (<https://www.vldb.org/archives/website/2005/program/paper/thu/p553-stonebraker.pdf>)
3. Boncz, P., Zukowski, M. and Nes, N. (2005). MonetDB/X100: Hyper-Pipelining Query Execution. *CIDR*. (<https://www.cidrdb.org/cidr2005/papers/P19.pdf>)
4. Schulze, R., Schreiber, T., Yatsishin, I., Dahimene, R. and Milovidov, A. (2024). ClickHouse - Lightning Fast Analytics for Everyone. *PVLDB*, 17(12), 3731–3744. (<https://doi.org/10.14778/3685800.3685802>)
5. Raasveldt, M. and Mühleisen, H. (2019). DuckDB: an Embeddable Analytical Database. *SIGMOD (demonstration)*, 1981–1984. (<https://doi.org/10.1145/3299869.3320212>)
6. Ailamaki, A., DeWitt, D. J., Hill, M. D. and Skounakis, M. (2001). Weaving Relations for Cache Performance. *VLDB*, 169–180. (<https://www.vldb.org/conf/2001/P169.pdf>)
7. Apache Parquet project (2026). File Format; Encodings; Page Index. Official documentation, accessed 26 October 2026. (<https://parquet.apache.org/docs/file-format/>; <https://parquet.apache.org/docs/file-format/data-pages/encodings/>; <https://parquet.apache.org/docs/file-format/pageindex/>)

8. Liu, Y., Tang, S., Hui, Y., Zhou, H. and Zhang, H. (2025). Selective Late Materialization in Modern Analytical Databases. PVLDB, 18(11), 4616–4628. (<https://doi.org/10.14778/3749646.3749717>)
9. Armenatzoglou, N., Basu, S., Bhanoori, N., Cai, M., Chainani, N., Chinta, K., Govindaraju, V., Green, T. J., Gupta, M., Hillig, S., et al. (2022). Amazon Redshift Re-invented. SIGMOD, 2205–2217. (<https://doi.org/10.1145/3514221.3526045>)
10. Lang, H., Mühlbauer, T., Funke, F., Boncz, P. A., Neumann, T. and Kemper, A. (2016). Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. SIGMOD, 311–326. (<https://doi.org/10.1145/2882903.2882925>)
11. ClickHouse (2026). Understanding ClickHouse Data Skipping Indexes; Lightweight DELETE; JSON Data Type. Official documentation, accessed 26 October 2026. (<https://clickhouse.com/docs/optimize/skipping-indexes>; <https://clickhouse.com/docs/guides/developer/lightweight-delete>; <https://clickhouse.com/docs/sql-reference/data-types/newjson>)
12. Liu, Y., Zhou, R., Yan, J. and Zhang, H. (2026). Workload-Aware Incremental Reclustering in Cloud Data Warehouses. Proceedings of the ACM on Management of Data (SIGMOD), 4(3). (<https://doi.org/10.1145/3802127>)
13. Zeng, X., Hui, Y., Shen, J., Pavlo, A., McKinney, W. and Zhang, H. (2023). An Empirical Evaluation of Columnar Storage Formats. PVLDB, 17(2), 148–161. (<https://doi.org/10.14778/3626292.3626298>)
14. Kuschewski, M., Sauerwein, D., Alhomssi, A. and Leis, V. (2023). BtrBlocks: Efficient Columnar Compression for Data Lakes. Proceedings of the ACM on Management of Data (SIGMOD), 1(2). (<https://doi.org/10.1145/3589263>)
15. Afroozeh, A. and Boncz, P. (2023). The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code. PVLDB, 16(9), 2132–2144. (<https://doi.org/10.14778/3598581.3598587>)
16. Afroozeh, A., Felius, L. and Boncz, P. (2024). Accelerating GPU Data Processing using FastLanes Compression. International Workshop on Data Management on New Hardware (DaMoN). (<https://doi.org/10.1145/3662010.3663450>)
17. Boncz, P., Neumann, T. and Leis, V. (2020). FSST: Fast Random Access String Compression. PVLDB, 13(11), 2649–2661. (<https://doi.org/10.14778/3407790.3407851>)
18. Afroozeh, A., Kuffó, L. X. and Boncz, P. (2023). ALP: Adaptive Lossless floating-Point Compression. Proceedings of the ACM on Management of Data (SIGMOD), 1(4). (<https://doi.org/10.1145/3626717>)
19. Afroozeh, A. and Boncz, P. (2025). The FastLanes File Format. PVLDB, 18(11), 4629–4643. (<https://doi.org/10.14778/3749646.3749718>)
20. DuckDB Foundation (2026). Execution Format. Official documentation, accessed 26 October 2026. (<https://duckdb.org/docs/stable/internals/vector>)
21. Pedreira, P., Erling, O., Basmanova, M., Wilfong, K., Sakka, L., Pai, K., He, W. and Chattopadhyay, B. (2022). Velox: Meta's Unified Execution Engine. PVLDB, 15(12), 3372–3384. (<https://doi.org/10.14778/3554821.3554829>)
22. Kersten, T., Leis, V., Kemper, A., Neumann, T., Pavlo, A. and Boncz, P. (2018). Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. PVLDB, 11(13), 2209–2222. (<https://doi.org/10.14778/3275366.3275370>)

23. Qiao, Y. and Zhang, H. (2025). Data Chunk Compaction in Vectorized Execution. Proceedings of the ACM on Management of Data (SIGMOD), 3(1). (<https://doi.org/10.1145/3709676>)
24. Behm, A., Palkar, S., Agarwal, U., Armstrong, T., Cashman, D., Dave, A., Greenstein, T., Hovsepian, S., Johnson, R., Sai Krishnan, A., et al. (2022). Photon: A Fast Query Engine for Lakehouse Systems. SIGMOD, 2326–2339. (<https://doi.org/10.1145/3514221.3526054>)
25. Kersten, T., Leis, V. and Neumann, T. (2021). Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. The VLDB Journal, 30(5), 883–905. (<https://doi.org/10.1007/s00778-020-00643-4>)
26. Bandle, M., Giceva, J. and Neumann, T. (2021). To Partition, or Not to Partition, That is the Join Question in a Real System. SIGMOD, 168–180. (<https://doi.org/10.1145/3448016.3452831>)
27. Kuiper, L. and Mühleisen, H. (2023). These Rows Are Made for Sorting and That's Just What We'll Do. IEEE ICDE, 2050–2062. (<https://doi.org/10.1109/ICDE55515.2023.00159>)
28. Kuiper, L., Boncz, P. and Mühleisen, H. (2024). Robust External Hash Aggregation in the Solid State Age. IEEE ICDE, 3753–3766. (<https://doi.org/10.1109/ICDE60146.2024.00288>)
29. Kuiper, L., Groß, P., Boncz, P. and Mühleisen, H. (2025). Saving Private Hash Join. PVLDB, 18(8), 2748–2760. (<https://doi.org/10.14778/3742728.3742762>)
30. Neumann, T. and Freitag, M. (2020). Umbra: A Disk-Based System with In-Memory Performance. CIDR. (<https://www.cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf>)
31. Leis, V., Boncz, P., Kemper, A. and Neumann, T. (2014). Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. SIGMOD, 743–754. (<https://doi.org/10.1145/2588555.2610507>)
32. Apache Arrow project (2026). Arrow Columnar Format. Official specification, accessed 26 October 2026. (<https://arrow.apache.org/docs/format/Columnar.html>)
33. Pace, W., She, C., Xu, L., Jones, W., Lockett, A., Wang, J. and Shah, R. (2025). Lance: Efficient Random Access in Columnar Storage through Adaptive Structural Encodings. arXiv preprint 2504.15247. (<https://arxiv.org/abs/2504.15247>)
34. Meta Platforms (2026). Nimble. Official repository, accessed 26 October 2026. (<https://github.com/facebookincubator/nimble>)
35. Vortex project (2026). Vortex. Official repository and documentation, accessed 26 October 2026. (<https://github.com/vortex-data/vortex>)
36. Zeng, X., Meng, R., Prammer, M., McKinney, W., Patel, J. M., Pavlo, A. and Zhang, H. (2025). F3: The Open-Source Data File Format for the Future. Proceedings of the ACM on Management of Data (SIGMOD), 3(4). (<https://doi.org/10.1145/3749163>)
37. Gienieccko, M., Kuschewski, M., Neumann, T., Leis, V. and Giceva, J. (2025). AnyBlox: A Framework for Self-Decoding Datasets. PVLDB, 18(11), 4017–4031. (<https://doi.org/10.14778/3749646.3749672>)
38. Dutta, A., Pedreira, P., Meng, X., Basmanova, M., Erling, O., Pershin, S., Tan, J., Gao, G., Wang, K., Wen, Z., et al. (2026). From Presto to Prestissimo: A Velox-Powered Modernization Journey. PVLDB, 19(12), 4399–4412. (<https://doi.org/10.14778/3827998.3828041>)
39. Lamb, A., Shen, Y., Heres, D., Chakraborty, J., Kabak, M. O., Hsieh, L.-C. and Sun, C. (2024). Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. Companion of SIGMOD, 5–17. (<https://doi.org/10.1145/3626246.3653368>)

40. Yogatama, B., Yang, Y., Kristensen, K., Sarda, D., Kim, A., Cockcroft, A., Teng, Y., Patterson, J., Kimball, G., McKinney, W., Gong, W. and Yu, X. (2026). Rethinking Analytical Processing in the GPU Era. CIDR. (<https://www.vldb.org/cidrdb/papers/2026/p12-yogatama.pdf>)
41. Trino Software Foundation (2026). Trino Documentation: Overview; Iceberg Connector. Official documentation, accessed 26 October 2026. (<https://trino.io/docs/current/overview.html>; <https://trino.io/docs/current/connector/iceberg.html>)
42. Vuppapapati, M., Miron, J., Agarwal, R., Truong, D., Motivala, A. and Cruanes, T. (2020). Building An Elastic Query Engine on Disaggregated Storage. USENIX NSDI, 449–462. (<https://www.usenix.org/conference/nsdi20/presentation/vuppapapati>)
43. Armbrust, M., Ghodsi, A., Xin, R. and Zaharia, M. (2021). Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. CIDR. (https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf)
44. Durner, D., Leis, V. and Neumann, T. (2023). Exploiting Cloud Object Storage for High-Performance Analytics. PVLDB, 16(11), 2769–2782. (<https://doi.org/10.14778/3611479.3611486>)
45. ClickHouse (2026). SharedMergeTree Table Engine. Official ClickHouse Cloud documentation, accessed 26 October 2026. (<https://clickhouse.com/docs/cloud/reference/shared-merge-tree>)
46. Huang, D., Liu, Q., Cui, Q., Fang, Z., Ma, X., Xu, F., Shen, L., Tang, L., Zhou, Y., Huang, M., et al. (2020). TiDB: A Raft-based HTAP Database. PVLDB, 13(12), 3072–3084. (<https://doi.org/10.14778/3415478.3415535>)
47. Yang, J., Rae, I., Xu, J., Shute, J., Yuan, Z., Lau, K., Zeng, Q., Zhao, X., Ma, J., Chen, Z., et al. (2020). F1 Lightning: HTAP as a Service. PVLDB, 13(12), 3313–3325. (<https://doi.org/10.14778/3415478.3415553>)
48. Ahn, M., Willhalm, T., May, N., Lee, D., Desai, S. M., Booss, D., Kim, J., Singh, N., Ritter, D. and Rebholz, O. (2024). An Examination of CXL Memory Use Cases for In-Memory Database Management Systems using SAP HANA. PVLDB, 17(12), 3827–3840. (<https://doi.org/10.14778/3685800.3685809>)
49. Bachras, M. and Jacobsen, H.-A. (2025). Environmental Footprints of Query Processing: A Vision for Sustainable Database Architectures. PVLDB, 18(11), 4064–4072. (<https://doi.org/10.14778/3749646.3749676>)
50. Birler, A., Kemper, A. and Neumann, T. (2024). Robust Join Processing with Diamond Hardened Joins. PVLDB, 17(11), 3215–3228. (<https://doi.org/10.14778/3681954.3681995>)
51. Zhao, J., Su, K., Yang, Y., Yu, X., Koutris, P. and Zhang, H. (2025). Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. Proceedings of the ACM on Management of Data (SIGMOD), 3(3). (<https://doi.org/10.1145/3725283>)
52. Zhang, Y., Chronis, Y., Patel, J. M. and Rekatsinas, T. (2023). Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis. PVLDB, 16(11), 2962–2975. (<https://doi.org/10.14778/3611479.3611501>)
53. Durner, D., Leis, V. and Neumann, T. (2021). JSON Tiles: Fast Analytics on Semi-Structured Data. SIGMOD, 445–458. (<https://doi.org/10.1145/3448016.3452809>)

Product and project names are trademarks of their respective owners. Their mention describes publicly documented technology and does not imply a business relationship, except where Bassirah names a Strategic Technology Partner.