LiquidCache: Efficient Pushdown Caching for Cloud-Native Data Analytics

Xiangpeng Hao
University of Wisconsin-Madison
xiangpeng.hao@wisc.edu

Andrew Lamb
InfluxData
alamb@influxdata.com

Yibo Wu
University of Wisconsin-Madison
wu668@wisc.edu

Andrea Arpaci-Dusseau
University of Wisconsin-Madison
dusseau@cs.wisc.edu

Remzi Arpaci-Dusseau
University of Wisconsin-Madison
remzi@cs.wisc.edu

ABSTRACT

We present LiquidCache, a novel pushdown-based disaggregated caching system that evaluates filters on cache servers before transmitting data to compute nodes. Our key observation is that data decoding, not filter evaluation, is the primary bottleneck in existing systems. To address this challenge, we transcode Parquet data into a lightweight "Liquid" format optimized for caching and filter evaluation. This format is co-designed with filter evaluation semantics to enable selective decoding, late filter materialization, and encoding-aware filter evaluation, delivering low decoding costs while preserving high compression ratios. The "Liquid" format exists exclusively in the cache, allowing easy adoption without breaking ecosystem compatibility. Through integration with Apache DataFusion and evaluation with ClickBench and TPC-H, we demonstrate that LiquidCache reduces cache CPU time by up to 10× without increasing memory footprint, and reduces network traffic by two orders of magnitudes compared to non-pushdown systems.

PVLDB Reference Format:
Xiangpeng Hao, Andrew Lamb, Yibo Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. LiquidCache: Efficient Pushdown Caching for Cloud-Native Data Analytics. PVLDB, 14(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
https://github.com/XiangpengHao/liquid-cache.

1 INTRODUCTION

Cloud-native analytical systems [6, 8, 11, 13, 26, 30, 35, 39, 63, 71] employ compute-storage disaggregation. In this architecture, compute nodes fetch data on demand from remote object stores, typically in Parquet format [37, 79] – the industry standard for analytical data. Despite disaggregation's benefits, reading from object stores incurs high access latency and per-request billing costs [27].

To mitigate these costs, industry employs disaggregated caching [4, 5, 22, 28, 74] with independently scalable shared cache servers. Yet this approach creates a critical challenge: compute and cache communicate via network, which is can easily become a bottleneck for analytical workloads which often transfer large amount of data. This network bottleneck ultimately limits the cache servers to primarily caching metadata rather than data [3, 4, 26, 81].

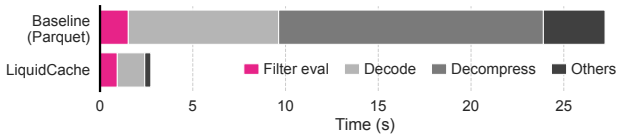

Figure 1: Time breakdown of filter pushdown for ClickBench Q22 – Data decoding and decompression consume over 90% of CPU time in Parquet, while filter evaluation takes less than 10%.

In this paper, we propose a new caching approach that leverages filter pushdown [7, 16, 18, 23, 36, 55, 65, 75, 76] – a classic database optimization technique – to reduce network traffic in disaggregated caching. The approach targets low-latency analytics – dashboards, interactive queries, LLM knowledge retrieval, and anomaly detection pipelines – where predicates typically reduce the result set by orders of magnitude and minimizing retrival latency is a key system goal. By pushing these filters to computationally-capable cache servers, we can prune irrelevant data before transmitting it over the network.

Despite this seemingly simple idea, evaluating filters on cache servers with limited computing power creates CPU bottlenecks, particularly when processing Parquet files. Studies show that filter pushdown to object storage even causes slowdown [51, 77]. Consequently, many cloud-native analytical systems [8, 21] have disabled filter pushdown for Parquet files, incurring prohibitively high network costs.

Our study reveals that the main bottleneck in filter pushdown is data decoding – a CPU-intensive task independent of filter complexity. Data decoding transforms disk-optimized data formats into memory layouts suitable for vectorized execution. While decoding simple linear formats like CSV or JSON is straightforward, Parquet decoding is more complex. Parquet employs sophisticated encoding schemes including nested data structures, cascading encoding, variable-length fields, and rich metadata for optimizations like data skipping. These features make Parquet highly efficient for storage but significantly increase decoding complexity. As shown in Figure 1, data decoding and decompression consume over 90% of baseline (Parquet) processing time, while actual filter evaluation takes less than 10%. With Parquet being the predominant format for analytical

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

data, cache servers must be able to decode and evaluate filters on Parquet data efficiently.

We propose LiquidCache, an efficient pushdown-enabled cache implemented on Apache DataFusion [41] – the state-of-the-art analytical system for Parquet built on open standards. Our key insight is to decouple logical data from its physical representation, where the cache server interprets Parquet data from object storage and gradually transcodes it into Liquid format. This Liquid format is co-designed with the filter pushdown logic, combining techniques like selective decoding, late filter materialization, and encoding-aware predicate evaluation to achieve significantly lower decoding cost than Parquet, while maintaining a similar compression ratio.

Rather than requiring a disruptive migration from Parquet to a new file format for all object storage data – which would break ecosystem compatibility and slow down adoption – LiquidCache takes a more pragmatic approach. It transparently transcodes Parquet data into "Liquid" format as data is accessed and cached on the server. By combining fine-grained batch-level transcoding, lightweight encoding transformations, and asynchronous background processing, LiquidCache hides the transcoding overhead while delivering seamless performance improvements. This incremental approach allows existing systems to adopt LiquidCache without modifying their data infrastructure and enables LiquidCache itself to incorporate future encodings without breaking compatibility.

We performed a rigorous evaluation of LiquidCache on ClickBench [19, 20] and TPC-H. The results show that LiquidCache achieves 10× lower decoding time than Parquet, 4× lower data size than Arrow, and two orders of magnitude lower network traffic than non-pushdown cache systems. Our contributions are:

- LiquidCache: a pushdown-enabled, disaggregated cache system that is first-of-its-kind for cloud-native analytics built upon production-grade systems.
- We design a novel LiquidCache format that co-designs with filter evaluation to effectively reduce the decoding cost.
- We show that LiquidCache can leverage state-of-the-art encodings [1, 17, 38] without breaking the Parquet ecosystem, allowing existing data analytical systems to benefit from our optimizations without compatibility concerns.
- We explore the design spaces of disaggregated cache, quantify their trade-offs using ClickBench and TPC-H, and show LiquidCache achieves 10x lower CPU usage than Parquet without increasing memory usage.

2 BACKGROUND

2.1 Cache for object storage

To reduce object storage latency, caching layers are commonly deployed between compute nodes and object storage. Over time, three main caching architectures have emerged: private cache, distributed cache, and disaggregated cache.

The private cache is built inside each compute node, utilizing spare memory or disk resources to cache data locally. This design is found in research systems like Crystal [26] and industry solutions like Amazon Redshift [6], Databricks [22], and BauPlan [70]. While the private cache is the simplest architecture to implement and deploy, it suffers from inefficient resource utilization since each compute node maintains an independent cache, caching duplicate data when nodes access the same data.

Distributed caching improves resource utilization by using distributed algorithms to connect multiple compute nodes into a single logical cache layer. This design is found in systems like Snowflake [74] and Alluxio [4]. While this approach eliminates data redundancy, it often requires complex consensus protocols to ensure data consistency across nodes. Additionally, the query engine has to carefully migrate data among nodes to maintain data locality and load balance. These requirements introduce significant complexity and performance overhead compared to simpler caching architectures.

The disaggregated cache represents a modern architectural paradigm that fully separates caching infrastructure from computing resources. In this design, cache servers are independent, dedicated services shared across multiple compute nodes. This approach has been adopted by major industry systems, including Google's Napa [3], FoundationDB [81], and Snowflake [74]. The key advantage of disaggregation is independent scalability – cache capacity can be expanded by adding memory or storage without impacting compute resources. However, compute nodes must access the cache over the network, which can become a performance bottleneck. This limitation has led many systems to use a disaggregated cache to store metadata rather than actual data.

2.2 Filter Pushdown

Filter pushdown [16, 18, 75, 76] is a classical optimization technique in database systems that evaluates predicates early in query execution. By pushing filters closer to data sources, systems reduce data processing and transmission volumes by discarding irrelevant rows before they reach higher query processing layers.

Filter pushdown has evolved across multiple stages of query execution. At the most basic level, query optimizers implement filter pushdown as a transformation rule that moves filter operators closer to data scanning operators in the query plan. This reduces the volume of intermediate data processed by subsequent operators. Advanced systems push filters down to specialized storage hardware like FPGAs [65], DPUs [34], Smart SSDs [23], and SmartNICs [36]. These hardware accelerators contain dedicated computing capabilities to evaluate filters while scanning data, ensuring that only relevant records are emitted upstream. In modern environments, major providers have integrated filter pushdown directly into their object storage services, e.g., Amazon S3 Select [7] and Azure Data Lake Storage [55] allow filters to be evaluated within the storage layer before data is transferred to compute nodes.

Despite being a classical optimization technique, filter pushdown faces several key challenges. First, filter evaluation can be computationally expensive, especially on lower-level hardware with limited processing capabilities. Second, after filter evaluation, data is transmitted in an uncompressed in-memory format, which can be significantly larger than the original compressed data – even after filtering. While re-compressing the filtered data could help, this would place an additional computational burden on the already resource-constrained lower-level hardware.

2.3 Apache Parquet and Arrow

Apache Parquet is the industry standard columnar storage format for analytical workloads. It provides advanced encoding schemes for efficient data compression, advanced data skipping capabilities, and extensive ecosystem support across analytics platforms. These features make it particularly well-suited for cloud-native systems as an open direct-access format among different analytical usages.

To work with Parquet data, query engines must first transcode it into an in-memory format, e.g., Apache Arrow. While Arrow was initially designed to enable zero-copy data sharing between processes, it has become the predominant in-memory representation format for analytical processing, offering optimized layouts for vectorized execution and standardized in-memory data exchange.

2.4 Filter pushdown on Parquet

Evaluating filters against a Parquet file involves four steps: 1. Decode Parquet metadata to locate relevant data pages needed for filter evaluation, 2. Decompress the data pages using general-purpose algorithms like LZ4 or Zstd, 3. Decode the Parquet-encoded columnar data into an in-memory representation, 4. Evaluate the filter predicates against the decoded in-memory data.

As shown in Figure 1, contrary to common assumptions, filter evaluation represents only a small fraction of the total processing time. The dominant cost comes from transcoding Parquet data into in-memory formats (e.g., Arrow) – a CPU-intensive operation that has been extensively optimized [79] and must be performed regardless of the filter predicates. This transcoding overhead is unavoidable since filters cannot be evaluated directly on Parquet's compressed format. Consequently, many cloud storage systems either do not support filter pushdown on Parquet files [8, 21, 55] or see degraded performance when attempting it [77].

2.5 Target workloads: low latency analytics

LiquidCache targets low-latency workloads including dashboards, anomaly detection pipelines, interactive queries, and LLM knowledge retrieval that demand sub-second response time. User studies show that response times above roughly 500 ms significantly hinder interactive exploration [50]. Unfortunately, fetching data from object storage already incurs more than 100 ms of first-byte latency [27], and metadata reads for formats such as Parquet often require two network round-trips before execution can even begin. Large-scale deployments of systems like Dremel [53] and Presto [69] report orders of magnitude slowdown when data is accessed over the network. These observations motivate caching techniques that minimize remote I/O and decoding overhead for such workloads.

3 LIQUIDCACHE ARCHITECTURE

Recent research has shifted toward memory disaggregation from compute servers [24, 33, 44, 52, 62, 64], enabling independent scaling of memory and compute resources. Disaggregated caching implements this vision by separating memory-intensive caching from compute-intensive processing using commodity hardware.

In this architecture, both cache and compute nodes run on commodity elastic compute servers but with different hardware configurations optimized for their roles. Cache servers are provisioned with high memory and modest CPU resources, while compute servers have high CPU and modest memory allocations. Connected through modern high-speed networks, this design allows multiple compute servers to share the same cache server, with each component scaling independently based on workload demands.

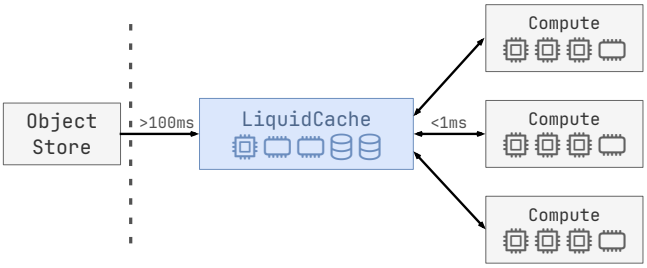

Figure 2: LiquidCache architecture – LiquidCache runs on a commodity elastic compute server, and is physically close to the compute servers that handle query execution.

Network bandwidth between cache and compute nodes is often a critical bottleneck. Filter pushdown addresses this by evaluating predicates at the cache server, reducing data transmission to compute nodes. These "disaggregated pushdown caches" filter data close to storage, significantly reducing network traffic and improving query performance.

This section presents the detailed design of LiquidCache, a novel disaggregated pushdown cache. We first outline the core system components and their interactions, then walk through the complete lifecycle of query execution from initial planning to final results, lastly discuss the caching mechanisms and policies.

3.1 System components

As shown in Figure 2, LiquidCache is a cache server that sits between compute nodes and object storage. Object storage is slow, with first-byte latencies over 100ms [27]. In contrast, LiquidCache's cache hits take less than 1 ms because the cache server runs physically close to the compute nodes.

The cache server reads from any object storage provider. While some providers allow pushdown filters to object storage, they either do not support Parquet [55] or are slower with pushdown enabled [77]. LiquidCache instead uses common object storage APIs and supports any storage provider.

The cache server provisions SSD-based elastic storage to store Parquet data retrieved from object storage. The cache server can seamlessly provision more elastic storage without downtime when additional storage capacity is required. The bundled CPU on the cache server manages network communication between compute nodes and object storage, and evaluates filters received from compute nodes. However, unlike compute nodes performing computationally intensive operations, the cache server's CPU requirements are minimal and cost-efficient.

Communication between the cache and compute servers occurs via Arrow Flight, a high-performance network protocol built on gRPC. Arrow Flight enables zero-copy data transfer by allowing compute nodes to directly interpret data from network buffers without de/serialization – a major cost in data-intensive systems [78].

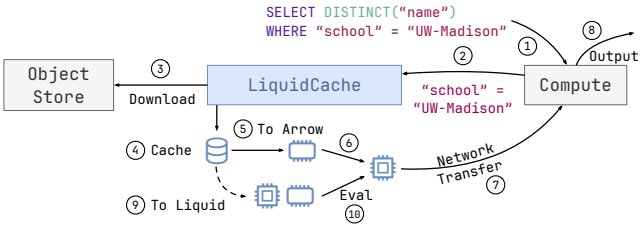

Figure 3: The life of a query in LiquidCache– Step 9 transcodes the Parquet into Liquid format in the background, allowing efficient predicate evaluation on a cache hit.

Compute nodes send SQL queries containing pushdown filters to the cache, which evaluates the filters and returns matching results.

The compute server functions like any standard data analytical server. LiquidCache provides a data connector (TableProvider) that users can easily swap in place of their existing connector. This connector serves as a bridge between object storage and LiquidCache's caching system. Behind the scenes, the connector determines which portions of the query plan should run on the compute server and which should be offloaded to the cache server. Once the data is received from the cache, it performs necessary data transformations to convert the data into a shape that the rest of the operators expect. With the disaggregated cache, the compute server is fully stateless and can be deployed as a standard VM or a serverless function.

3.2 Life of a query

With a disaggregated cache, the cache server and compute server cooperate to execute a query. As in data lake architecture [13], the catalog server sends a pair of SQL queries and file locations (on object storage) to the compute server. A conventional compute server will download the data from object storage and process the query on its own hardware. This section discusses the life of a query of LiquidCache, as shown in Figure 3, starting from a cold cache.

1. Get schema. When the compute server receives a query and file location, it needs to add the file's schema to its catalog. While traditional systems read this directly from object storage, LiquidCache reads the schema from the cache server. If not already cached, the cache server downloads Parquet metadata from object storage and then returns the table schema to the compute server.

2. Query planning. After obtaining the schema, the compute server generates and optimizes a query plan, pushing filter operators down to the scanning operator. LiquidCache employs standard filter pushdown strategies without introducing new rules. As shown in Figure 1 and discussed in Section 2.4, data decoding consumes over 90% of CPU time during filter operations, while filter evaluation takes less than 10%. Given these findings, LiquidCache prioritizes optimizing decoding rather than developing complex filter strategies.

3. Split the query. The compute server then splits the query plan into the data scanning operator and the remaining operators. In Figure 3, the data scanning operator corresponds to scanning the table with the filter "`school = UW-Madison`", and the compute operators correspond to the "`DISTINCT`" aggregation. LiquidCache implements a query optimizer rule that decides which operators should be pushed down to the cache server, and it will encode the query plan into a protobuf message and send it to the cache server for execution.

4. Execute the query. The cache server processes the SQL query with an embedded query engine, first checking for liquid-encoded data in its in-memory cache. The Liquid format (Section 4) provides high compression ratios and efficient predicate evaluation. If not in memory, the cache server checks its local disk cache. As a last resort, it downloads the data from object storage. For data not already in Liquid format, the cache server submits a background task to transcode it, enabling efficient predicate evaluation on subsequent cache hits. Lastly, the cache server streams the filtered results to the compute server via Arrow Flight, processing and sending data batches in parallel.

5. Return results. Finally, the filtered data is streamed back to the compute server, which continues processing the remainder of the query plan. Step 9 in Figure 3 shows that the cache server concurrently transcodes newly fetched Parquet data into Liquid format so that subsequent queries can evaluate predicates without repeating the conversion.

3.3 What to pushdown?

Filters. The cache server evaluates filters referencing only columns from the same table and containing no expensive UDF.

Projections. Column projection is always pushed down because it is virtually free when scanning Parquet data.

Aggregations. LiquidCache pushes down inexpensive aggregations such as `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`. Expensive operations like `DISTINCT` remain on the compute node to avoid building large in-memory hash tables on the cache server.

Cost-based pushdown. Currently, LiquidCache always pushes down the operators above, and more sophisticated cost-based policies [51, 77] are complementary and left for future work.

3.4 Local mode

Pushing down non-selective filters may increase network traffic because filtered data is transferred in an uncompressed format. LiquidCache therefore supports an optional "local mode" in which selected tables are cached in Liquid format directly on the compute node. In our current prototype, the user specifies which tables remain local; integrating automatic selection based on table statistics is left for future work.

3.5 Cache mechanisms

Cache Parquet bytes on cache-local disk. On cache misses, the server downloads the requested data range from object storage based on the query and Parquet metadata. It merges multiple small ranges into larger ones to minimize object store requests, which are billed per request rather than by volume [27]. Users can prepopulate the cache to avoid cold start delays. When elastic storage fills up, the cache server either evicts the least recently used files or expands storage without downtime. Disk space management is outside this paper's scope, practical deployments can choose any caching/eviction policy, or simply provision more elastic storage.

Cache Liquid data in memory. LiquidCache caches each column in batches of 8192 rows so that queries load only the required

pieces. Entries are identified by file name, row group, column index, and row number. A column-level LRU policy exploits the common pattern that batches within a column are accessed together while different columns are independent. Evicted batches are written to disk using a FlatBuffers [29]-style representation to avoid future deserialization overhead.

Data caching vs. result caching. LiquidCache caches data rather than final/intermediate query results. Caching Liquid-encoded data allows diverse filters to reuse the same cached blocks while avoiding repeated Parquet decoding. Result caching could skip filter evaluation, but would benefit only identical queries.

Handling new and deleted data. In a typical data-lake deployment [13], a catalog server orchestrates ingest and compaction. LiquidCache caches at the granularity of columns within a file, so when a file is removed from the catalog so is the corresponding liquid column/file. This approach keeps the cache consistent without a complex invalidation protocol.

4 LIQUID FORMAT

Efficient filter evaluation requires storing data in a format optimized for filtering operations. Simply caching Parquet data is insufficient, as it prioritizes compression ratios and sequential scan performance over filter evaluation efficiency. Instead, LiquidCache takes a novel approach: it caches the logical data rather than its physical representation. This is achieved by actively interpreting data from object storage and transcoding it into specialized physical representations optimized for filtering operations.

4.1 Dilemma: Decoding speed vs. memory usage

Decoding consumes significant CPU cycles [38], while DRAM already accounts for nearly half of data-center cost [44]. Highly compressed formats like Parquet decode slowly but use little memory, whereas in-memory layouts such as Arrow decode instantly but quadruple space usage. LiquidCache resolves this tension by co-designing its encoding with the filter evaluation process. All encodings permit independent decoding of each element so that values that fail early filters are never materialized. Techniques such as dictionary encoding and bit packing are combined in a cascade so that only the minimal representation necessary for the current filter stage is produced (Section 4.3).

4.2 Liquid data representation

LiquidCache combines several state-of-the-art data encoding techniques for high compression ratios and efficient filter evaluation. These include FSST [17] for string compression, FastLanes [1] for integer bit-packing, and standard encoding techniques like dictionary encoding and FoR (frame-of-reference) encoding.

This subsection presents LiquidCache's current encoding strategy. While we focus on string and integer encodings, LiquidCache can incorporate any appropriate columnar encoding technique that preserves independent element decodability, such as those in [38]. The system's flexible architecture allows dynamic updates to its encoding chain as new techniques emerge.

4.2.1 Encode strings. As shown in Figure 4, LiquidCache encodes string arrays in three cascading steps: first use dictionary encoding to deduplicate the strings, then use bit-packing to compress the keys, and finally use FSST encoding to compress the dictionary values.

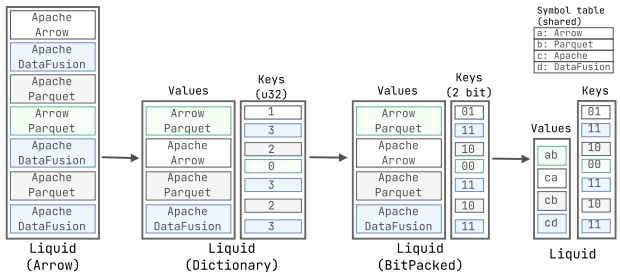

Figure 4: How LiquidCache encodes string arrays – The leftmost diagram shows the Arrow string representation, it will first be converted to dictionary encoding to eliminate the repeated strings, and then the keys are compressed with bit-packing based on the dictionary size, finally the dictionary is compressed with FSST encoding. Each of the intermediate representations can be used for filter evaluation. The corresponding data size and transcoding time can be found in Section 5.7.

The key array consists of unsigned integers that reference entries in the dictionary. The bit width needed to encode these keys is determined by the dictionary size – specifically, $\lceil log_2(n) \rceil$ bits where n is the number of unique strings. For example, with 10 unique strings, each key requires $\lceil log_2(10) \rceil = 4$ bits. LiquidCache leverages FastLanes [1] encoding to efficiently bit-pack these keys using the minimal required bit width.

For dictionary value compression, LiquidCache employs FSST encoding [17], which decomposes long strings into shorter substrings and maintains a symbol table mapping these substrings to compact codes. FSST compression's effectiveness heavily depends on its symbol table's quality – optimal compression is achieved when the symbol table is trained on the target data. However, constructing symbol tables is expensive, and the table itself consumes storage space, creating a trade-off between compression ratio and overhead. LiquidCache balances this trade-off by leveraging Parquet's structure. It builds one FSST symbol table per column chunk using the dictionary page (typically the first page) as training data. This table is reused to compress all string arrays within that chunk, amortizing construction cost while maintaining good compression by training on representative data.

4.2.2 Encode integers. For integer encoding, LiquidCache combines Frame-of-Reference (FoR) and bit-packing techniques. For each integer array, LiquidCache first determines the minimum and maximum values. The array is normalized by subtracting the minimum value from each element to contain only non-negative integers. This transformation is particularly beneficial for negative numbers in 2's complement representation. The normalized values are then bit-packed using FastLanes encoding, similar to the dictionary key encoding described earlier, except that the integer bit width is calculated based on the range of values (max-min) rather than the dictionary size.

4.2.3 Encode floating numbers. For floating-point columns, LiquidCache adopts the ALP scheme [2]. When values are whole numbers,

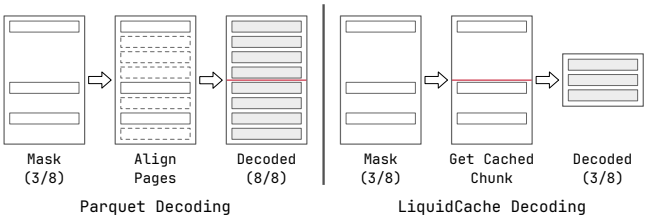

Figure 5: LiquidCache's selective decoding compared to Parquet's decoding – Parquet (left) has to decode an entire page even if just one element is needed. LiquidCache (right) allows each element to be decoded independently.

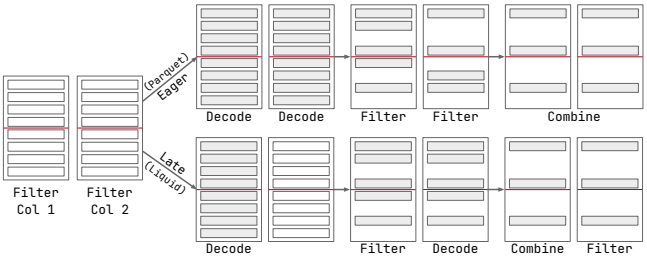

Figure 6: LiquidCache's late filter materialization compared to Parquet's eager materialization – The first row shows the filter evaluation process of Parquet, and the second row shows LiquidCache. With two filter columns, Parquet eagerly materialize both columns, while LiquidCache only materializes the first column and use the filter mask to selectively decode the second column.

they are converted to integers using PseudoDecimal [38]; otherwise ALP applies vectorized compression to the significant bits.

4.2.4 Lossy encoding. Lossy schemes [48, 59, 68] can further reduce size when applications tolerate small errors. Because LiquidCache caches the logical equivalent of Parquet files, we currently store only lossless encodings. Users may, however, apply lossy compression when generating the Parquet data itself.

LiquidCache encodes data at batch-size level (default to 8192). This fine-grained encoding strategy enables better compression ratios, as smaller integer arrays typically exhibit narrower value ranges requiring fewer bits for encoding. LiquidCache adapts encoding parameters based on actual data characteristics rather than pre-defined schema-level decisions, effectively decoupling logical data representation from physical storage format.

4.3 Co-design with filter evaluation

The guiding principle is to avoid decoding work whenever possible: (1) decode late by applying filters before materializing data, and (2) decode light by evaluating predicates on encoded or partially decoded values.

4.3.1 Selective decoding. Filter pushdown has two stages: (1) build filter mask over predicate columns, and (2) build output over projection columns. Selectively decoding applies the filter mask from stage 1 to selectively decode the projection columns where the corresponding bit in the filter mask is true [67]. While selective decoding has been applied to Parquet in many systems [18, 41], its benefits are limited by Parquet's page-based compression scheme. As illustrated in Figure 5, even when the filter mask selects 3 out of 8 elements, a Parquet decoder must decode the entire page containing all 8 elements since it cannot decode individual elements within a page. Therefore, selective decoding only benefits Parquet in cases where entire pages can be skipped. In contrast, LiquidCache pushes the selective decoding to its extreme, where an element is decoded if and only if it passes the previous filters.

4.3.2 Late filter materialization. Late filter materialization takes one step further by applying selective decoding to the filter evaluation itself (i.e., stage 1). It targets scenarios with multiple chained filters across different columns – a pattern commonly found in analytical queries. Figure 6 illustrates late filter materialization in action. In the eager approach without late filter materialization (first row), the system first decodes all filter columns, then evaluates predicates on each column independently, and finally combines the resulting boolean masks to determine which rows satisfy all conditions. With late filter materialization enabled (second row), the system processes columns sequentially – it decodes the first filter column, evaluates its filter to produce a boolean mask, Then, this mask is used to selectively decode only the qualifying elements from the second filter column. This cascading process means each subsequent column needs to decode progressively fewer elements as more filters are applied. Late filter materialization reduces both decoding work and filter evaluations by processing only data that passed previous filters. LiquidCache leverages bit deposit operations [46] to accelerate the bit mask manipulations required for late filter materialization.

4.3.3 Evaluate filters on encoded data. Filter operations can often be performed directly on encoded data, eliminating decoding when filter evaluation is encoding-aware. LiquidCache's string encoding preserves properties that enable these optimizations. For equality predicates on string arrays, rather than decoding to find matches, LiquidCache can encode the search target and perform comparisons directly on the encoded representations, significantly improving efficiency. Since LiquidCache uses dictionary encoding as the outermost layer for string arrays, the filter evaluation can operate solely on dictionary values, eliminating the need to decode the values array while also reducing the number of comparisons because the value array of the dictionary has unique values.

4.3.4 Evaluate filters on partially encoded data. Not all filter operations can be evaluated on encoded data. Substring pattern matching, for instance, requires access to the fully decoded strings. However, LiquidCache's cascading encoding strategy (Section 4.2) allows decoding to proceed only as far as necessary for each filter type. For example, decoding stops after dictionary values are available for substring searches, avoiding the overhead of key decoding and dictionary materialization. For prefix matching, only the initial bytes of each string need to be decoded. In addition to file evaluation, LiquidCache also transmits the partially encoded data (when applicable) over the network to reduce the network traffic.

Engineering challenges. Filter evaluation happens in the query engine, while data decoding occurs in format-specific readers. This separation creates a dependency challenge: readers need knowledge of filter expressions to optimize decoding, which would need to create reverse dependencies to the query engine (beyond the forward dependency where query engines rely on readers). LiquidCache avoids this complexity by unifying data representation and filter evaluation in a single layer. By embedding an extensible query engine in the cache server that supports all filter expressions, LiquidCache can optimize decoding strategies specifically for each filter type.

4.4 Efficient transcoding

When LiquidCache reads Parquet files from object storage, it must transcode them into Liquid format to leverage the abovementioned optimizations. While transcoding between formats incurs overhead, LiquidCache employs several key techniques to make this process highly efficient. This section examines three critical aspects of LiquidCache's transcoding design: (1) on-demand fine-grained transcoding that only converts data needed by queries, (2) background transcoding that hides conversion costs, and (3) deep integration with Parquet's reading pipeline to minimize overhead.

4.4.1 On demand fine-grained transcoding. Transcoding data between formats is a well-established practice, commonly seen in ETL (Extract-Transform-Load) pipelines that transform data between formats through a dedicated transcoding service. Traditional full-file transcoding is inefficient since, in most analytical workloads, only a few columns are frequently accessed.

LiquidCache takes a more fine-grained approach by transcoding data on-demand at column granularity. Only those columns are transcoded into liquid format when a query requests specific columns. This selective transcoding is further optimized through predicate pushdown – LiquidCache only transcodes data batches that pass filter predicates, avoiding unnecessary work on rows that will be filtered out. While transcoding does incur an upfront cost, it is a one-time investment that benefits all subsequent queries accessing the same data. This aligns well with typical analytical workload patterns where queries are highly repetitive [72], allowing the transcoding cost to be amortized across many query executions.

4.4.2 Background transcoding. Rather than blocking query processing while transcoding data from Arrow to Liquid format, LiquidCache performs this conversion asynchronously in the background. By intelligently scheduling transcoding during periods of lower CPU utilization, LiquidCache maximizes resource efficiency while minimizing impact on query performance. This background transcoding approach takes advantage of several common patterns observed in analytical workloads:

First, query execution frequently involves compute-intensive operators like joins and aggregations, where compute nodes spend significant time processing each batch of data before requesting the next batch from the cache server. During these natural processing gaps, LiquidCache can efficiently transcode data in the background without impacting query latency.

Second, when the cache server experiences a miss and must fetch data from object storage, the orders-of-magnitude slower object store I/O allows the cache server to transcode previously fetched batches while waiting for new data. This effectively hides the transcoding cost behind unavoidable I/O latency.

Third, analytical systems typically exhibit spiky workload patterns [72] requiring CPU over-provisioning to handle peak loads. LiquidCache exploits these quieter periods to perform transcoding work efficiently using otherwise idle CPU cycles.

4.4.3 Deep integration with Parquet reading. In addition to the design optimizations discussed above, LiquidCache also employs several engineering optimizations that directly optimize the transcoding process. LiquidCache rewrites the core of the Parquet reader such that the decoding process is aware of a later transcoding. For example, LiquidCache implements a new Arrow string representation called StringView, which allows LiquidCache to reuse the Parquet decoding buffer as the string buffer for the dictionary encoding, which saves multiple memory copies for large string arrays. Our StringView implementation has been upstreamed to Parquet and is now the default string representation for DataFusion [31, 32]. As another example, LiquidCache reworked the Parquet decoding pipeline such that it can reuse the decompression buffer for both predicate evaluation and building output data – saving one decompression step for the output data.

4.5 Discussion

Comparison with other modern data formats. Modern file formats such as BtrBlocks [38], LanceDB [42], Nimble [54], and Vortex [66] address some of the problems that LiquidCache tries to address. These formats, like LiquidCache, combine various encodings and compression algorithms, but require data sources to adopt their specific format, breaking the ecosystem that Parquet took years to build. Even minor updates to Parquet itself have historically taken years to be accepted and deployed across data systems. A complete rewrite of the file format is likely slow to adopt and fails to capture the rapid evolution of data systems. LiquidCache instead focuses on efficient and non-intrusive transcoding, allowing easy adoption by existing systems. Unlike existing formats focused on standalone design, LiquidCache takes a holistic approach by co-designing data representation with filter evaluation, delivering superior end-to-end performance.

Ephemeral format vs persistent format. Rather than creating another fixed file format that will eventually become outdated, LiquidCache intentionally remains ephemeral and adaptable (hence "Liquid"), eliminating the need for a stable specification that would require agreed-upon changes in the ecosystem. This leverages the unique position of Liquid's data – only in the cache, and the cache server is the only producer and consumer of Liquid data, allowing LiquidCache to freely add or remove encodings without any impact on the rest of the ecosystem. This approach is particularly valuable given the rapid evolution of data systems over the past decade, where new encoding/compression algorithms emerge every year while ecosystems remain locked into an older format for compatibility reasons [37].

5 EVALUATION

Our evaluation answers the following questions:

- How does LiquidCache's performance compare to state-of-the-art caching designs in latency, network usage, CPU time, and memory usage?
- How does Liquid format compare to established formats like Arrow and Parquet?
- What overhead does Liquid transcoding introduce, and can LiquidCache mask it from query latency?
- How well does LiquidCache handle diverse real-world analytical workloads?

5.1 Implementation details

We implement LiquidCache on Apache DataFusion [41], a high-performance analytical engine consistently ranking among top performers for Parquet workloads [40]. Our implementation consists of approximately 22k lines of Rust code, with an additional 5k lines contributed to upstream DataFusion, Arrow, and Parquet. LiquidCache implements a physical optimizer rule that replaces the Parquet reader with LiquidCache's reader, enabling integration with existing systems [11, 30, 35, 39, 42, 71] through minimal code changes – typically under 10 lines.

5.2 Evaluation setup

We evaluate LiquidCache using ClickBench [19, 20], an industry-standard analytical benchmark with 15GB (100M rows) of real-world web analytics data. This benchmark includes many short, selective queries typical of low-latency workloads. We focus on queries with complex filter patterns and variable-length fields [61].

Table 1 details characteristics of our selected query subset: query ID, columns projected, data size processed, filters applied, selectivity (percentage of rows passing filters), and data types. The queries span diverse scenarios from single-column filters to complex multi-column predicates, with selectivities ranging from highly selective (<0.01%) to broad (13.2%), and data sizes from 0.2GB to 14.8GB.

Experiments run on CloudLab [25] 6525 machines with 16 cores (32 threads) x86_64, 128GB RAM and SATA SSD, using 10Gbps network to simulate typical cloud environments. We disabled TLS encryption and used the default Arrow Flight without compression to minimize overhead. Each query was executed five times, with results averaged across the final three runs for warm-up. All latency measurements represent end-to-end execution time from SQL parsing to result retrieval. Unless otherwise specified, we report LiquidCache benchmarks on fully transcoded format.

ID	# Cols	Size	# Filters	Selectivity	Data Types
Q10	2	0.3 GB	1	2.0%	String, Int
Q19	1	0.2 GB	1	<0.01%	Int
Q20	0 (opt.)	2.7 GB	1	<0.01%	String
Q21	2	3.0 GB	2	<0.01%	String, Int
Q22	4	5.7 GB	3	0.02%	String, Int
Q23	104	14.8 GB	1	<0.01%	String, Int
Q31	5	1.5 GB	1	13.2%	String, Int

Table 1: Characteristics of selected ClickBench queries – Each query is described by its ID, number of columns projected, total data size processed, number of filters applied, selectivity (percentage of rows that pass the filters), and the data types involved. Q20 originally projects all 104 columns but is optimized to a simple count after projection and aggregation pushdown.

5.3 Baseline implementations

We implement three representative baselines matching the caching architecture in Section 2.1, representing key industry approaches: file serving, Parquet filter pushdown, and Arrow filter pushdown.

LiquidCache: Our proposed cache system combines Parquet's memory efficiency with Arrow's performance through its novel Liquid encoding format, transcoding data on-the-fly to a representation tailored for filter evaluation.

Arrow (pushdown): A filter pushdown cache storing data in Arrow format rather than Parquet, eliminating Parquet decoding overhead. It uses embedded DataFusion for filter evaluation, providing fast data access at a higher memory cost than Parquet-based approaches.

Parquet (pushdown): A filter pushdown cache evaluating predicates directly on cached Parquet files before network transfer. Like Arrow (pushdown), it uses embedded DataFusion for filter evaluation, transferring matching records to the compute engine via Arrow Flight. While reducing memory usage significantly, it incurs CPU overhead from filter pushdown on Parquet.

Parquet (file server): The simplest disaggregated caching form, serving Parquet files directly from a static file server in the same cluster. This widely adopted approach requires only a basic HTTP server with range request support. Despite easy deployment, it lacks filter pushdown capabilities, necessitating full-column chunk transfers regardless of query selectivity.

Other approaches exist between Arrow (pushdown) and Parquet (pushdown), such as pushdown systems caching uncompressed (or lightly compressed) Parquet data [13]. These systems fall between the two extremes regarding memory consumption and compute overhead and are not included here for space reasons.

5.4 Overall results

Our first experiment (Figure 7) compares LiquidCache with the baselines on four important metrics: latency, network traffic, cache CPU time, and cache memory usage. The x-axis shows the query id as described in Table 1, and the y-axis shows the corresponding metrics; lower is better.

5.4.1 Latency. Figure 7(a) shows Arrow (pushdown) and LiquidCache perform similarly, both significantly outperforming Parquet (pushdown) and Parquet (file server). Parquet (file server) consistently shows worst performance, with latencies up to 13 seconds. Parquet (pushdown) performs better but still reaches 2 seconds, while LiquidCache and Arrow (pushdown) never exceed 0.5 seconds. LiquidCache achieves latency comparable to Arrow (pushdown) on most queries but shows higher latency on Q20 and Q23, which involve filters on large string columns requiring LiquidCache to decode data before applying filters, while Arrow (pushdown) applies filters directly on in-memory Arrow data.

5.4.2 Network traffic. Figure 7(b) shows corresponding network traffic on a logarithmic scale (lower is better).

The three pushdown-enabled systems (LiquidCache, Arrow (pushdown), and Parquet (pushdown)) exhibit similar network traffic patterns since they transfer only records matching filter predicates. LiquidCache achieves slightly lower traffic by transferring partially-encoded data (e.g., dictionary-encoded Arrow) over the network. In

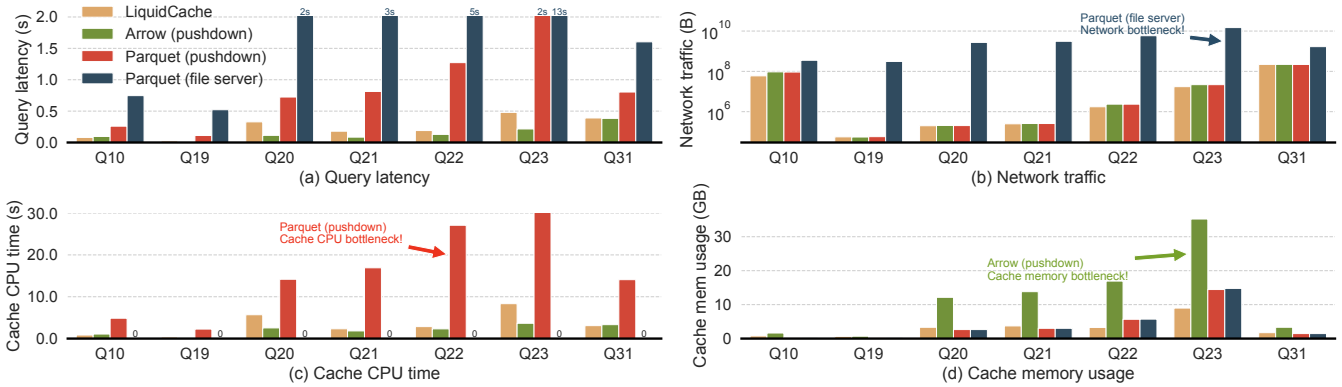

Figure 7: ClickBench results comparing LiquidCache with three baselines (lower is better). Parquet (file server) is limited by network traffic, Parquet (pushdown) by CPU usage, and Arrow (pushdown) by memory footprint; LiquidCache avoids all three bottlenecks.

contrast, Parquet (file server) transfers orders of magnitude more data by sending entire unfiltered Parquet chunks. This gap is particularly pronounced for highly selective queries (Q19-Q23), where pushdown systems transfer only small fractions of records. The difference narrows for less selective queries (Q10, Q31), where most data passes filters. Excessive network traffic saturates bandwidth and incurs substantial CPU overhead in cache and compute nodes' network stacks for packet processing.

5.4.3 CPU time on cache server. Cache servers with limited CPU power struggle with computationally intensive Parquet decoding, as noted in Section 2.4. We instrumented our implementation to measure CPU time consumed by pushdown operations, including decompression/decoding and filter/aggregation evaluation.

Figure 7(c) compares the total CPU core time consumed by each system on the cache server. Parquet (pushdown) spends significantly more CPU time than other baselines since it must fully decode Parquet data on the cache server. In contrast, Parquet (file server) uses minimal CPU, simply serving files without processing. Among pushdown-supporting systems, Arrow (pushdown) is most efficient since it operates directly on in-memory data without decoding. LiquidCache achieves comparable CPU efficiency to Arrow (pushdown) despite working with compressed data. This efficiency stems from Liquid's faster decoding scheme (Figure 10) and LiquidCache's ability to skip unnecessary decoding through tight filter pushdown integration (Figure 13).

5.4.4 Cache memory consumption. We analyze cache memory usage across different queries. For fair comparison, each experiment begins with an empty cache and reports memory used for caching, excluding runtime data structures.

As shown in Figure 7(d), Arrow (pushdown) has the highest memory usage, while LiquidCache maintains a memory footprint comparable to Parquet. For Query 23 alone, Arrow consumes more than 30 GB of cache memory. Parquet variants (pushdown and file server) show identical memory usage as they cache the same compressed Parquet data. LiquidCache achieves comparable or better memory efficiency than Parquet, thanks to its advanced cascading encoding scheme. Despite its high compression ratio, LiquidCache preserves random access capabilities to individual data elements, as demonstrated in Figure 10.

5.5 TPC-H results

Although LiquidCache targets short, selective queries, we also benchmark TPC-H to evaluate larger-than-memory workloads. We run the scan-heavy queries (Q4,6,11,12,14,15,16,20) at scale factor 100, totaling 100 GB data, with only 24GB max system memory, including 8GB cache and 16GB for runtime data structures (e.g., join hash tables). We additionally compare to Apache ORC [10]. Converting the dataset using the same Snappy compression yields a slightly better ratio for ORC (32%) than Parquet (34%), confirming prior observations [49, 79].

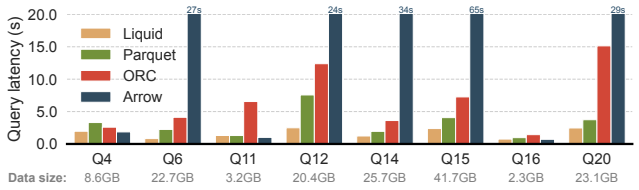

Figure 8: TPC-H SF100 (100GB data) with 8GB cache. Data size indicates query scan size, excluding runtime structures. Arrow is best when data fits in cache and worst otherwise. Parquet outperforms ORC due to better query engine integration. Liquid consistently delivers the best performance when the input does not fit in the cache.

Figure 8 shows the TPC-H SF100 results benchmarked with 8GB data cache, the y-axis shows the query latency, and data sizes indicate the corresponding query's data size, excluding runtime structures. Queries have varying data sizes, from 2.3GB to 41.7GB. When the data size fits in cache, Arrow performs the best (similar to Liquid) and performs the worst otherwise. Although Parquet and ORC have similar compression ratio, Parquet generally outperforms ORC by a large margin due to its better query engine integration – both DataFusion and DuckDB best support Parquet while the former has minimal ORC support and the latter has no ORC support

– highlighting the importance of query engine and data format co-design. Liquid consistently performs the best, thanks to its efficient encoding that reduces data size and filter-aware decoding that skips unnecessary decoding and file reads.

These results highlight a key limitation of Arrow (pushdown) for disaggregated caching: Its substantial memory requirements make it prohibitively expensive to deploy at scale on cache servers. In contrast, LiquidCache's memory footprint closely matches Parquet's efficiency while providing efficient filter evaluation, enabling a practical transition from Parquet-based to Liquid-based caching without increasing memory costs.

5.6 Decoding revisited

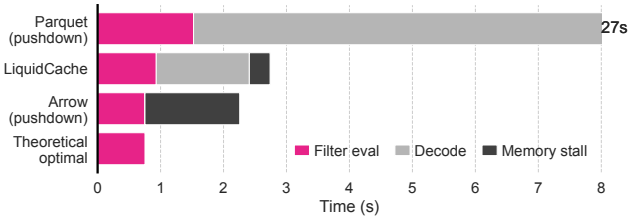

Figure 9: Decomposed CPU time spent on cache server for ClickBench Q22. LiquidCache spends similar total time as Arrow (pushdown), but primarily on decoding compressed data, while Arrow (pushdown) suffers from memory stalls due to scanning 4× more uncompressed data.

We revisit LiquidCache's decoding time and how it compares to the theoretical optimal. We sampled the cache server execution and categorized time into: filter evaluation, decoding, and memory stall. Filter evaluation represents the useful time spent evaluating the filter. Decoding is the time spent converting data to Arrow format for vectorized execution. Memory stall is time spent waiting for data to be ready for CPU execution, including cache misses, memory allocation/copying for filtering, etc.

As shown in Figure 9, LiquidCache significantly reduces overall CPU time from 27s to 2.7s, reaching a similar overall time as Arrow (pushdown). While LiquidCache and Arrow (pushdown) spend comparable total execution time, they exhibit markedly different performance characteristics. LiquidCache dedicates most processing time to efficiently decoding compressed data. Thanks to its filter-pushdown co-design (Section 5.9), it only decodes the subset of data that passes filters. In contrast, Arrow (pushdown) experiences significant memory stalls due to operating on uncompressed data that is 4× larger. Both systems achieve similar filter evaluation times, approaching the theoretical optimum – which would evaluate all filters directly on encoded data, eliminating decoding overhead while maintaining a compact memory footprint.

5.7 Transcoding cost

Instead of requiring all data sources to produce Liquid-encoded data, LiquidCache progressively transcodes upstream Parquet data into Liquid on the fly, facilitating integration into existing systems. This section analyzes direct transcoding costs among Liquid's different data representations, then demonstrates how LiquidCache

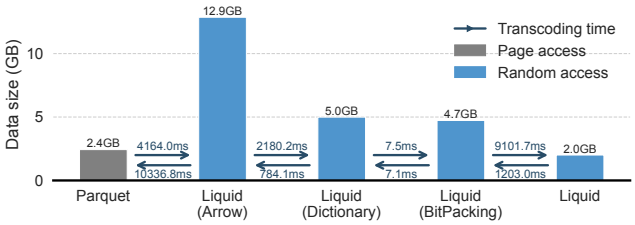

Figure 10: Memory usage and transcoding cost of "Title" column of the ClickBench – the largest column in the dataset. The bar height indicates the encoded data size, arrows denote transcoding time. Parquet only allows page-level access, while the rest of the encodings permit random element access.

effectively hides these costs from the critical query execution path through background processing.

Figure 10 presents a detailed analysis using the "Title" column from ClickBench – the dataset's largest column. The figure shows memory consumption (y-axis) for different encoding formats (x-axis), with arrows indicating transcoding times between formats. While Parquet restricts access to page-level granularity, all other encodings support random access to individual elements. Our evaluation reveals several key findings:

- Liquid's fully encoded format achieves compression ratios comparable to Parquet while preserving random access capabilities
- Encoding Liquid takes similar time as encoding Parquet, but decoding Liquid is over 2× faster than decoding Parquet.
- Dictionary encoding provides significant compression (>2×), matching the compression gains from Liquid (BitPacking) to full Liquid, but with 4× longer encoding time
- BitPacking provides a further 6% compression and encodes more than 300× faster than dictionary encoding

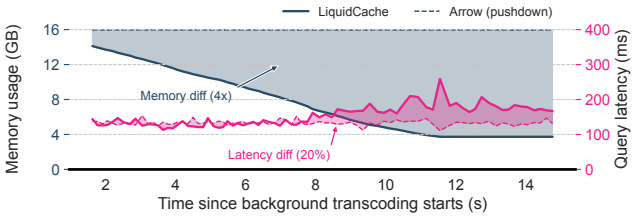

Figure 11: LiquidCache's latency and memory usage on Q21 – LiquidCache gradually transcodes Arrow into Liquid in background, avoiding latency spikes on the critical path of query processing.

We analyze how LiquidCache handles transcoding costs during query execution using Q21 from ClickBench as a case study. This query contains multiple complex filters and demonstrates typical transcoding patterns observed across other queries. We restrict background processing to only four threads to simulate realistic cache server conditions with limited CPU resources. Real systems can use all idle cores for transcoding when under-loaded – a typical pattern in cloud analytical systems [72].

Figure 11 illustrates the relationship between transcoding progress, memory usage, and query latency for both LiquidCache and Arrow (pushdown). The x-axis represents time elapsed since transcoding begins, while the left and right y-axes show memory consumption and query latency, respectively. The shaded area represents the difference between LiquidCache and Arrow (pushdown).

On cache misses, LiquidCache first decodes Parquet data into Arrow format, then queues Arrow-to-Liquid transcoding as a background task while immediately proceeding with query execution. As background transcoding progresses, memory usage steadily decreases, achieving a 4× reduction compared to Arrow format, while query latency increases by only 20%. As discussed in Section 4.4, LiquidCache's design ensures transcoding never blocks the critical query execution path, as queries can operate on any intermediate data without waiting for transcoding to complete. The near-identical initial query performance between LiquidCache and Arrow (pushdown) confirms this non-blocking behavior.

5.8 Cold run latency

We analyze how LiquidCache's transcoding process affects query latency, particularly during initial execution when transcoding occurs concurrently with query processing.

Figure 12 compares execution latency between LiquidCache and baseline Parquet across different storage devices. The left region displays cold run latency, while the right shows stabilized warm run latency. Storage devices range from remote (S3-far: cross-continent us-west to eu-central) to local (Memory: kernel page cache). Intermediate options include S3/S3-express (nearest region: Nevada to Oregon), MinIO (same cluster), and SSD (locally attached storage). The y-axis shows latency with a horizontal line indicating transcoding time. LiquidCache (blocking) disables background transcoding, sequentially processing one batch at a time.

For slow storage (non-memory devices), execution is IO-bounded, allowing LiquidCache to overlap IO stalls with CPU-intensive transcoding. Transcoding time remains negligible compared to IO time, sometimes even smaller than network variance. With background transcoding, LiquidCache achieves near-identical latency to direct Parquet reads.

Transcoding overhead becomes visible when storage throughput exceeds transcoding throughput. Reading from the kernel page cache shows transcoding time exceeding total execution time. However, for typical LiquidCache deployments with network-attached storage, IO time dominates transcoding overhead and wastes otherwise idling CPU.

5.9 Ablation study

We analyze how each decoding optimization described in Section 4.3 contributes to overall performance. Using the same experimental setup, we selectively enable optimizations one at a time to measure their impact.

"Liquid fully decoded" decodes entire arrays into Arrow format even when accessing single elements - a coarser granularity than Parquet's page-level decoding (Figure 13). It outperforms Parquet (pushdown) on single-filter queries like Q20, where full-column decoding is necessary because Liquid data decodes faster than Parquet (Figure 10). Selective decoding provides significant benefits

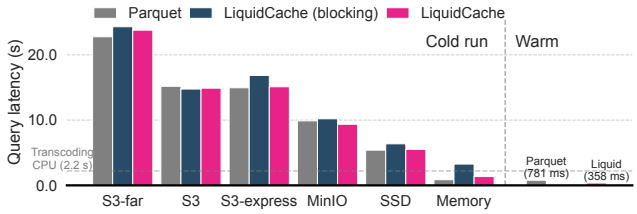

Figure 12: Impact of transcoding overhead. Baseline Parquet reads directly from files. LiquidCache (blocking) converts one batch at a time, whereas LiquidCache overlaps transcoding with I/O. Storage devices: S3-far (cross-continent), S3/S3-express (nearest region), MinIO (same cluster), SSD (local), Memory (page cache).

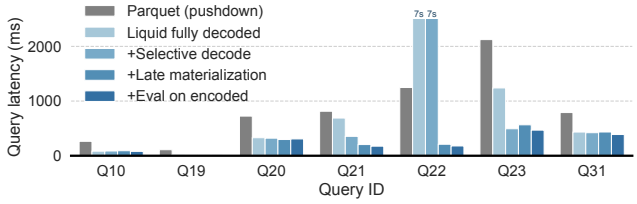

Figure 13: Ablation study of LiquidCache's decoding optimizations vs Parquet (pushdown) as the baseline.

when constructing output data using filter evaluation masks to decode only relevant data. This is particularly effective for queries with selective filters (e.g., Q21, Q23). However, even for selective queries like Q22, it sees limited benefit as they are bottlenecked by the filter evaluation phase, highlighting the importance of late filter materialization.

Parquet (pushdown)	Full decode	+Selective decode	+Late material.	+Eval on encoded
34.1s	19.3s	13.1s (-32%)	5.0s (-61%)	0.4 s (-91%)

Table 2: Ablation study based on a variation of Q22. Eval on encoded data shows more than 98% improvement over baseline Parquet (pushdown).

Filtering directly on encoded data requires filter expressions to be compatible with the encoding scheme (e.g., equality filters on strings). While this optimization shows modest benefits in our evaluation, we demonstrate its potential impact through an additional experiment on a variation of Q22 with compatible predicates on large columns. As shown in Table 2, this achieves an 11× improvement over late filter materialization alone, suggesting significant performance gains when applying this technique to larger columns.

5.10 Cache dynamics

We evaluate LiquidCache's cache reuse behavior across different filter patterns and cache management operations. Using a synthetic workload with identical string columns A and B, we measure latency from a cold cache with data stored on a local SSD.

Operation	Lat (ms)	Operation	Lat (ms)
1. A = Madison	5909	6. B = Madison	5648
2. A = Utah	10	7. **A like '%Utah%'** & B ≠ Utah	266
3. **A > Madison**	236	8. A = Utah	20
4. Flush cache to disk		9. Delete cache	
5. A ≠ Utah	374	10. A = Utah | B = Madison	10910

Table 3: LiquidCache latency following different operations. Operations are executed in the order of the table. The bolded filters can not be evaluated on encoded data and must be decoded first.

Table 3 demonstrates LiquidCache's cache behavior across different operations. Operation 1 represents a cache miss requiring SSD data loading and Liquid transcoding. Operation 2 hits cache despite different filters, enabling encoded data evaluation without decoding. Operation 3 introduces a new filter type requiring prior decoding. After disk flush, operation 5 loads data in Liquid format, allowing encoded evaluation. Operation 6 loads a new column with latency similar to operation 1. Operation 10 loads and transcodes both columns A and B following their deletion.

LiquidCache only transcodes once, and all filter operations can be applied on the transcoded data with efficient filter evaluation. New data can be loaded to the cache, and old transcoded data can be flushed to disk without losing the transcoding progress.

5.11 Cases for non-pushdown

Pushdown transfers filtered but uncompressed data over the network, which can exceed the compressed unfiltered data size when filters are non-selective. Theoretically, pushdown benefits require filter selectivity below the data compression ratio. While this condition typically holds, some workloads violate it. ClickBench Q27 demonstrates this case with 99% filter selectivity and 31% compression ratio, where LiquidCache performs 3× slower than Parquet (file server). Adaptive pushdown could address this by dynamically deciding filter pushdown based on cardinality estimation, which we leave for future work.

6 RELATED WORK

Data Lakehouse. Modern data analytics have shifted from on-premise to cloud-native Lakehouse architectures [12, 13, 56], where data is stored in object storage using open direct-access formats like Parquet rather than proprietary engine-specific formats. While this architectural shift enables greater flexibility and interoperability, it creates new challenges for modern query engines [15, 40, 58, 60] that must efficiently process remote Parquet data. LiquidCache is an important component in Lakehouse architecture that allows query engines to efficiently evaluate filters on Parquet data with low latency by sending the filtered data to the query engine, reducing CPU and network costs.

Modern encoding and columnar file formats. Since the initial release of Parquet in 2013 [73], many new encoding schemes [1, 2, 17, 38, 43, 45, 47] have been proposed to improve its compression and decoding efficiency. However, introducing these new encoding schemes to Parquet would break backward compatibility, and the ecosystem has effectively locked in [37] to a minimal set of encoding schemes that are well-supported by major query engines.

File formats like Vortex [66], Nimble [54], and BtrBlocks [38] have been proposed to replace Parquet, even Parquet itself is evolving to modernize [9]. However, adopting these new formats remains slow due to the same compatibility concerns. Learning from these lessons, LiquidCache takes a pragmatic approach by non-intrusively and progressively transcoding Parquet data into a format tailored for query engine needs, while maintaining compatibility with existing systems.

Resource disaggregation The disaggregation of compute and storage has proven highly successful in modern cloud platforms [6, 22, 35, 74]. Building on this success, researchers and industry have been exploring ways to disaggregate further compute and memory resources. Several systems like LegoOS [64], TPP [52], FaRM [24], AIFM [62], Pond [44], Redy [80], and Hao et al. [33] have investigated various software architectures for disaggregated memory systems. However, these approaches typically rely on specialized hardware such as RDMA or CXL to achieve the high network bandwidth required for effective memory disaggregation. LiquidCache takes a different approach. Rather than physically separating DRAM from compute resources, it implements a logical disaggregation of software components. By decoupling the memory-intensive cache from CPU-intensive compute, LiquidCache enables independent scaling of these components while maintaining high performance without requiring specialized hardware.

7 FUTURE WORK

LiquidCache is a first step toward a fully disaggregated caching architecture. Several extensions remain:

Substrait query plans. LiquidCache currently relies on DataFusion's internal plan format. Supporting the Substrait would allow Spark and DuckDB to share the same cache.

More efficient cache memory management. LiquidCache uses a simple LRU policy today, but prior work shows that conventional policies perform poorly for analytical workloads [14, 57].

Support for additional pushdown. Beyond projection, filtering, and simple aggregation, future versions could push down joins or employ sideways information passing to reduce traffic further.

Optimizing non-selective filters. For highly non-selective predicates, pushing filters down can increase network traffic. Future work could dynamically decide whether to push down filters based on cardinality estimates.

8 CONCLUSION

This paper presents LiquidCache, the first practical disaggregated cache system that achieves low network traffic, low CPU usage, and low memory consumption. LiquidCache is based on the key observation that data decoding – not filter evaluation – is the bottleneck in filter pushdown. LiquidCache addresses this issue by co-designing the Liquid format with filter pushdown semantics to reduce data decoding overhead dramatically. Rather than requiring existing systems to migrate their data, LiquidCache transparently and incrementally transcodes Parquet data into its optimized format. Our comprehensive evaluation on ClickBench demonstrates that LiquidCache delivers 10x lower CPU usage than state-of-the-art systems without increasing memory usage.

REFERENCES

[1] Azim Afroozeh and Peter Boncz. 2023. The fastlanes compression layout: Decoding> 100 billion integers per second with scalar code. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2132–2144.

[2] Azim Afroozeh, Leonardo X Kuffo, and Peter Boncz. 2023. Alp: Adaptive lossless floating-point compression. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

[3] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Zongchang Chen, Ming Dai, et al. 2021. Napa: Powering scalable data warehousing with robust query performance at Google. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2986–2997.

[4] Alluxio. 2024. *Alluxio - Data Orchestration for AI and Analytics.* Alluxio, Inc. https://www.alluxio.io A distributed cache platform that accelerates AI and analytics workloads by providing high-speed data access across different storage systems, offering up to 4x faster AI model training and 8 GB/s throughput per client.

[5] Amazon Web Services. 2024. Amazon ElastiCache for Valkey and for Redis OSS. https://aws.amazon.com/elasticache/redis/ Accessed: August 2024.

[6] Amazon Web Services. 2024. *Amazon Redshift - Cloud Data Warehouse.* Amazon Web Services, Inc. https://aws.amazon.com/redshift/ A cloud data warehouse service offering SQL analytics at scale with features including serverless computing, zero-ETL integration, and ML capabilities.

[7] Amazon Web Services. 2024. *Querying data in place with Amazon S3 Select.* Amazon Web Services. https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html Part of the Amazon Simple Storage Service (S3) User Guide.

[8] Amazon Web Services. 2024. Use S3 Select with Spark to improve query performance. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-s3select.html. Amazon EMR Release Guide.

[9] Apache Parquet. 2024. Page Index - Apache Parquet. https://parquet.apache.org/docs/file-format/pageindex/ Accessed: 2025-02-24.

[10] Apache Software Foundation. 2013. Apache ORC: Optimized Row Columnar file format for big data. https://orc.apache.org/. Accessed: 2025-06-10.

[11] Apache Software Foundation. 2025. Apache DataFusion Comet. https://github.com/apache/datafusion-comet A high-performance accelerator for Apache Spark built on Apache DataFusion.

[12] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[13] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, Vol. 8. 28.

[14] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. 2017. Recache: Reactive caching for fast analytics over heterogeneous data. (2017).

[15] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, et al. 2022. Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 International Conference on Management of Data*. 2326–2339.

[16] Michał Bodziony, Rafał Morawski, and Robert Wrembel. 2022. Evaluating pushdown on nosql data sources: experiments and analysis paper. In *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments*. 1–6.

[17] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.

[18] Boudewijn Braams. 2018. Predicate pushdown in parquet and Apache spark. *Ph. D. dissertation* (2018).

[19] ClickHouse. 2022. ClickBench: A Benchmark For Analytical Database Management Systems. https://benchmark.clickhouse.com. Website. Accessed: 2025-02-17.

[20] ClickHouse. 2022. ClickBench: A Benchmark for Analytical Databases. https://github.com/ClickHouse/ClickBench. GitHub repository. Accessed: 2025-02-17.

[21] Databricks. 2024. Amazon S3 Select. https://docs.databricks.com/aws/en/connect/external-systems/amazon-s3-select. Databricks Documentation.

[22] Databricks. 2024. *Optimize performance with caching on Databricks.* Databricks, Inc. https://docs.databricks.com/aws/en/optimizations/disk-cache Documentation describing Databricks disk caching feature for accelerating data reads by creating copies of remote Parquet data files in nodes' local storage.

[23] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. 2013. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1221–1230.

[24] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[25] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*. 1–14.

[26] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: a unified cache storage system for analytical databases. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2432–2444.

[27] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2769–2782.

[28] Kira Duwe, Angelos Anadiotis, Andrew Lamb, Lucas Lersch, Boaz Leskes, Daniel Ritter, and Pınar Tözün. [n.d.]. The Five-Minute Rule for the Cloud: Caching in Analytics Systems. ([n. d.]).

[29] Google and contributors. 2014. FlatBuffers: Memory Efficient Serialization Library. https://github.com/google/flatbuffers. Accessed: 2025-06-10.

[30] GreptimeTeam. 2025. GreptimeDB. https://github.com/GreptimeTeam/greptimedb An open-source, cloud-native, unified time series database for metrics, logs and events, supporting SQL/PromQL/Streaming.

[31] Xiangpeng Hao and Andrew Lamb. 2024. *Using StringView / German Style Strings to Make Queries Faster: Part 1- Reading Parquet.* Apache DataFusion. https://datafusion.apache.org/blog/2024/09/13/string-view-german-style-strings-part-1/ Apache DataFusion Blog.

[32] Xiangpeng Hao and Andrew Lamb. 2024. *Using StringView / German Style Strings to Make Queries Faster: Part 2 - String Operations.* Apache DataFusion. https://datafusion.apache.org/blog/2024/09/13/string-view-german-style-strings-part-2/ Apache DataFusion Blog.

[33] Xiangpeng Hao, Xinjing Zhou, Xiangyao Yu, and Michael Stonebraker. 2024. Towards Buffer Management with Tiered Main Memory. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

[34] Jiasheng Hu, Philip A Bernstein, Jialin Li, and Qizhen Zhang. 2024. DPDPU: Data Processing with DPUs. *arXiv preprint arXiv:2407.13658* (2024).

[35] InfluxData. 2025. InfluxDB. https://github.com/influxdata/influxdb Scalable datastore for metrics, events, and real-time analytics.

[36] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. 2021. Farview: Disaggregated memory with operator off-loading for database engines. *arXiv preprint arXiv:2106.07102* (2021).

[37] Laurens Kuiper. 2025. *Query Engines: Gatekeepers of the Parquet File Format.* DuckDB Foundation. https://duckdb.org/2025/01/22/parquet-encodings.html Accessed: 2025-02-17.

[38] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: efficient columnar compression for data lakes. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[39] LakeSail. 2025. Sail. https://github.com/lakehq/sail A computation framework to unify batch processing, stream processing, and compute-intensive (AI) workloads.

[40] Andrew Lamb. 2024. Apache DataFusion is now the fastest single node engine for querying Apache Parquet files. https://datafusion.apache.org/blog/2024/11/18/datafusion-fastest-single-node-parquet-clickbench/. Accessed: 2025-02-17.

[41] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data*. 5–17.

[42] LanceDB. 2025. LanceDB. https://github.com/lancedb/lancedb Developer-friendly, serverless vector database for AI applications.

[43] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. 2016. Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience* 46, 11 (2016), 1547–1569.

[44] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[45] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-Based Lossless Floating-Point Compression. *Proc. VLDB Endow.* 16, 7 (March 2023), 1763–1776. https://doi.org/10.14778/3587136.3587149

[46] Yinan Li, Jianan Lu, and Badrish Chandramouli. 2023. Selection Pushdown in Column Stores Using Bit Manipulation Instructions. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[47] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3058–3070.

[48] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. 2021. Decomposed bounded floats for fast compression and queries. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2586–2598.

[49] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A deep dive into common open formats for analytical dbmss. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3044–3056.

[50] Zhicheng Liu and Jeffrey Heer. 2014. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2122–2131.

[51] Jianan Lu, Ashwini Raina, Asaf Cidon, and Michael J Freedman. 2025. Fusion: An Analytics Object Store Optimized for Query Pushdown. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 540–556.

[52] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 742–755.

[53] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, et al. 2020. Dremel: A decade of interactive SQL analysis at web scale. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3461–3472.

[54] Meta. 2024. Nimble: A New File Format for Storage of Large Columnar Datasets. https://github.com/facebookincubator/nimble. Formerly known as "Alpha".

[55] Microsoft. 2024. *Azure Data Lake Storage query acceleration.* Microsoft. https://learn.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-query-acceleration

[56] Fatemeh Nargesian, Erkang Zhu, Renée J Miller, Ken Q Pu, and Patricia C Arocena. 2019. Data lake management: challenges and opportunities. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1986–1989.

[57] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. 2024. HPCache: memory-efficient OLAP through proportional caching revisited. *The VLDB Journal* 33, 6 (2024), 1775–1791.

[58] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3372–3384.

[59] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827.

[60] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.

[61] Adrian Riedl, Philipp Fent, Maximilian Bandle, and Thomas Neumann. 2023. Exploiting Code Generation for Efficient LIKE Pattern Matching.. In *VLDB Workshops*.

[62] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.

[63] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse-Lightning Fast Analytics for Everyone. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3731–3744.

[64] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.

[65] Malcolm Singh and Ben Leonhardi. 2011. Introduction to the IBM Netezza warehouse appliance. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*. 385–386.

[66] SpiralDB. 2024. Vortex: An extensible, state-of-the-art columnar file format. https://github.com/spiraldb/vortex. Accessed: 2024.

[67] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2018. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518.

[68] Zhaoyuan Su, Ammar Ahmed, Zirui Wang, Ali Anwar, and Yue Cheng. 2024. Everything You Always Wanted to Know About Storage Compressibility of Pre-Trained ML Models but Were Afraid to Ask. *arXiv preprint arXiv:2402.13429* (2024).

[69] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, et al. 2023. Presto: A decade of SQL analytics at Meta. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[70] Jacopo Tagliabue, Ryan Curtin, and Ciro Greco. 2024. FaaS and Furious: abstractions and differential caching for efficient data pre-processing. arXiv:2411.08203 [cs.DB] https://arxiv.org/abs/2411.08203

[71] Tonbo.io. 2025. Tonbo. https://github.com/tonbo-io/tonbo A portable embedded database using Apache Arrow.

[72] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3694–3706.

[73] Deepak Vohra and Deepak Vohra. 2016. Apache parquet. *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools* (2016), 325–335.

[74] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 449–462.

[75] Cong Yan, Yin Lin, and Yeye He. 2023. Predicate pushdown for data science pipelines. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–28.

[76] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid pushdown and caching in a cloud DBMS. *Proceedings of the VLDB Endowment* 14, 11 (2021).

[77] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2024. FlexpushdownDB: rethinking computation pushdown for cloud OLAP DBMSs. *The VLDB Journal* 33, 5 (2024), 1643–1670.

[78] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)*. 15–28.

[79] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An empirical evaluation of columnar storage formats. *Proceedings of the VLDB Endowment* 17, 2 (2023), 148–161.

[80] Qizhen Zhang, Philip A Bernstein, Daniel S Berger, and Badrish Chandramouli. 2021. Redy: Remote dynamic memory cache. *arXiv preprint arXiv:2112.12946* (2021).

[81] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. 2023. FoundationDB: A Distributed Key-Value Store. *Commun. ACM* 66, 6 (2023), 97–105.