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 CPU bottleneck in existing systems. To address this challenge, we propose caching logical data rather than its physical representation by transcoding upstream files on-the-fly into an optimized "Liquid" format. This format is co-designed with filter evaluation semantics to enable selective decoding, late materialization, and encoding-aware filter evaluation, delivering efficient filter evaluation while preserving compression benefits. The "Liquid" format exists exclusively in the cache, allowing easy adoption without changing existing storage formats, and enabling LiquidCache itself to evolve and incorporate future encodings while maintaining ecosystem compatibility. Through integration with Apache DataFusion and evaluation with ClickBench, we demonstrate that LiquidCache achieves up to 10× lower cache server CPU usage compared to state-of-the-art systems without increasing memory footprint.

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:
The source code, data, and/or other artifacts have been made available at https://github.com/XiangpengHao/liquid-cache.

1 INTRODUCTION

Cloud-native analytical systems [6, 8, 10, 12, 24, 27, 32, 36, 51, 57] employ compute-storage disaggregation. This architecture requires compute nodes to fetch data on demand from remote object stores, typically in Parquet format [34, 65] – the industry standard for analytical data. Despite disaggregation's benefits, reading from object stores incurs high access latency and per-request billing costs [25].

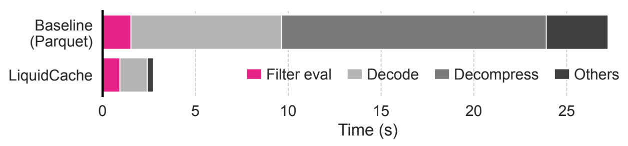

Figure 1: Time breakdown of filter pushdown for ClickBench Q22 – Data decoding and decompression consume over 90% of CPU time in baseline (Parquet), while filter evaluation takes less than 10%, contradicting the assumption that filter evaluation is the main bottleneck.

To mitigate these costs, industry employs disaggregated caching [4, 5, 20, 26, 60] with independently scalable shared cache servers. Yet this approach creates a critical challenge: analytical workloads generate substantial network traffic as cached data traverses the network to compute nodes. This network bottleneck ultimately limits the cache servers to primarily caching metadata rather than serving as effective data caches [3, 4, 24, 67].

In this paper, we propose a new caching approach that leverages filter pushdown [7, 14, 16, 21, 33, 45, 53, 61, 62] – a classic database optimization technique – to reduce network traffic in disaggregated caching. Analytical queries frequently include filters that can substantially reduce the data volume transferred. 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 often degrades performance for Parquet data [63]. Consequently, many cloud-native analytical systems [8, 19] have disabled filter pushdown for Parquet files, incurring prohibitively high network costs.

Contrary to common belief, our study reveals that the main bottleneck in filter pushdown is not filter evaluation but 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

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

and decompression consume over 90% of baseline (Parquet) processing time, while actual filter evaluation takes less than 10% – challenging the common assumption that filter evaluation itself is the main bottleneck. With Parquet being the predominant format for analytical 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 [38] – 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 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 [17, 18], comparing it with state-of-the-art systems. 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, 15, 35] 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 show that LiquidCache achieves 10x lower CPU usage than Parquet without increasing memory usage.

2 BACKGROUND

2.1 Cache for object storage

Cloud-native analytical systems leverage object storage for cost-effective durability and scalability. However, its high latency (>100 ms) [25] and per-request billing make it unsuitable for low-latency analytics requiring <10 ms response times [56]. Caching layers are commonly deployed between compute nodes and object storage to mitigate these limitations. 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 [24] and industry solutions like Amazon Redshift [6] and Databricks [20]. 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, leading to redundant data copies when multiple nodes need to 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 [60] 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 [67], and Snowflake [60]. 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 [14, 16, 61, 62] is a classical optimization technique in database systems that evaluates predicates as early as possible 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 [53], DPUs [31], Smart SSDs [21], and SmartNICs [33]. These hardware accelerators contain dedicated computing capabilities to evaluate filters while scanning data, ensuring that only relevant records are emitted upstream. In modern cloud environments, major providers have integrated filter pushdown directly into their object storage services. Systems like Amazon S3 Select [7] and Azure Data Lake Storage [45] 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 a 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 [65] 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, 19, 45] or see degraded performance when attempting it [63].

3 LIQUIDCACHE ARCHITECTURE

Recent research has shifted toward memory disaggregation from compute servers [22, 30, 41, 43, 50, 52], 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.

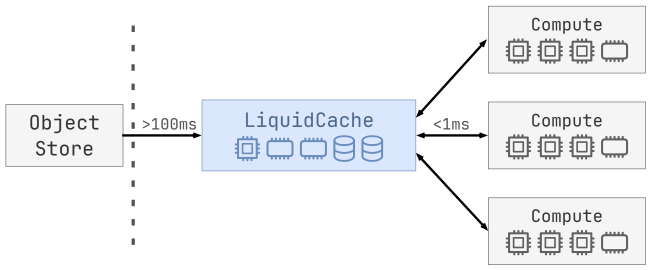

Figure 2: LiquidCache architecture – LiquidCache runs on a commodity elastic compute server, and is physically close to the compute servers that handle query execution.

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 [25]. 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 [45] or are slower with pushdown enabled [63]. 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 that perform computationally intensive operations such as aggregation, sorting, and joins, the cache server's CPU requirements are deliberately minimal and cost-efficient. A typical cache server provisions a memory/CPU ratio of 16:1, compared to 2:1 for compute nodes.

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

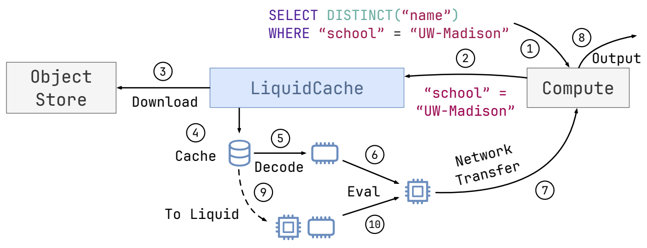

Figure 3: The life of a query in LiquidCache– Step 9 transcodes the Parquet into Liquid format in the background, allowing efficient predicate evaluation on a cache hit.

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. Like most cloud-native systems, a higher-level metadata 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. The compute server unparses the scanning operator back into SQL text and transmits it to the cache server. This approach allows the cache server to receive SQL queries with pushdown filters, similar to S3-Select [7], enabling various analytical systems to share the same cache infrastructure. LiquidCache also intends to support Substrait [55] in the future once it matures to handle the full range of filters that LiquidCache supports.

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.

3.3 Cache mechanisms

Cache Parquet bytes on cache-local disk. On cache misses, the server downloads the requested data range from object storage based on 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 [25]. Users can pre-populate 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. Once Parquet data is transcoded into Liquid format, LiquidCache caches them in memory. LiquidCache manages the data chunk at batch size (e.g., often 8192 rows) granularity in memory, i.e., each cache entry is one continuous data of one column of one file. This fine-grained cache management allows LiquidCache to cache only the data that the query needs. Each cached entry is uniquely identified by the file name, row group index, column index, and row number. LiquidCache uses a hierarchical hash table to reduce the index lookup cost – each column reader only uses row numbers to locate its cached entries.

When memory is full, LiquidCache uses a column-level LRU eviction policy. This approach recognizes that batches within the same column are typically accessed together, while different columns are accessed independently with varying frequencies. During eviction, LiquidCache simply removes data from memory rather than writing it back to disk, as the logical data already exists in Parquet format on disk.

Result caching vs data caching. LiquidCache caches data before filtering rather than after filtering. This design choice stems from our observation that data decoding, not filter evaluation, is the primary performance bottleneck. By caching data in the Liquid format, LiquidCache eliminates the heavy decoding overhead, enabling efficient evaluation of filters against the cached data. While result caching can benefit repetitive filters, it requires exact matches between filter expressions and data, significantly reducing cache hit rates. Our caching approach provides more flexibility and better cache utilization.

4 LIQUID FORMAT

Cache logical data, not its physical representation

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.

LiquidCache combines several state-of-the-art data encoding techniques for high compression ratios and efficient filter evaluation. These include FSST [15] for string compression, FastLanes [1] for integer bit-packing, and standard encoding techniques like dictionary encoding and FoR (frame-of-reference) encoding.

The key principle of LiquidCache encoding is that candidate encodings should never form data dependencies among elements, i.e., each element should be able to be decoded independently without depending on other elements. As a counter-example of this principle, the RLE (run-length encoding) encoding is not used in LiquidCache because decoding each RLE element requires knowing the values of all previous elements. As we will show in Section 4.2, this principle is the key to achieving efficient filter evaluation. LiquidCache combines these encoding primitives to form a cascade encoding chain, each of the intermediate encoding steps holds the logical equivalent but physically different representation of the data, LiquidCache exploits their unique properties to achieve high filter evaluation efficiency (detailed in Section 4.2).

Unlike traditional file formats that require strict compatibility between readers and writers through agreed-upon specifications, which often impedes the adoption of new encoding and compression techniques; LiquidCache takes a different approach. Since LiquidCache operates as a self-contained caching system where data is both written and read by the same components, it can freely evolve its encoding schemes without concerns about ecosystem-wide compatibility. This architectural freedom enables LiquidCache to incorporate cutting-edge compression and encoding techniques as they emerge, maximizing performance and efficiency.

4.1 Liquid data representation

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 [35]. The system's flexible architecture allows dynamic updates to its encoding chain as new techniques emerge.

4.1.1 Encode strings. As shown in Figure 4, LiquidCache encodes string arrays using dictionary encoding with unique values and reference keys, similar to Parquet's approach. The dictionary's key and value arrays are then encoded separately.

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

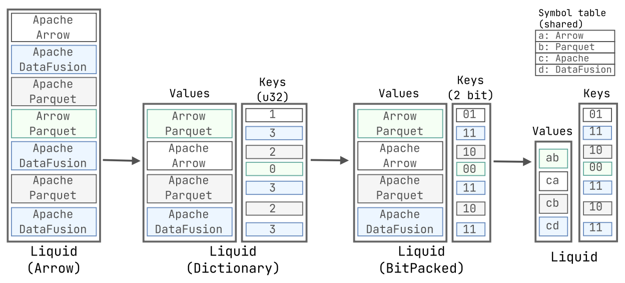

Figure 4: LiquidCache's string representation – Each of the steps is a valid representation that can be used for filter evaluation, the corresponding data size and transcoding time can be found in Section 5.9.

leverages FastLanes [1] encoding to efficiently bit-pack these keys using the minimal required bit width.

For dictionary value compression, LiquidCache employs FSST encoding [15], 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.

4.1.2 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 integer bit width is calculated based on the range of values (max-min) rather than dictionary size.

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.

4.2 Co-design with filter evaluation

A good data representation without deep query engine integration does not exploit the full potential of the data representation. As we will show in Section 5.10, naively using the Liquid format for filter evaluation can be slower than simply evaluating on Parquet. Liquid

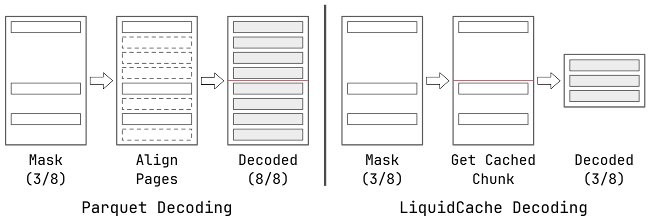

Figure 5: LiquidCache's selective decoding comparing to Parquet's decoding – Parquet (left) has to decode an entire page even if just one element is needed. LiquidCache (right) allows each element to be decoded independently.

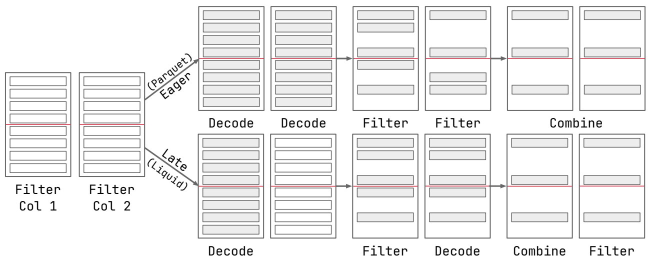

Figure 6: LiquidCache's late materialization compared to Parquet's eager materialization – A Parquet decoder has to materialize all columns before evaluating filters, while LiquidCache only materializes columns as needed.

format co-designs with the filter evaluation process to exploit its unique ability to decode individual elements without decoding the entire array.

4.2.1 Selective decoding. Filter evaluation on filtered columns produces a boolean mask identifying elements that satisfy predicates. This mask enables selective decoding, only decoding elements from projection columns that pass the filters. While conceptually straightforward, selective decoding provides limited benefits for Parquet files due to its page-based design. As illustrated in Figure 5, even when the filter mask selects 9 out of 16 elements, a Parquet decoder must decode the entire page containing all 16 elements since it cannot decode individual elements within a page. Selective decoding only benefits Parquet in cases where entire pages can be skipped. In contrast, LiquidCache is designed from the ground up with efficient row-level filter evaluation as a core use case. Its element-wise encoding scheme allows independent decoding of any requested elements without touching unneeded data.

4.2.2 Late materialization. Unlike selective decoding, which optimizes output data construction, late materialization focuses on optimizing the filter evaluation process. It explicitly targets scenarios with multiple chained filters across different columns – a pattern commonly found in analytical queries.

Figure 6 illustrates late materialization in action. In the eager approach without late materialization, the system first decodes all columns involved in filtering, then evaluates predicates on each column independently, and finally combines the resulting boolean masks to determine which rows satisfy all conditions. With late materialization enabled, the system processes columns sequentially – it decodes the first column, evaluates its filter to produce a boolean mask, Then, this mask is used to selectively decode only the qualifying elements from the second column. This cascading process means each subsequent column needs to decode progressively fewer elements as more filters are applied. Late materialization reduces both decoding work and filter evaluations by processing only data that passed previous filters.

Late materialization introduces heavy overhead from bit deposit operations [42]. For late materialization to be profitable, selective decoding must provide substantial benefits on subsequent columns. However, this is challenging to achieve with Parquet due to its page-based compression scheme, preventing efficient selective decoding within pages.

4.2.3 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.

While this encoding-aware approach is highly efficient, not all filter operations can be evaluated on encoded data. Substring pattern matching, for instance, requires access to the raw string data. However, LiquidCache's cascading encoding strategy (Section 4.1) 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.

This selective-decoding strategy ensures that common filter operations like equality, inequality, and pattern matching are evaluated using the most efficient representation possible, minimizing unnecessary decoding work. In contrast, Parquet's general-purpose compression schemes make encoding-aware filter evaluation impossible since they destroy element-level information during compression.

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.

4.3 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.

4.3.1 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 [58], allowing the transcoding cost to be amortized across many query executions.

4.3.2 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 [58] requiring CPU over-provisioning to handle peak loads. LiquidCache exploits these quieter periods to perform transcoding work efficiently using otherwise idle CPU cycles.

4.3.3 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 [28, 29]. 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.

4.4 Discussion

Comparison with other modern data formats. Modern file formats such as BtrBlocks [35], LanceDB [39], Nimble [44], and Vortex [54] 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 [34].

5 EVALUATION

Our evaluation aims to answer the following key questions:

- How does LiquidCache's performance compare to state-of-the-art caching designs in terms of latency, network usage, CPU time, and memory usage?
- What are the trade-offs between Liquid's novel encoding strategy versus established formats like Arrow and Parquet?
- How effectively does Liquid's background transcoding mechanism hide transcoding overhead from query latency?
- How well does LiquidCache handle diverse real-world analytical workloads?

5.1 Implementation details

We implement LiquidCache on top of Apache DataFusion [38], a high-performance analytical engine consistently ranking among top performers for Parquet workloads [37]. Our implementation consists of approximately 12k lines of Rust code, with an additional

5k lines contributed to upstream DataFusion, Arrow, and Parquet. LiquidCache implements a custom TableProvider interface that seamlessly replaces existing Parquet readers with our optimized versions. This design enables LiquidCache to integrate with existing systems [10, 27, 32, 36, 39, 57] through minimal code changes – typically requiring less than 10 lines of modification.

5.2 Evaluation setup

We evaluate LiquidCache using ClickBench [17, 18], an industry-standard analytical benchmark with 15GB (100M rows) of real-world web analytics data. This dataset contains complex filter patterns and variable-length fields [49], presenting significant challenges for filter pushdown optimizations.

We focus primarily on queries with filters of varying selectivity. Other ClickBench queries (full scans or compute-intensive operations) are outside our focus but are included in Figure 16, demonstrating that LiquidCache maintains competitive performance across all workloads. Table 1 details key characteristics of our selected query subset, including query 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. The queries span diverse scenarios – from simple single-column filters to complex multicolumn predicates, with selectivities ranging from highly selective (<0.01%) to relatively broad (13.2%), and data sizes from 0.2GB to 14.8GB.

Experiments run on a CloudLab [23] machine (6525) with 16 cores (32 threads) and 128GB RAM, using a 10Gbps network to simulate typical cloud environments. We disabled TLS encryption and used the default Arrow Flight without compression to minimize overhead. Each query executed five times, with results averaged across the final three runs to account for warm-up effects. All latency measurements represent end-to-end execution time from SQL parsing to result retrieval.

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

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.

5.3 Baseline implementations

We implement three representative baselines that match the caching architecture discussed in Section 2.1, representing key approaches used in industry today: file serving, Parquet filter pushdown, and Arrow filter pushdown. These baselines cover the spectrum of performance-memory trade-offs in disaggregated caching systems.

LiquidCache: Our proposed cache system combines the memory efficiency of Parquet with the performance of Arrow through its novel Liquid encoding format, transcoding data on the fly to a representation tailored for filter evaluation.
Arrow (pushdown): Represents a filter pushdown cache that stores data in Arrow format rather than Parquet, eliminating Parquet decoding overhead. It uses embedded DataFusion for filter evaluation, providing fast data access at the cost of higher memory usage than Parquet-based approaches.
Parquet (pushdown): Implements a filter pushdown cache that evaluates 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 significantly reducing memory usage, it incurs CPU overhead from filter pushdown on Parquet.
Parquet (file server): The simplest form of disaggregated caching, 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 [12]. These systems fall between the two extremes regarding memory consumption and compute overhead and are not included here for space reasons.

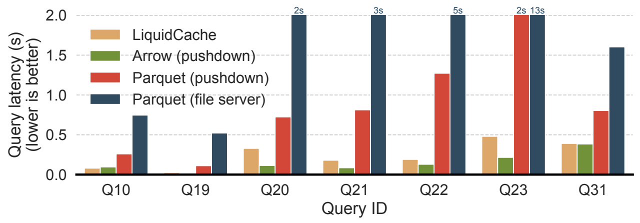

Figure 7: Overall query latency of LiquidCache and baselines (lower is better). Arrow (pushdown) and LiquidCache perform similarly, and both outperform Parquet (pushdown) and Parquet (file server).

5.4 Overall results

Our first experiment compares the end-to-end latency of LiquidCache and the baselines. Figure 7 shows the different queries on the x-axis and latency on the y-axis (lower is better).

Arrow (pushdown) and LiquidCache perform similarly, both significantly outperforming Parquet (pushdown) and Parquet (file server). Parquet (file server) consistently shows the worst performance, with latencies up to 13 seconds for some queries. Parquet (pushdown) performs better but still reaches latencies up to 2 seconds, while both 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. In contrast, Arrow (pushdown) can directly apply filters on in-memory Arrow data.

This experiment demonstrates that LiquidCache achieves latency comparable to Arrow (pushdown) while significantly outperforming other approaches. The following experiments analyze resource usage across all systems to explain these performance differences.

5.5 Network traffic

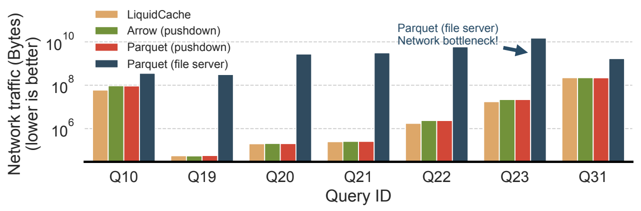

Figure 8: Network traffic results (log scale) – Pushdown-enabled systems have similar network traffic, while Parquet (file server) has much higher traffic because it has to transfer large amounts of unfiltered data.

Figure 8 quantifies how filter pushdown reduces network traffic between cache and compute nodes compared to file serving, measured as total bytes received at the client's network interface. The x-axis shows the different queries, while the y-axis displays network traffic on a logarithmic scale (lower is better).

As expected, the three pushdown-enabled systems (LiquidCache, Arrow (pushdown), and Parquet (pushdown)) exhibit similar network traffic patterns since they only transfer records matching the filter predicates. In contrast, Parquet (file server) transfers orders of magnitude more data by sending entire unfiltered Parquet chunks over the network. This performance gap is particularly pronounced for highly selective queries (Q19-Q23), where pushdown systems transfer only a small fraction of records. The difference narrows for less selective queries (Q10, Q31) where most data passes the filters. Excessive network traffic not only saturates precious bandwidth but also incurs substantial CPU overhead in cache and compute nodes' network stacks for packet processing.

Static file servers like Parquet (file server) are ill-suited for disaggregated caching due to excessive network overhead, being viable only for full table scans or low-selectivity queries. Filter pushdown systems substantially reduce network traffic by filtering at the cache server, making them far more appropriate for disaggregated caches.

5.6 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 filter pushdown operations, including decompression/decoding and filter evaluation.

Figure 9 compares the total CPU core time consumed by each system on the cache server, showing different queries on the x-axis and aggregated compute time across all cores on the y-axis. 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,

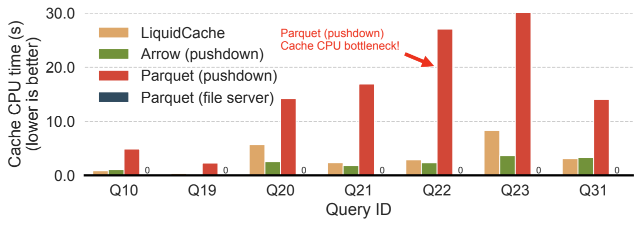

Figure 9: Total CPU core time of LiquidCache vs baselines. Parquet (pushdown) spends significantly more CPU time than other baselines because it must decode Parquet on the cache server.

Arrow (pushdown) is most efficient since it operates directly on in-memory data without decoding. Notably, LiquidCache achieves comparable CPU efficiency to Arrow (pushdown) despite working with compressed data. This efficiency stems from Liquid's faster decoding scheme (Figure 13) and LiquidCache's ability to skip unnecessary decoding through tight filter pushdown integration (Figure 15).

Parquet (pushdown) is ill-suited for disaggregated caching due to its high CPU demands, which undermine the benefits of cache-compute separation. In contrast, LiquidCache delivers similar CPU efficiency to Arrow (pushdown), making it ideal for disaggregated caching environments.

5.7 Cache memory consumption

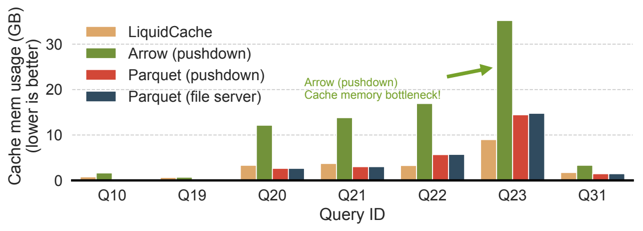

Figure 10: Memory usage of LiquidCache and baselines. Arrow (pushdown) requires significantly more memory than other systems, incurring prohibitively high operational cost, while LiquidCache achieves memory efficiency comparable to Parquet's compressed format.

Memory cost represents half of data center expenses [41]. We analyze each system's memory footprint by measuring 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.

Figure 10 compares memory consumption across systems with queries on the x-axis and cache memory usage on the y-axis. 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 13.

Maintaining low memory usage is critical for cloud-native cache servers, as memory and other resources (e.g., compute and network) are provisioned together. High memory requirements force the allocation of correspondingly high compute resources, leading to inefficient utilization and higher operational costs, as unnecessary computing, network, and storage resources are bundled with larger memory servers.

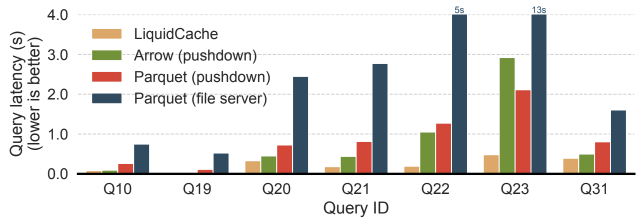

Figure 11: Query latency under 16GB memory constraint. LiquidCache performs the best, while Arrow (pushdown)'s latency degrades significantly, becoming comparable to or worse than Parquet (pushdown).

Constrained memory experiment. To evaluate performance under memory constraints, we limited the cache server's memory to 16GB using cgroups and ran the same queries. Figure 11 shows that LiquidCache consistently performs best, while Arrow (pushdown)'s performance degrades significantly, becoming comparable to Parquet (pushdown). For Q23, Arrow (pushdown) performs worse than Parquet (pushdown) because caching Parquet data is only beneficial when entire column chunks are cached in memory. Without Parquet's page index [9], a cache miss forces the Parquet reader to scan from the beginning of the column chunk until reaching the desired page. This sequential scan proves slower than Parquet (pushdown), which avoids the overhead of cache state management.

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.

Having demonstrated that existing approaches each suffer from significant limitations – Parquet (file server)'s high network overhead, Parquet (pushdown)'s excessive CPU usage, and Arrow (pushdown)'s substantial memory footprint – we have shown that it is possible to architect a system that avoids these three bottlenecks. In the following sections, we analyze the internal mechanisms that enable LiquidCache's superior performance characteristics.

5.8 Decoding revisited

The core finding of this study is that decoding is the bottleneck for filter pushdown on Parquet. We revisit LiquidCache's decoding time and how it compares to the theoretical optimal. We sampled the cache server execution and categorized time into three categories: filter evaluation, decoding, and memory stall. Filter evaluation represents useful time spent evaluating the filter. Decoding is 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 12, LiquidCache significantly reduces overall CPU time from 27s to 2.7s, a 10× improvement, reaching 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 5.10), 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.

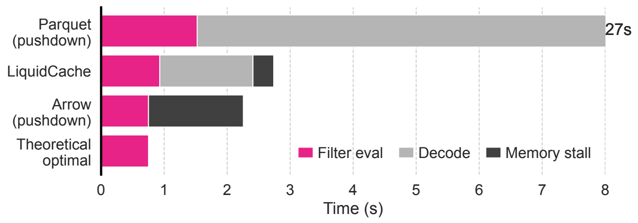

Figure 12: Decomposed CPU time spent on cache server for ClickBench Q22. LiquidCache spends similar total time as Arrow (pushdown), but primarily on decoding compressed data, while Arrow (pushdown) suffers from memory stalls due to scanning 4× more uncompressed data.

5.9 Transcoding cost

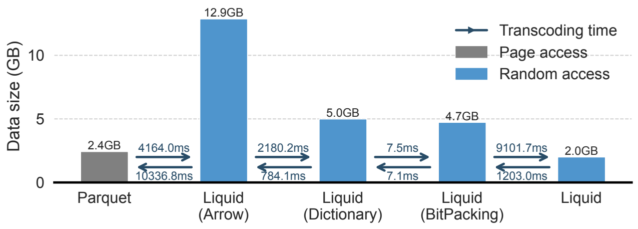

Figure 13: Memory usage and transcoding cost of "Title" column of the ClickBench – the largest column in the dataset. Liquid's encoding achieves a similar compression ratio as Parquet yet still maintains random access capabilities to each element.

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 first analyzes direct transcoding costs among Liquid's different data representations, then demonstrates how LiquidCache effectively hides these costs from the critical query execution path through background processing.

Figure 13 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 from Arrow to Liquid takes similar time as Arrow to Parquet, but Liquid decoding to Arrow is over 2× faster than Parquet decoding
- Dictionary encoding provides significant compression (>2×), matching the compression gains from Liquid (BitPacking) to full Liquid, but with 4× longer encoding time
- BitPacking offers modest additional compression (6%) but processes data over 300× faster than dictionary encoding

While compression ratios and timings vary across data types and workloads, these patterns hold consistently. We provide an open-source analysis tool in our artifact for examining other columns.

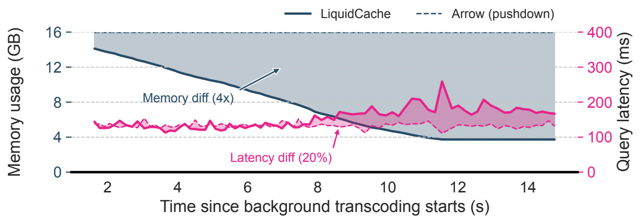

Figure 14: LiquidCache's latency and memory usage on Q21 – LiquidCache gradually transcodes Arrow into Liquid in background, avoiding latency spikes on the critical path of query processing.

Next, we analyze how LiquidCache handles transcoding costs during query execution by case-studying Query 21 from ClickBench. 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 [58].

Figure 14 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 left and right y-axes show memory consumption and query latency, respectively. The shaded area represents the difference between LiquidCache and Arrow (pushdown).

When a cache miss occurs, 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%. This modest latency increase demonstrates the efficiency of LiquidCache's Liquid encoding for query processing. As discussed in Section 4.3,

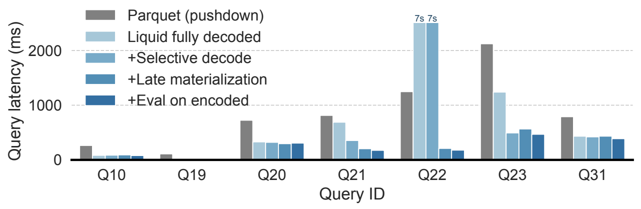

Figure 15: Ablation study of LiquidCache's decoding optimizations vs Parquet (pushdown) as the baseline.

LiquidCache's design ensures transcoding never blocks the critical query execution path – queries can efficiently operate on data in any intermediate representation without waiting for transcoding to complete. The near-identical initial query performance between LiquidCache and Arrow (pushdown) during the first few seconds confirms this non-blocking behavior, with no latency spike.

5.10 Ablation study

We now analyze how each decoding optimization described in Section 4.2 contributes to overall performance. Using the same experimental setup, we selectively enable optimizations one at a time to measure their impact. Together, these techniques achieve up to 10× performance improvement over Parquet (pushdown).

Figure 15 illustrates the performance impact of different decoding optimizations across queries. The most basic configuration, "Liquid fully decoded", decodes entire arrays into Arrow format even when accessing single elements - a coarser granularity than Parquet's page-level decoding. Despite this limitation, it still outperforms Parquet (pushdown) on single-filter queries like Q20, where full-column decoding is necessary. This occurs because Liquid data decodes faster than Parquet, as demonstrated in Figure 13.

Selective decoding provides significant benefits when constructing output data using filter evaluation masks to decode only relevant data. This optimization is particularly effective for queries with highly selective filters (e.g., Q21, Q23). However, selective queries, like Q22, see limited benefit as they are bottlenecked by the filter evaluation phase, highlighting the importance of late materialization. Late materialization optimizes queries with multiple filters (e.g., Q21, Q22) by fully decoding only the first filtered column.

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 10× improvement over late materialization and more than 76× improvement over baseline Parquet (pushdown).

The ability to filter directly on encoded data provides additional optimization opportunities when filter expressions are compatible with the encoding scheme (e.g., equality filters on strings). While this optimization shows modest benefits in our evaluation since only Q21 and Q31 have compatible filters on small columns, we demonstrate its potential impact through an additional experiment

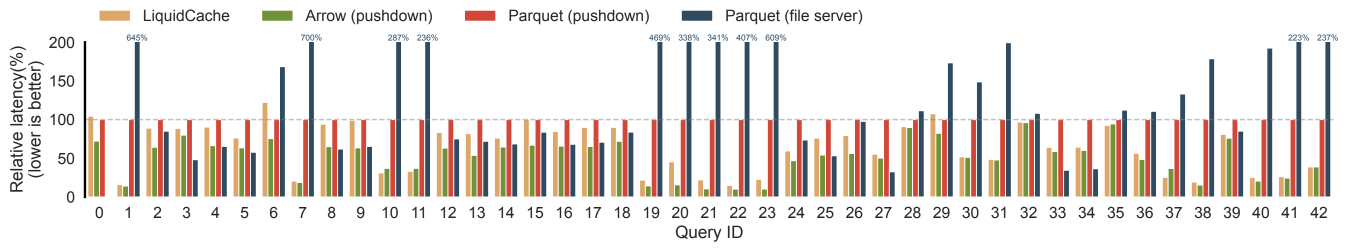

Figure 16: ClickBench latency relative to Parquet (pushdown). Lower is better.

on a variation of Q22 with predicates on large columns. As shown in Table 2, this achieves an 11× improvement over late materialization alone, suggesting significant performance gains when applying this technique to larger columns.

5.11 Full ClickBench results

LiquidCache is designed as a plug-and-play system; any query engine that understands Arrow [10, 18, 32, 36, 37, 39, 47, 48, 57] can benefit from it with minimal changes. Figure 16 shows the entire ClickBench results running with the same setup as previous experiments. The x-axis represents query ID, and the y-axis shows latency relative to Parquet (pushdown).

For queries with filters, LiquidCache consistently performs similarly to Arrow (pushdown) despite using 4× less memory. For queries without filters, LiquidCache maintains comparable performance to Arrow (pushdown) and ranks among the best performers across the entire benchmark.

The full ClickBench reveals an interesting observation: Parquet (file server) occasionally performs best, even for queries with filters, such as Q27. The filter on Q27 is highly non-selective, with 99% of data passing through, resulting in uncompressed Arrow IPC format being sent over the network and causing significantly higher traffic than Parquet (file server), which transfers unfiltered but compressed Parquet data. These results highlight that while filter pushdown effectively optimizes most analytical workloads, some workloads—especially those without filters or with non-selective filters—benefit from bypassing pushdown altogether. We leave the development of a dynamic approach for deciding whether to push down filters based on cardinality estimation as future work.

6 RELATED WORK

Data Lakehouse. Modern data analytics have shifted from on-premise to cloud-native Lakehouse architectures [11, 12, 46], 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 [13, 37, 47, 48] 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 and send the filtered data to the query engine, reducing both CPU and network costs.

Modern encoding and columnar file formats. Since the initial release of Parquet in 2013 [59], many new encoding schemes have been proposed to improve its compression and decoding efficiency, including FSST [15], ALP [2], FastLanes [1], Roaring [40], and Pseudodecimal [35]. However, introducing these new encoding schemes to Parquet would break backward compatibility, and the ecosystem has become locked in [34] to a minimal set of encoding schemes that are well-supported by major query engines.

File formats like Vortex [54], Nimble [44], and BtrBlocks [35] have been proposed to replace Parquet, even Parquet itself is evolving to modernize [9]. However, adoption of 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, 20, 32, 60]. Building on this success, both researchers and industry have been exploring ways to further disaggregate compute and memory resources. Several systems like LegoOS [52], TPP [43], FaRM [22], AIFM [50], Pond [41], Redy [66], and Hao et al. [30] 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 CONCLUSION

This paper presents LiquidCache, a novel pushdown-enabled, disaggregated cache system for cloud-native analytics that achieves unprecedented performance while maintaining compatibility with existing infrastructure. By co-designing the Liquid format with filter pushdown semantics, LiquidCache dramatically reduces data decoding overhead – the primary bottleneck in pushdown-based cache systems. 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. 2025. Apache DataFusion Comet. https://github.com/apache/datafusion-comet A high-performance accelerator for Apache Spark built on Apache DataFusion.

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.]).

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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).

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

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

[46] 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.

[47] 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.

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

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

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

[55] Substrait Project. 2024. Substrait: Cross-Language Serialization for Relational Algebra. https://substrait.io. https://substrait.io Accessed: 2024.

[56] Raphael Taylor-Davies and Andrew Lamb. 2022. Querying Parquet with Millisecond Latency. InfluxData Blog. https://www.influxdata.com/blog/querying-parquet-millisecond-latency/

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

[58] 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.

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

[60] 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.

[61] 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.

[62] 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).

[63] 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.

[64] 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.

[65] 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.

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

[67] 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.