SqueezeCache: Beyond "Optimal" Eviction for Data Analytics

Xiangpeng Hao*[,1], Nikhil Nayak[1], Proteet Paul[1], JP Guthi[1], Andrew Lamb[2], Jacopo Tagliabue[3], Andrea Arpaci-Dusseau[1], Remzi Arpaci-Dusseau[1]
[1]University of Wisconsin-Madison [2]InfluxData [3]Bauplan Labs

ABSTRACT

In disaggregated cloud analytics, caching is essential to mask storage latency. However, existing mechanisms—treating data as opaque blocks to keep or evict—are fundamentally inefficient: queries often access only a small fraction of cached data (e.g., checking a string prefix), so most in-memory bytes are wasted even under optimal replacement policies.

We introduce SqueezeCache, a caching system that "squeezes" before evicting data. SqueezeCache caches data by how it is *used*, not just how frequently it is accessed: instead of discarding an entry, it transforms data into compact, lossy representations (e.g., string fingerprints, quantized integers, or extracted datetime components) that can still answer common query predicates. When representation is insufficient, SqueezeCache fetches full data from storage. By leveraging column lineage, SqueezeCache adapts to query patterns, generalizing result caching and materialized views while enabling cross-engine sharing. Evaluation on ClickBench shows that SqueezeCache improves cache hit ratios by up to 4× and reduces query latency by up to 22× compared to standard approaches.

PVLDB Reference Format:
Xiangpeng Hao*[,1], Nikhil Nayak[1], Proteet Paul[1], JP Guthi[1], Andrew Lamb[2], Jacopo Tagliabue[3], Andrea Arpaci-Dusseau[1], Remzi Arpaci-Dusseau[1]. SqueezeCache: Beyond "Optimal" Eviction for Data Analytics. PVLDB, 14(1): XXX-XXX, 2020.
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/LiquidCache.

1 INTRODUCTION

Caching is a core primitive in data-intensive systems. It bridges the gap between compute and storage, enabling systems to scale beyond memory while masking storage latency and cost. Modern cloud-native analytics, where data typically resides in remote storage [4–6, 47, 63], relies on effective caching even more, as the compute-storage gap is even larger [27]. The effectiveness of caching is determined by both *policies* – deciding which entries to keep in memory – and *mechanisms* – controlling how to move data between compute and storage.

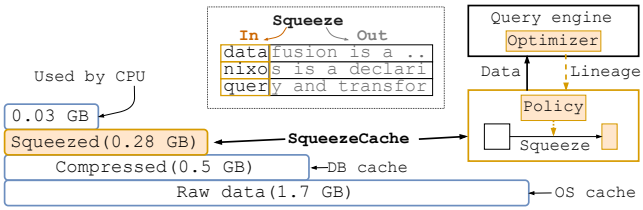

Figure 1: SqueezeCache caches data by how it is *used*, not just how frequently it is accessed. It significantly closes the gap between cached data size and utilized data size. Numbers shown for ClickBench Q25.

Prior work proposes many cache *policies* [33, 44, 49, 68, 71] that improve hit rates under diverse workloads. Cache *mechanisms* have received less attention and follow two main designs: (1) memory-mapped designs relying on the OS to move pages [8, 9, 22], and (2) application-managed designs that compress data before writing to disk [29, 32, 72].

Our study finds that **most cached data is never accessed, even under Belady's [13] optimal replacement policy.** The issue lies in the mechanism. Modern analytical systems typically cache data in *semantic units* (e.g., vectors, column chunks, or row groups). While this granularity aligns with columnar processing, it introduces a source of inefficiency: retaining a unit for a query operation (e.g., a filter, extraction, or projection) forces the cache to store the entire data representation of every value in that unit, even if the operation only consumes a few bits of information per value.

Consider a simple query: `SELECT a ... WHERE b = ''`. To process this, an engine first evaluates the filter `b = ''`. Our key observation is that such filters often rely on a tiny subset of the data: determining if a string is empty requires only its length, not its full byte content. Yet, because conventional caches treat data as opaque blocks, they must store the entire string just to preserve this length information.

This granularity mismatch creates a massive gap between data stored and data utilized. As Figure 1 illustrates with ClickBench Q25, while the CPU consumes only 0.03 GB of information to execute the query, the OS cache holds 1.7 GB of raw data. Even application-managed compressed caches are forced to retain 0.5 GB.

SqueezeCache addresses this inefficiency by introducing a third option: *squeeze*. Instead of the binary choice—evict entirely or keep entirely—offered by conventional mechanisms, SqueezeCache transforms entries into compact, lossy representations (e.g., retaining only the string length). This approach reduces the memory footprint to 0.28 GB (Figure 1) while preserving enough information to resolve most filters. By decoupling retention from eviction, SqueezeCache complements existing policies: it maintains these lightweight summaries in memory and incurs disk I/O only when a query explicitly demands full fidelity.

While simple in concept, squeezing requires a holistic design

*Contact: xiangpeng.hao@wisc.edu

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

across the system to fully unlock its potential. We summarize three challenges below. First, the cache must efficiently transform user input into a squeezable representation and reconstruct the original view on `get`. Second, the squeezeable data must perform eager evaluation on essential data first, and only read full data when necessary. Third, we need a squeeze policy – along with the cache policy – to decide how much to squeeze for each entry.

We propose SqueezeCache to scale beyond memory limits by addressing these challenges. First, SqueezeCache provides **Squeezable Data Layouts** (Section 4) for common datatypes (strings, integers, floating-point numbers, and datetimes) that structurally decompose data to allow partial retention. Second, it employs **Lineage Pushdown** (Section 7.1) to enable predicate evaluation directly within the cache using existing query engine infrastructure. Third, it uses a **Multi-State Squeeze Policy** (Section 6) that treats data fidelity as a spectrum, dynamically adjusting the "squeeze level" to balance memory pressure and I/O cost. SqueezeCache requires no changes from end users and has become the default "eviction" mechanism in the broader LiquidCache project.

We evaluated SqueezeCache using real-world workloads including ClickBench and JSONBench. Our results show that SqueezeCache improves cache hit ratios by up to 4× and reduces end-to-end query latency by up to 22× compared to baselines. Additionally, it reduces I/O traffic by up to 12× in memory-constrained environments. Our contributions are as follows:

- We propose a squeeze operator that enables larger-than-memory pushdown caching.
- We build SqueezeCache, a reference implementation that integrates squeeze into existing systems.
- We evaluate SqueezeCache through comprehensive experiments.

2 BACKGROUND

2.1 Caching Policies and Mechanisms

Efficient caching is fundamental to database performance, and a plethora of cache replacement policies have been proposed over the decades. Classical algorithms like LRU-K [49], ARC [44], and 2Q [33] are widely deployed, balancing recency and frequency to maximize hit rates. Recent years have seen a surge in machine learning-based and heuristic-driven policies presented at top-tier venues. For instance, ReCache [10] introduces reactive caching for heterogeneous data, ChronoCache [28] enables predictive mid-tier query result caching, while FIFO-based queues [68] and SIEVE [71] challenge the complexity of traditional eviction schemes by demonstrating that simpler, scalable eviction can match or outperform sophisticated ones. More recently, approaches like linear elastic caching [35] leverage ski-rental algorithms to optimize for cloud cost models, and other works focus on ensuring cache consistency and monotonicity [18].

Beyond policy (what to evict), significant attention has been paid to the *mechanism* of caching (how and where to store data). In the era of disaggregated storage, systems like Crystal [25], STsCache [34, 39], and TSCache [41] explore unified and semantic caching to push data closer to compute. Tiered memory systems [32, 42, 48, 52] leverage CXL and NVMe to extend DRAM capacity, while domain-specific caches like MinIO [46] mitigate data stalls in DNN training. At the infrastructure level, CacheLib [14] provides a production-grade caching engine deployed at scale, and Ditto [54] introduces elastic memory-disaggregated caching. However, most existing work treats the cached unit as an opaque block—either keeping it entirely or evicting it entirely. SqueezeCache differs by introducing a Squeeze mechanism that partially retains data, bridging the gap between binary eviction decisions.

2.2 Caching for Cloud-Native Analytics

Cloud-native architectures have changed the assumptions of database caching. With compute-storage separation [47, 63], data resides in remote object stores, making local caching critical to mask latency and reduce costs [27]. Cloud warehouses like Snowflake [57], Google Napa [3], Amazon Redshift [60], and Krypton [19] rely on local SSDs and memory for hot data. FlexPushdownDB [69] hybridizes caching with computation pushdown, while CompuCache [70] explores remote caching using spot VMs. SqueezeCache aligns with this trend, optimizing the *value* density of cached data.

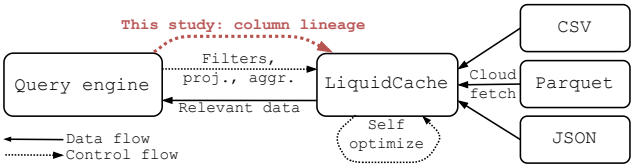

Figure 2: LiquidCache architecture. LiquidCache serves as the host environment for SqueezeCache. It sits between cloud storage and the query engine, exposing a schema-aware interface that supports control flow pushdown. This allows SqueezeCache to analyze query intent and optimize data layout to improve memory utilization.

2.3 Architectural Prerequisites

SqueezeCache relies on a cache that is not merely a block store but an active component with insight into data semantics and query intent. It targets the "Smart Cache" architecture emerging in modern warehouses and lakes (e.g., Snowflake [57], Crystal [25], LiquidCache [31]). We identify three prerequisites: **1. Schema-Awareness.** The cache must understand data structure (e.g., columns, types) to apply type-specific squeezing (e.g., datetime extraction) that generic page caches cannot. **2. Fine-Grained Accessibility.** The storage format must support reading sub-components (e.g., columns or byte ranges) without fetching entire objects. Columnar formats like Parquet and Arrow naturally satisfy this. **3. Control-Plane/Data-Plane Coordination.** The query engine must inform the cache about data usage (e.g., "this column is only used for a year filter"). This pattern is common in systems with pushdown interfaces [25, 69].

LiquidCache. We implement SqueezeCache within LiquidCache [31], a caching system that explicitly satisfies these prerequisites. Figure 2 shows the architecture: it sits between the cloud storage and the query engine, exposing a schema-aware interface that supports control flow pushdown. LiquidCache automatically manages data fetching and encoding, providing the necessary hooks for SqueezeCache to intercept eviction decisions and inspect column lineage. While we use LiquidCache as our reference implementation, the principles of SqueezeCache apply to any system fitting this architectural pattern.

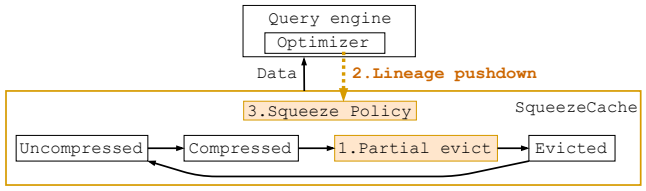

Figure 3: Overview of SqueezeCache. It has three core components: squeezable data layouts, lineage pushdown, and squeeze policy.

2.4 Relationship to Existing Mechanisms

SqueezeCache generalizes **result caching** and **materialized views**. Unlike these engine-specific approaches, SqueezeCache enables cross-engine sharing and provides a unified caching layer for global memory management. Furthermore, SqueezeCache improves cache hit rates by matching column lineage rather than requiring exact query matches. **Predicate caching** [53] is a special case of SqueezeCache where the squeeze policy generates a predicate-specific bitmask. Like predicate caching, SqueezeCache automatically determines column usage and applies similar optimizations.

3 OVERVIEW

The core insight of SqueezeCache is that caching effectiveness should be driven by *how data is used*, not just *how frequently it is accessed*. By analyzing query patterns, a cache can transform data into compact, lossy representations—"squeezing" it—to retain only the essential information needed for query processing. To illustrate, consider a (stackoverflow) query counting posts mentioning "Rust", grouped by day of the week:

```
SELECT EXTRACT(DOW FROM Date) AS day_of_week, COUNT(*)
FROM table WHERE Title LIKE "%Rust%"
GROUP BY day_of_week;
```

Conventionally, a cache stores the full `Date` (64-bit integer) and `Title` (string). However, the query only requires the day-of-week (3 bits) and a substring match on the title. Storing the full data wastes most of memory. SqueezeCache addresses this inefficiency through three fundamental components, as shown in Figure 3.

1. Squeezable Data Layouts (Section 4). The foundation of SqueezeCache is the ability to store data in lossy, query-efficient formats, in addition to the standard lossless compression. Instead of opaque blocks, we design data layouts that allow partial retention. For example, `Date` can be squeezed to just the day-of-week, and `Title` can be squeezed to a membership fingerprint. The main challenge is designing these layouts to be compact enough to save significant memory while remaining expressive enough to answer common predicates without accessing the full data.

2. Lineage Pushdown (Section 5). To decide *what* to squeeze, the cache must understand query intent. SqueezeCache employs Lineage Pushdown to bridge the semantic gap between the query engine and the cache. By analyzing the column lineage (e.g., `Date` → `EXTRACT(DOW)`), the system identifies the minimal subset of information required. The challenge lies in extracting this information from complex query plans with low overhead and communicating it to the cache without tightly coupling the systems.

3. Squeeze Policy (Section 6). Finally, the system must decide *when* and *how much* to squeeze. Squeezing is not a binary decision but a spectrum. SqueezeCache introduces a policy that manages the lifecycle of cached entries, transitioning them through multiple states—from fully uncompressed, to losslessly compressed, to lossy "squeezed" representations, and finally to eviction. The challenge is to balance the trade-offs between memory pressure, CPU cost for transformation, and the potential I/O cost of retrieving full data when the squeezed representation is insufficient.

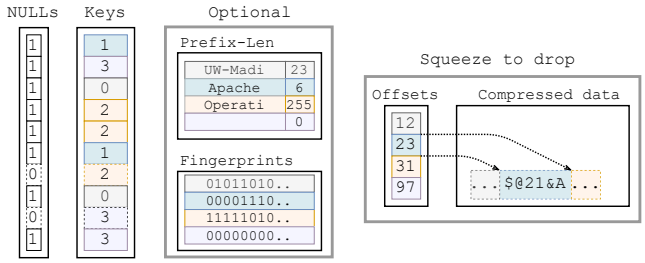

Figure 4: Overview of string layout. The byte-view of the string is similar to previous work, where each string has a header (which encodes string length, offset to the buffer, and a prefix), and a string buffer which contains the actual string data.

4 SQUEEZABLE DATA LAYOUTS

We define *Squeezing* as the general process of transforming in-memory data into more compact forms to free up space, ranging from lossless compression to lossy partial retention, and ultimately to disk eviction. While lossless compression is essential, it is a well-studied topic and is handled in prior systems like LiquidCache [31], BtrBlocks [36], and FastLanes [1] using cascading encoding. In this section, we focus on the novel contribution of SqueezeCache: **Partial Eviction**. Partial eviction creates lossy, memory-efficient representations that retain just enough information to answer common query predicates. Different data types and workloads require different partial eviction strategies. The following subsections show how these approaches apply to each data type.

4.1 Squeezing Strings

String workloads are dominated by comparisons (95%) and substring searches (5%). Since comparisons are often resolved by length or the first few bytes, SqueezeCache adopts a split representation similar to Umbra and Arrow's `StringView`: a fixed-size header inlines a short prefix and metadata (length, offset), while the full string resides in a separate buffer. Figure 4 shows the memory layout of our squeezeable string. It has a null array (one bit per string) to indicate which strings are null (note null strings are not empty strings). By default, all strings are dictionary encoded (we follow the same convention as Parquet). It then has a key array, which is a 16-bit unsigned integer array that stores dictionary keys. The actual string data is then compressed using FSST [16]. We store an additional offset array (4 bytes per unique string) to help locate the strings we need to decompress (selective decompression).

4.1.1 Droppable string buffer. Prior work inlines a prefix in the header, arguing that prefixes matter most. We take this further: under memory pressure, we drop the compressed string buffer from memory entirely. When a full string is needed (e.g., the prefix is

insufficient or the query projects the full value), we read the corresponding byte range from disk, locate the offset via the string index, and decompress only the required string. FSST's fast random access makes this practical. Since offsets and the compressed buffer dominate string storage, dropping the buffer frees significant memory and allows more strings to remain cached.

4.1.2 Single string buffer. Our string layout uses a single buffer rather than multiple buffers as in some prior designs. This saves 4 bytes per string by eliminating per-string buffer tracking. Fine-grained caching (typically 8K rows per batch) makes a single buffer practical.

4.1.3 Optimistic string length. We use 1 byte to encode string length instead of the standard 4 bytes. Value 255 serves as a sentinel indicating the string exceeds 255 bytes, triggering a disk read or decompression for the full length. Prior studies show over 99% of real-world strings are shorter than 255 bytes, so this saves 3 bytes per string with negligible performance impact.

4.1.4 String fingerprint. String fingerprints [56] accelerate substring search by hashing each 1-gram into buckets, producing a bit array encoding which characters may appear. Fingerprints quickly rule out strings that cannot contain the search needle. More buckets reduce false positives at higher storage cost; advanced hash functions can further reduce collisions. SqueezeCache uses 32 buckets with round-robin assignment by default.

4.1.5 Optional prefix and fingerprint. Both prefix and fingerprint have storage costs. If lineage analysis (Section 5.1) or administrator configuration indicates a column is never used for substring matching, we skip the fingerprint; similarly for prefixes. These components act as optional lightweight indexes—dropping them does not affect correctness.

4.2 Squeezing Integers

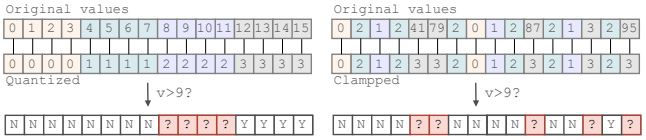

Figure 5: Overview of squeezable integer. Integers can be squeezed by quantizing (left) or clamping (right) to a fixed range. Squeezed integers are smaller but can still answer many filters.

Integers are the second most frequent data type. When used in predicates (typically comparisons), full precision is often unnecessary. For example, comparing a sensor ID rarely requires all 64 bits. Squeezing reduces precision; more squeezing saves memory but increases ambiguity (requiring disk reads).

Figure 5 shows two modes of integer squeezing in SqueezeCache: (1) quantization (left), and (2) clamping (right). We borrow the idea of quantization from machine learning and quantize integers to lower precision before fully evicting them to disk. In the example in Figure 5, the 16 values (4 bits per value) are quantized into 4 buckets (2 bits per value), cutting memory usage by half. When evaluating a predicate, we first quantize the predicate value, and compare the quantized value with the quantized column values. If the quantized values match, we treat the result as ambiguous; otherwise, we can safely decide the comparison from the quantized values. In the example in the Figure, 75% of the values can be evaluated by the quantized values.

Quantization works well for comparisons but loses information and cannot recover original values. SqueezeCache also supports clamping, which limits integers to a fixed range—values outside are clamped to the boundary (e.g., values > 3 become 3 in Figure 5). Unlike quantization, clamping preserves exact values within the range, making it suitable for columns used in both predicates and projections. Clamping works best when values cluster around a center with occasional outliers (e.g., sensor readings in anomaly detection); quantization suits columns used only for filtering where exact values are unneeded.

For signed integers, we first convert them to unsigned integers, using FoR (frame of reference) encoding – subtract every value by the minimum value of the column. Once quantization and clamping are applied, we then apply the bit-packing encoding to the integers. By default, we will quantize/clamp the integers to use half of the bits to represent the integer.

4.3 Squeezing Decimals

Decimals are physically integers: Decimal128 uses 128-bit integers to represent values, and Decimal256 uses 256-bit integers. The decimal datatype contains two parameters: precision and scale. For example, 123.45 has precision 5 and scale 2. This representation makes decimal computation isomorphic to integer computation, so it can reuse the integer squeezing techniques from the previous section. One exception is when the value range exceeds `u64`, the widest integer type supported by our primitive integer encoding. In that case, we first check the value range of the array and fall back to the fixed-length byte array encoding as discussed in LiquidCache [31].

4.4 Squeezing Floating Numbers

Floating numbers are first losslessly compressed using ALP[2], which are encoded into integers by multiplying them with an exponent (e) and an inverse factor (f). Floats that cannot be losslessly encoded using this approach are stored separately as exceptions. The encoded integers are subsequently compressed using Frame of Reference encoding. SqueezeCache squeezes the encoded integers using the quantization approach described in Section 4.2. Since the percentage of exceptional values is expected to be small, our implementation doesn't apply squeezing to these values.

4.5 Squeezing DateTimes

Time-related types (datetimes, timestamps) are physically integers. For example, Arrow's Date32 is a 32-bit integer (days since 1970-01-01). While integer techniques apply, we can exploit datetime semantics, specifically extract operations. For example, `extract(year, datetime)` is common. Snowflake reports datetime extractions in 16% of queries, making optimization crucial.

Figure 6 shows an example of datetime and its semantic based squeezing. At the top is the original integer value (1767811243). We can treat it as a normal integer and use the integer squeezing technique described in the previous section, but this does not fully use

datetime semantics. Knowing that this integer represents seconds since 1970-01-01T00:00:00Z, we can decode it into semantic components (year, month, day, etc.). If the query only uses day-of-week, we can squeeze the datetime to store only the day-of-week component in memory. SqueezeCache will automatically analyze the user SQL queries, and find all the operations applied to the datetime column, and pass the extracted information down to the cache (Section 5.1).

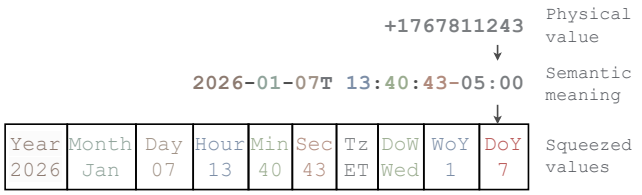

Figure 6: Datetime squeezing. Although physically stored as a single integer, datetime can be squeezed with different semantic components.

An alternative approach is to decompose the datetime into statically pre-defined components, such as year, month, and day, and only cache the "columns" that are actually being used in the user query. Although simple, this approach is not efficient for components like day-of-week or week-of-year, where the required information is not neatly represented by year/month/day. Our approach generalizes hard-coded year-month-day extraction and handles more complex extraction operations.

Multiple components can be extracted from the same datetime column, e.g., both year and month are frequently used, then both of them can be squeezed and cached in memory. Some components are superset of others, e.g., month is a superset of quarter, i.e., knowing month implies knowing quarter. In this case, we will only squeeze to the month component.

Once the desired components are extracted, SqueezeCache performs a cascading encoding to the extracted values: perform frame of reference encoding, then bit-packing encoding. The final value is then stored in the cache. A squeeze of Date32 to day-of-week uses only 3 bits per entry, leading to almost a 10× memory reduction. The squeezed data supports both predicate evaluation (when predicates extract a squeezed component) and final projection (when the projection only needs that component).

4.6 Squeezing Variants (JSON)

Variant is a datatype popularized by Databricks and Snowflake for semi-structured data (e.g., JSON). Conventionally, they are either stored as strings (requiring decoding and repeating field names) or as nested Parquet columns (which compress well but require a known schema). Variant provides a middle ground: arbitrary fields without repeating field names (similar to binary JSON), with native Parquet support. Since 2023, Variant has appeared in over 10% of Snowflake queries.

As shown in Figure 7, queries typically extract only a few fields, e.g., `loc.city` and `id`. A squeezed variant extracts these fields, avoiding the slow path of dynamic extraction. Under memory pressure, instead of evicting the entire variant, we shred it into columns for frequently accessed fields. A later `variant_get` reconstructs the object from shredded columns if available.

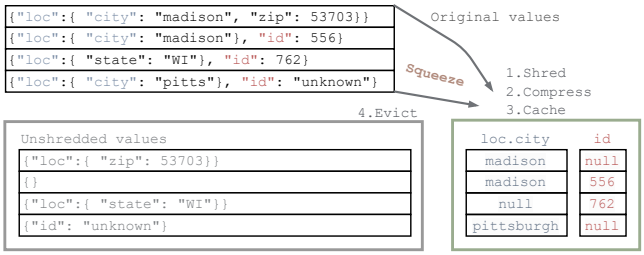

Figure 7: Squeeze a variant column into a set of shredded columns. Variant is a new datatype popularized by Databricks. We can shred the variant column into a set of shredded columns, and only cache the shredded columns in memory. When a variant get operation is called, we can reconstruct the variant object from the shredded columns.

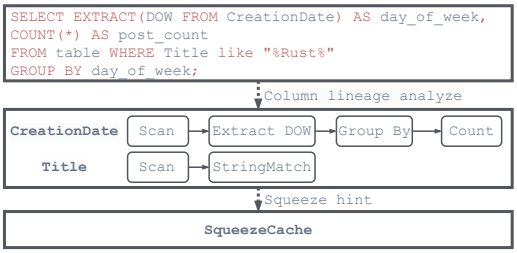

Figure 8: Column lineage example. SqueezeCache analyzes the query to extract the lineage of each column. For `CreationDate`, the lineage reveals that only the day-of-week component is needed. For `Title`, the lineage indicates a substring match, suggesting a fingerprint is sufficient. This information guides the cache to squeeze the data into compact, query-specific representations.

The Variant related operations are implemented as user-defined functions (UDFs) in the query engine. Users can call `variant_get` to extract fields from the variant column, or `variant_to_json` to convert a variant object to a JSON string. Similar to time-based squeezing, one variant column can be shredded into multiple columns, and we only cache the shredded columns in memory.

5 LINEAGE PUSHDOWN

Squeezing leverages the specific ways queries access data to reduce memory usage. However, the cache cannot deduce these access patterns from the raw data requests alone; it requires higher-level semantic information from the query engine. This section introduces "lineage pushdown," a mechanism that analyzes query plans to extract column usage patterns and pushes this context down to the cache, enabling it to safely serve partial data.

5.1 Lineage analysis

To serve queries using squeezed data, SqueezeCache must ensure that the lossy representation retained in memory is sufficient to answer the query. This requires determining not just which columns are accessed, but specifically which semantic components of those columns are used. SqueezeCache employs lineage analysis to reconstruct the operation history for each column. For instance, instead of fetching the full date on a cache hit, SqueezeCache might return 2025-01-01 (derived from 2025-07-26) if the lineage confirms the query only performs a year extraction.

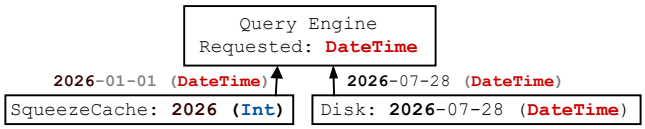

Figure 9: SqueezeCache creates a synthetic value when the cached data type does not match the expected type. The lineage analysis ensures that the unused components are not needed by the query.

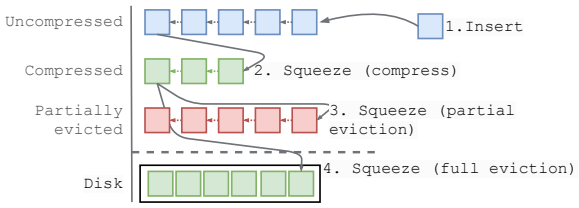

Figure 10: Overview of the life of an array in SqueezeCache. Arrays are inserted as uncompressed data. When the cache fills, squeeze operations progressively transition entries: uncompressed data is compressed, compressed data is partially evicted, and finally, squeezed data is evicted from the cache.

As shown in Figure 8, SqueezeCache analyzes the query's column lineage to determine which components to keep. For `CreationDate`, the lineage `[scan -> extract(dow) -> group by]` indicates only the `DOW` component is needed. For `Title`, the lineage `[scan -> filter(like)]` suggests that a membership fingerprint is sufficient to filter most rows.

Once we have the lineage, we simply check whether the extraction operations are the first operation applied to that column, if not, it means the query reads the full data, thus conflicts with our optimization. For example, if the lineage is [date -> extract(year) -> filter(year = 2025)], then we can safely apply the optimization, but if the lineage is [date -> filter(year > 2025-01-01) -> extract(year)], then we cannot apply the optimization.

Simple string/regex matching to determine extraction usage is neither sound nor complete: columns can have aliases, and nested expressions may still be optimizable even without pattern matches. SqueezeCache performs lineage analysis at the logical plan level. The analysis is implemented as an optimizer rule that recursively traverses expressions bottom-up and builds a column-usage map. Physical plan analysis is possible but more verbose.

Once the lineage is built, we need to pass it down to the LiquidCache. Since lineage analysis is implemented as an optimizer rule (i.e., there is no direct function call to receive its output), it has to embed the lineage information into the logical plan. SqueezeCache embeds the lineage information into the data provider node's schema's metadata, which allows a per-field metadata. This way, when the query engine reads from LiquidCache, LiquidCache checks schema metadata to decide which optimizations it can apply.

The cache API also needs to support this metadata. Specifically, the conventional `get` and `insert` APIs receive an optional cache-operation hint parameter. This hint tells the cache what operation is being applied, and the cache uses it to decide whether `get` can be served from squeezed data. For example, if the hint is `extract(year)`, and the cache also only has year component in memory, the cache will return the year component directly, without looking at the original disk data.

5.2 Handling Data Type Mismatches

Mismatches occur when squeezed data has a different physical type than the original schema. We handle two primary cases.

The first case involves scalar extraction, such as extracting the year from a datetime. While the squeezed data is logically an integer (the year), the query engine expects a datetime. As shown in Figure 9, SqueezeCache returns a synthetic datetime value where only the relevant component is valid (e.g., 2026-01-01T00:00:00Z when only the year 2026 is needed). This is safe because the lineage analysis ensures that the unused components are not needed by the query. The second case involves structural mismatches, such as a query scanning a struct with fields `a` and `b` when only `b` is actually used. If SqueezeCache returns only `b`, the schema mismatch breaks downstream operators. While decoupling logical from physical types could solve this, current standards like Arrow bind them tightly. Instead, SqueezeCache employs a lightweight plan optimization pass. Based on lineage analysis, it rewrites the query schema to request only the squeezed components (e.g., column `b` directly), removing the struct wrapper.

5.3 Tracking Squeeze Metadata

To determine optimal squeeze targets (e.g., whether to retain year or month), SqueezeCache must track usage patterns. Naive per-access tracking introduces prohibitive memory and synchronization overheads. To mitigate this, SqueezeCache adopts a hybrid strategy.
Coarse-grained tracking handles the common case where operations apply uniformly to an entire column (e.g., table scans). Usage is tracked at the query plan level (control plane), incurring zero execution-time overhead.
Fine-grained tracking handles filtered data where usage diverges across entries. To minimize overhead, SqueezeCache maps expressions to 2-byte IDs via a registry. Each entry maintains an 8-byte header storing the four most recent operation IDs. These are packed into a single 64-bit word, allowing updates via lock-free atomic Compare-And-Swap (CAS) operations. This design ensures thread safety with negligible impact on scan performance.

6 SQUEEZE POLICY

Once data is squeezable, the next question is when and how to squeeze it. Unlike traditional eviction which makes a binary decision (keep or evict), SqueezeCache exposes a fine-grained spectrum of memory states. This structure creates a rich optimization space and serves as a research framework for future work to explore complex policies that dynamically navigate these tradeoffs. This section discusses the default squeeze policy in SqueezeCache.

Figure 10 shows an array's life cycle in SqueezeCache. Arrays transition through four states: uncompressed, compressed, partially evicted, and evicted. Inserted as uncompressed, they are compressed, then partial eviction, and finally evicted as the cache fills. Specifically, squeeze targets the uncompressed queue, then compressed, then partial evcited.

6.1 Squeeze Spectrum

While conventional eviction offers a binary choice (keep or evict), squeezing offers a spectrum of options. SqueezeCache can choose "how far" to squeeze each entry, transitioning it from uncompressed, to compressed (lossless), and finally to partially evicted (lossy).

Lossless squeeze (compression) is the first step. This generalizes cascading encoding, balancing compression ratio against decoding performance. Our squeeze policy extends this tradeoff to include I/O costs: holding compressed data reduces memory pressure while still avoiding expensive disk reads. Prior work on learning optimal cascading encodings can directly inform this stage of the squeeze policy. *Lossy squeeze* (partial eviction) is the next step, trading fidelity for memory. Aggressive squeezing reduces memory usage but increases the likelihood that a query will require a disk read. Unlike the lossless stage, lossy squeezing forces a choice about *what* data to discard, as we discuss in the next section.

6.2 Selection Policy

When the system is under memory pressure, it must pick a cached entry to squeeze, similar to an eviction policy. Figure 10 shows that SqueezeCache uses three queues to manage cached entries and prioritizes squeezing less-squeezed entries. Alternatively, the cache can use a single queue, but this ignores the entry's current state. Within each queue, we can apply standard eviction policies such as LRU, FIFO, S3-FIFO, and SIEVE. SqueezeCache implements these policies, and by default uses FILO (first in, last out), since analytical cache workloads are often scan-intensive.

Similar to eviction, squeezing can be passive (triggered when the cache runs out of memory) or active (background threads proactively squeeze entries). This approach trades extra CPU overhead for better latency on a cache full. SqueezeCache by default uses a passive squeeze policy.

Unlike eviction, squeezing must decide what to do with the squeezed entry after the state transition. For example, if the squeeze policy uses a LRU, should it add the squeezed entry to the front or the back of the LRU queue? SqueezeCache by default treats a squeezed entry as a fresh insert (it is inserted into the corresponding queue as a new entry).

Partial eviction is lossy, so when we squeeze a compressed entry into a partially-evicted entry, we also write the full data to disk and record its location. This allows later hydration (Section 6.4) to read it back. By default, SqueezeCache writes the compressed data (LiquidArray [31]) to disk, because it is highly compressed, and requires no deserialization on read.

6.3 Retention Policy

Given lineage information (Section 5.1), the cache must decide which components to retain. The simplest case is when the query extracts a single component, such as `extract(year)` from a datetime column. Here, the cache keeps only the year in memory and squeezes the rest to disk. When queries reference multiple fields (e.g., year and month, or several JSON paths), the cache ideally uses a cost-based policy considering memory pressure, query history, and compressibility. Currently, SqueezeCache retains all referenced fields; we leave cost-based selection to future work.

A second complexity arises from compatibility between fields: some can be derived from others. For example, if lineage analysis shows that both month and quarter are extracted from the same datetime, we observe that quarter can be derived from month—so retaining month provides quarter at no extra cost. Similarly, if a query extracts both "location" and "location.country" from a variant, the latter can be derived from the former. Currently, SqueezeCache does not exploit these derivations; we leave compatibility-aware policies as future work.

6.4 Hydration Policy

Similar to cache policy, squeeze policy must handle workload changes. For example, one query may care about the year, but gradually more queries care about the month. The cache needs to dynamically adjust the squeezed data to better reflect the current workload. The process of increasing the size of the squeezed data is called hydration, similar to the promotion in the conventional cache.

Hydration is triggered on a "cache miss" for squeezed data: we read data from disk, or we find that a partially-evicted entry is not enough to answer the query, or we find that compressed data is critical enough that it is better to cache it uncompressed.

Similar to squeeze, hydration is also a spectrum: we can hydrate in one shot to full uncompressed data, or gradually hydrate step by step. SqueezeCache implements multiple hydration strategies, and users can choose the one that best fits their workload. By default, SqueezeCache uses a conservative strategy: it only hydrates to compressed data and never hydrates to uncompressed data, as we rarely see a batch of data that is so critical to justify the uncompressed memory size.

6.5 Discussion: When Squeeze is Not Beneficial

There are two types of squeezing: compression and partial eviction. Compression is always beneficial in our experiments because analytical data is usually compressible, and modern encoding techniques provide high compression ratios with efficient decoding. However, partial eviction is not always beneficial. The key trade-off is the remaining in-memory size versus how much I/O it can save. We discuss a few empirical cases where partial eviction is less beneficial:

- Data is already well-compressed. Partial eviction happens after compression (as a remedy for imperfect compression). If data is already near-perfectly compressed, partial eviction is unlikely to help. For example, if an integer array can be perfectly compressed using run-length encoding, then there is no room for partial eviction to make a difference. This occurs frequently in synthetic datasets (e.g., TPC-H, TPC-DS), where strings are simple categorical values and integers have narrow ranges. Real-world data is often more complex and diverse [57, 60, 61].
- Full projection is required. If the query actually needs the full data, partial eviction is unlikely to help. For example, in an ETL pipeline that transforms and rewrites data into another system/shape/format, partial eviction is unlikely to be beneficial.

For columns that are not suitable for partial eviction (which SqueezeCache can infer from lineage analysis plus a sample of the data), SqueezeCache skips partial eviction and caches the full data as compressed data.

7 EVALUATION

Our evaluation demonstrates three key findings:

- **Strong end-to-end performance:** On ClickBench, SqueezeCache reduces query latency by up to 22× and I/O by up to 12× compared to baselines (Section 7.3).
- **Superior cache efficiency:** SqueezeCache achieves 2–4× higher cache hit ratios than compression-only baselines across memory budgets (Section 7.4).
- **Mechanism versatility:** Squeezing benefits diverse operations—predicate evaluation, substring search, and projection expressions—by adapting squeezing to query lineage (Sections 7.5–7.7).

7.1 Implementation

We implemented SqueezeCache on top of LiquidCache [31] with 33k lines of Rust, including 14K for squeezable data types, 11K for lineage pushdown, and 7K for the squeeze policy. The project has engaged over 20 community contributors and is now merged into the main branch as the default eviction mechanism. Crucially, all techniques integrate via query optimization rules, requiring no modifications to downstream query engines.

We run the evaluation on an AMD 9900x CPU with 24 threads (12 cores), on NixOS with Linux kernel 6.12, equipped with PCIe5.0 NVMe SSD. Benchmarks are compiled with Rust 1.94.0-nightly (and also work with the latest stable Rust). For memory-constrained benchmarks, we use cgroups to limit memory usage. In addition to the benchmarks reported in this study, we continuously run performance evaluations (including ClickBench, TPC-H, TPC-DS, and JSONBench) on every commit to the main branch, and publish results in our open-source repository.

7.2 Baselines

We compare SqueezeCache with the following baselines: **Arrow:** Arrow caches inserted data as-is, without compression or partial eviction. This baseline represents the best choice for low latency queries today, e.g., BauPlan [58]. **LiquidCache:** LiquidCache [31] compresses data but does not perform partial eviction. This baseline represents the state-of-the-art compression-only cache.

7.2.1 Non-baselines. Eviction policies like LRU-K [49], SIEVE [71], and S3-FIFO [68] are orthogonal to SqueezeCache. Result caching, predicate caching, and materialized views are also excluded, as they are either special cases of SqueezeCache or they operate at different caching hierarchy and lack the cross-engine reuse required for lakehouse-wise data analytics.

7.3 End-to-end Evaluation

7.3.1 ClickBench. ClickBench [20] is 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 filter-heavy queries from Q20 to Q30. They include complex filter patterns and variable-length fields. These queries represent the target workload for SqueezeCache: scan-intensive operations over variable-length data where squeeze can retain query-relevant information. The remaining queries do not benefit from squeezing but perform similarly, incurring no slowdown.

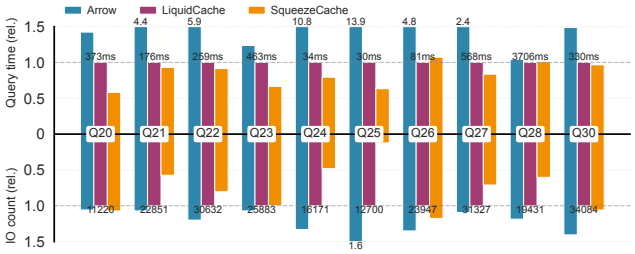

Figure 11: ClickBench overall performance comparison. Query time and I/O count comparison between SqueezeCache and the baselines. Values are normalized to the LiquidCache baseline. The I/O device is kernel page cache, so end-to-end query time mainly reflects in-memory processing time.

Figure 11 shows the overall performance comparison between SqueezeCache and the baselines. The x-axis shows query number. The y-axis shows query time (upward) and I/O count (downward). Values are normalized to the LiquidCache baseline. The values shown on LiquidCache are raw query time and I/O count. We deliberately run with kernel page cache so end-to-end query time mainly reflects in-memory processing time; otherwise, query time would be dominated by I/O. Since queries have dramatically different data sizes, we set the cache size of all variants to roughly half of the actual memory usage for LiquidCache. This ensures each query exercises both memory and disk access. We report I/O count (number of io_uring syscalls) as a separate metric because cloud systems price I/O by number of requests. Even when I/O latency is negligible, data-intensive systems still reduce I/O to save cost.

SqueezeCache performs best on 9 of 10 queries, reducing query time by up to 2× vs. LiquidCache and 22× vs. Arrow. The sole regression is within 5%. Regarding I/O, SqueezeCache leads on 7 of 10 queries, reducing counts by up to 8× vs. LiquidCache and 12× vs. Arrow. For the other 3, SqueezeCache matches LiquidCache and outperforms Arrow.

Not all queries improve on both I/O and end-to-end query time. Q20 and Q23 mainly benefit from faster in-memory processing with the same I/O count. They mostly benefit from squeezing to string fingerprints (Section 7.6), which reduces the number of rows to decode. Q21 and Q28 mainly benefit from I/O reduction, but overall query time does not improve much because (1) I/O is virtually free in this setup and (2) these queries spend substantial time on computation outside scanning (e.g., regex matching).

7.3.2 TPC-H and TPC-DS.. As discussed in Section 6.5, while TPC-H and TPC-DS are popular analytical benchmarks, they are less suitable for our workloads for two reasons. First, their synthetic data is simple and highly compressible [31], and multiple studies show it does not reflect real-world workloads [57, 60, 61]. Partial eviction is rarely necessary. Second, their queries are join-heavy and less scan-/I/O-intensive, so our gains are less pronounced than in scan-intensive, low-latency workloads. Nevertheless, SqueezeCache can gracefully fall back without introducing overhead by skipping partial eviction and only using compression to save memory.

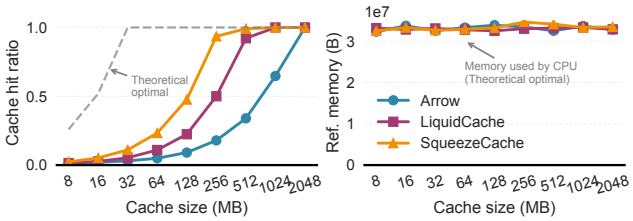

Figure 12: Cache size vs cache hit ratio (left) and actual referenced memory (right). Cache hit ratio is the number of batches cached in memory divided by the total number of batches. Actual referenced memory is LLC (last level cache) references × cacheline size. It approximates total memory traffic from memory to CPU.

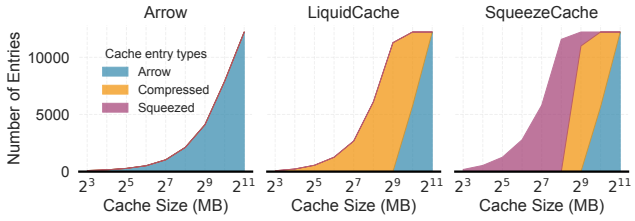

Figure 13: Cache entry composition comparison. Number of different types of entries in the cache. Arrow has only one type of entry, LiquidCache additionally has compressed entries, and SqueezeCache additionally has squeezed entries.

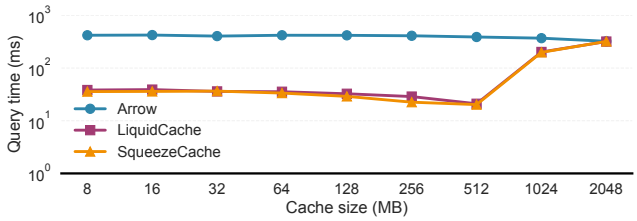

Figure 14: Query time comparison with kernel page cache as the I/O backend. Query time comparison between SqueezeCache and the baselines, with kernel page cache as the I/O device. Kernel page cache is a transparent cache mechanism provided by Linux kernel, a cache hit only costs a few microseconds.

7.4 Cache Dynamics

Having established the end-to-end benefits, we now analyze how SqueezeCache's cache efficiency varies with memory budget. We use ClickBench Q25, a representative query that exercises string predicate evaluation. Figure 12 shows the cache size vs cache hit ratio (left) and actual referenced memory (right). We compare SqueezeCache with two baselines: (1) Arrow, which caches inserted data as-is, and (2) LiquidCache, which compresses data but does not perform partial eviction. The x-axis sweeps cache size from 8MB to 2048MB on a log scale. The y-axis shows cache hit ratio (number of batches cached in memory divided by total batches) and actual referenced memory (LLC references × cacheline size, approximating total memory traffic from memory to CPU).

7.4.1 Overall results. SqueezeCache consistently outperforms LiquidCache by 2× in terms of cache hit ratio, and 4× compared to Arrow. All of the variants are able to cache all data at 2048MB cache size. Specifically, at 256MB cache size, SqueezeCache is able to cache 93.5% of data, while LiquidCache and Arrow are only able to cache 50.0% and 17.9% of data, respectively.

7.4.2 How far are we from optimal? SqueezeCache is a major step towards optimal cache utilization, but gaps remain. The gray dashed line shows the theoretical optimum, needing only 32MB—60× smaller than total data. We compute this from the right plot: CPU LLC (last level cache) references indicate memory-to-CPU traffic during query execution. SqueezeCache is suboptimal because it manages data in batches, not individual rows, with fixed prefix lengths per batch. Smaller batches could help, but at the cost of higher metadata overhead and fewer SIMD optimization opportunities.

7.4.3 How does the cache composition look like? In SqueezeCache, cached entries are not fixed: they can be Arrow, Compressed, or Squeezed. Figure 13 shows how their relative entry count changes with the cache size. Again, we compare SqueezeCache with LiquidCache and Arrow. The stacked area chart shows the relative entry count of each type of entries. By default, we use a simple squeeze policy that prioritizes squeezing less-squeezed entries (Arrow first, then Compressed, then Squeezed). This is evident in the figure: when cache size is small (e.g., 128MB), all three variants end up with a single entry type in the cache: Arrow, Compressed, and Squeezed, respectively. This happens because, under tight memory budgets, entries are squeezed to their most compact format. As the cache size increases, other types of entries are gradually possible to be cached in memory.

7.4.4 It's not all about IO. One might think these workloads are only bounded by I/O, and query time is mainly a function of the number of I/Os. However, as we will show here and in other studies [31], the in-memory data processing time is also a significant portion of the query time. Figure 14 shows query time comparisons between SqueezeCache and the baselines, with kernel page cache as the I/O backend. Kernel page cache is a transparent cache mechanism provided by Linux kernel, a cache hit only costs a few microseconds. With this almost ideal I/O backend, one might expect similar query times. However, SqueezeCache is still an order of magnitude faster than the Arrow baseline (when cache size is below 512MB). This is because the squeezed data in SqueezeCache is much more efficient to process, i.e., relevant data are closer to each other, leading to fewer CPU memory stalls. Similar effects also appear in LiquidCache, where its selective decompression significantly reduces memory stalls. After 512MB cache size, both SqueezeCache and LiquidCache start to accommodate Arrow entries. As a side effect, more queries are served from Arrow data, and query time increases and eventually matches the baseline.

7.5 Squeeze Helps Equality Predicates

We now examine *how* squeezing achieves the gains shown above. Using ClickBench Q25, we demonstrate that SqueezeCache's prefix squeezing retains sufficient information to resolve most selective predicates without disk I/O. This query finds the top 10 non-empty search phrases in alphabetical order—a pattern that appears ubiquitously in real-world queries.

	Arrow	Compressed	Squeezed	Evicted
LiquidCache	1	5850	N/A	6367
SqueezeCache	2	6	11556	654
Avg size/batch	298 KB	43.9 KB	23.1 KB	0

Table 1: Cache composition for ClickBench Q25 (256MB cache, 565MB data). Columns show batch counts per state. SqueezeCache squeezes 95% of entries vs. baseline's 52% eviction, with average entry sizes shown in the last row.

	# Read evicted (total)	# Read squeezed (total)
LiquidCache	6,367 (6,367)	N/A
SqueezeCache	654 (654)	239 (11,556)

Table 2: I/O operations for reading evicted vs. squeezed data (total batch reads in parentheses). For evicted data, every batch read is one I/O. For squeezed data, I/O occurs only if the squeezed data cannot resolve the predicate or full data is needed.

```
SELECT "SearchPhrase" FROM hits
WHERE "SearchPhrase" <> ''
ORDER BY "SearchPhrase"  LIMIT 10;
```

7.5.1 Is it a good fit for SqueezeCache? The predicate `SearchPhrase <> ''` works well with squeezable strings (Section 4.1): the prefix alone often decides the predicate. Although selectivity is only 13.1%, DataFusion also adds a dynamic filter [11] of the form `SearchPhrase >= X` based on current top-10 results. This filter allows SqueezeCache to prune batches using squeezed prefixes, benefiting queries without explicit filters.

7.5.2 Cache composition. The on-disk data for this query is about 565MB; we set cache size to 256MB. Table 1 shows the cache composition after query execution. SqueezeCache caches data at batch granularity (8,192 rows), with each entry in one of four states: Arrow (uncompressed), Compressed (using LiquidArray [31]), Squeezed (using techniques from this paper), or Evicted (on disk). Entries start as Arrow and are progressively squeezed as memory pressure increases (Section 6.1).

In the baseline, roughly half of cached entries are evicted to disk, and the other half are cached as compressed data. In SqueezeCache, 95% of cached entries are squeezed (partially evicted while keeping essential data in memory). This is the intended behavior: by squeezing each entry, SqueezeCache can keep more entries in cache. The last row shows the average size of the cached entries, with Arrow being the largest at 298 KB per entry, compression making it 7x smaller at 43.9KB, squeezing it makes it almost 2x smaller again at 23.1KB. With the same cache space, caching squeezed data can accommodate roughly twice as many entries in memory. If those squeezed entries can answer the filter predicate, then the query can be evaluated without reading the entire data from disk.

7.5.3 Does squeezing help predicate evaluation? This query has two filters: the user-specified `SearchPhrase <> ''` and dynamically generated `SearchPhrase >= X`. Squeezed prefixes can answer both, though the dynamic filter is more selective. Table 2 shows the I/O breakdown. The baseline reads all 6,367 evicted batches. SqueezeCache reads only 654 evicted batches, plus 239 of 11,556 squeezed batches (2.1%) that require full data to resolve ambiguity.

Figure 15 shows the 37 dynamic filters generated during query execution. We track three metrics: *Selectivity* is the percentage of rows passing the filter; below 10^{-4} means fewer than 1 row per 8,192-row batch is expected to pass. *Ambiguity* shows the percentage of rows that could be pruned with the full data but can not be pruned with squeezed data. *Batches evaluated* shows how many batches the query engine evaluates with each filter.

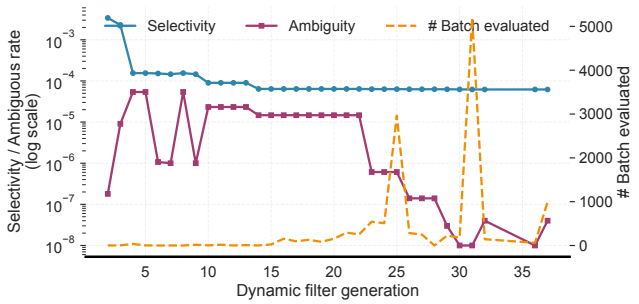

Figure 15: Dynamic filter generations, with their selectivity and ambiguous rate. Selectivity is the percentage of rows that pass the filter. Ambiguity shows the percentage of rows that can not be pruned with squeezed data. As query progresses, more selective filters are generated and ambiguity drops.

As shown in Figure 15, as the query engine updates filters, selectivity drops and stabilizes around 0.6×10^{-4} after 14 filters. This corresponds to 203 batches that require full data reads (final projection). The ambiguity rate is consistently much lower than the selectivity and drops to as low as 4×10^{-8}. The dataset has 100M rows, indicating that only 4 rows (expected) are ambiguous during runtime, and that translates to up to 4 IOs.

Next, we analyze prefix length sensitivity. SqueezeCache defaults to a prefix length of 7 bytes, but other lengths are possible. We measure *differentiability*: the fraction of unique values captured by the prefix. Higher differentiability means the prefix alone can distinguish more values, reducing I/Os needed to resolve predicates. Figure 16 shows the differentiability of the squeezed data for each prefix length. It shows three string columns: SearchPhrase, URL, and Title. The SearchPhrase represents human input string, URL is structured string, and Title is between the two. SearchPhrase is the most differentiable column, and URL is the least differentiable column. Note that the differentiability is computed after we exclude the common prefix of the batch. In general, longer prefix length has higher differentiability, but at the cost of higher storage overhead. SqueezeCache's default prefix length of 7 balances differentiability, storage overhead, and memory alignment.

7.6 Squeeze Helps Substring Predicates

SqueezeCache adapts its squeeze strategy based on query patterns. Prefixes are ineffective for substring searches in ClickBench Q20 which finds how many urls contain the substring `'google'`

```
SELECT COUNT(*) FROM hits WHERE "URL" LIKE '%google%';
```

SqueezeCache uses fingerprints in this case to reduce memory required for caching. To evaluate the effectiveness of fingerprints, we ran Q20 using the same setup as the previously (Baseline) and

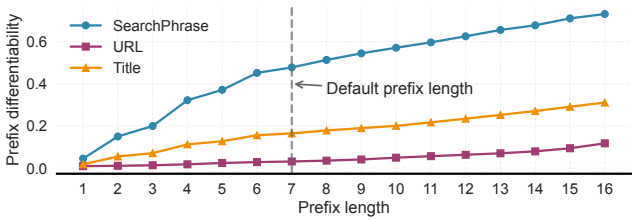

Figure 16: Prefix length sensitivity analysis. Differentiability is computed as the number of unique values captured by the prefix, divided by the total number of unique values in the batch.

	IO	Decode rows (time)	Query time
LiquidCache	11450	19M (6.8 s)	380 ms
SqueezeCache	11981	10M (3.0 s)	191 ms

Table 3: Fingerprint performance on ClickBench Q20. IO: batch reads. Decode rows: rows decompressed (CPU time in parentheses). Fingerprints reduce decoded rows by 47%, halving query latency.

configured SqueezeCache to use column lineage analysis to automatically squeeze to string fingerprints, configured to use 32 buckets (i.e., 4 bytes per string) with a simple round-robin bucket assignment. The results are shown in Table 3. The I/O counts are similar (fingerprints do not reduce I/O), but SqueezeCache requires slighly more IO due to fingerprint overhead. When reading from fast NVMe drives, decoding often dominates query time [31] as is the case in our experiments. Even though the same I/O is performed, SqueezeCache significantly reduced the number of decoded rows from 19M to 10M (6.8 seconds to 3.0 seconds), resulting in a 57.4% reduction in overall query time. Fingerprints improve performance even when I/O count stays the same due to SqueezeCache's row-level decompression (inherited from LiquidCache) allows skipping individual rows rather than entire batches.

Fingerprints filter out 57.7% of the string values for this query and configuration. For the remaining 42.3% of the string values, 99.97% are false positives (do not contain the substring "google"). Better assignment strategies for fingerprints exist [56], which we plan to pursue in future work, but even the simple round-robin assignment is effective.

7.6.1 Fingerprint overhead analysis. Figure 17 sweeps bucket sizes from 4 to 64; more buckets reduce collisions and improve filter-out rates, at 1 bit per string per bucket. SqueezeCache uses 32 buckets by default. Both test patterns ("google" and "database") have near-zero occurrence, so higher filter-out rates indicate more effective fingerprints. SearchPhrase in general is more fingerprint-friendly than URL and Title, potentially because it is much shorter than the other two. We also profiled and optimized fingerprint creation and querying through compiler auto-vectorization, making both processes as fast as string scanning. In summary, this case study demonstrates that SqueezeCache's lineage analysis allows it to adapt to diverse workloads (e.g., substring search) and automatically apply the appropriate squeeze strategy.

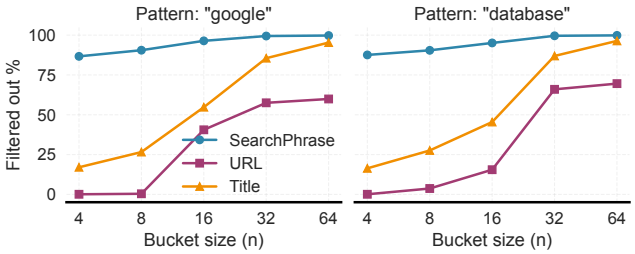

Figure 17: Fingerprint bucket size vs filter effectiveness. The bucket size is the number of buckets used to store the fingerprint.

JSON object	Referenced by query	Squeezed
4.16 MB	0.16 MB	0.09 MB

Table 4: Average batch size comparison. JSON object shows the size of the JSON object in the table. Referenced by query shows the size of the JSON fields referenced in the query. Squeezed shows the size of the squeezed data.

7.7 Squeeze Identifies Sub-Fields

Now we study a new category of squeezing: sub-field extraction. We consider two workloads: (1) nested-JSON-like extraction and (2) datetime extraction. JSON (and other semi-structured data) extraction is a fundamental pillar of semi-structured data processing. Existing systems either disallow direct queries on semi-structured data until it is transformed into queryable data through an ETL pipeline, or require ad-hoc annotations or pre-computed JSON-specific indexes/subsystems to accelerate JSON queries [26]. SqueezeCache naturally supports direct JSON queries through squeeze: lineage analysis captures which fields are used, and the cache squeezes and caches those fields in memory. Datetime extraction is another common example that benefits from squeezing, but it is usually not worth building a special-purpose system for. It fits naturally into SqueezeCache's framework.

We optimize both SqueezeCache and Parquet's Variant support, with our changes upstreamed [30]. All data is pre-shredded to isolate caching effects from runtime extraction overhead. We analyze JSONBench Q1 [21], which counts the number of events in the "bluesky" table, grouped by the "event" field.

```
SELECT data['commit.collection'] AS event, COUNT(*) AS count
FROM bluesky GROUP BY event ORDER BY count DESC;
```

Table 4 shows the size of the JSON object in the table, the size used in the query, and the size of the squeezed data. The original JSON object is 4.16 MB, while the query only references 0.16 MB. The squeezed data is what SqueezeCache caches: it automatically extracts used fields and applies cascading encoding into LiquidArray [31], yielding 0.09 MB per batch—a 46× reduction in size.

Table 5 shows runtime performance: the baseline caches only 2.5% of data, while SqueezeCache caches 100% in memory with 0 I/O, yielding 3.2× lower query latency. Notably, SqueezeCache generalizes JSON-specific optimizations—it supports efficient semi-structured queries automatically without special handling.

Next, we study a less common optimization that benefits from the same framework: datetime extraction. This is a real-world stackoverflow query that counts the number of posts by the day of the week. We run this query on the StackOverflow math dataset, which is

	# IO	Cache ratio	Query time
LiquidCache	120	2.5%	49 ms
SqueezeCache	0	100%	15 ms

Table 5: Json bench runtime performance comparison. # IO shows the number of IOs caused by reading evicted data and reading squeezed data. Cache ratio shows the ratio of the cached data to the total data.

	Arrow	Compressed	Squeezed	Evicted
LiquidCache	2	258	N/A	235
SqueezeCache	3	240	252	0

Table 6: Cache composition comparison. Number of cached batches, each batch consists 8192 rows.

Arrow	Compressed	Year	Month	Day	DoW
24 MB	9.0 MB	2.2 MB	3.0 MB	3.7 MB	2.3 MB

Table 7: Date time squeezed size comparison. Arrow shows the size of the Arrow format. Compressed shows the size of the LiquidArray format. Year, Month, Day, DoW show the size of the year, month, day, and day-of-week components (squeezed).

roughly 12GB after compression, again, we set the memory budget to roughly half of the actual memory usage. Table 6 shows the cache composition comparison between the baseline and SqueezeCache. With the same memory budget, baseline can cache 260 batches in memory, and 235 batches are required to be read from disk. SqueezeCache can cache all 295 batches in memory, with no IO required.

```
SELECT EXTRACT(DOW FROM "CreationDate") AS day_of_week,
  COUNT(*) AS post_count
FROM "Posts" GROUP BY day_of_week;
```

Table 7 further explains their performance difference. Baseline only caches the full datetime column (Arrow or compressed LiquidArray), which is wasteful when the query only uses (for example) year. LiquidArray can effectively compress the datetime column by 2.7×, allowing much more data to be cached. The squeezed data can further reduce the memory usage. Here we showcase four commonly extracted date components: year, month, day, and day-of-week. They further reduce memory usage by 4.1×, 3.0×, 2.4×, and 3.9×, respectively.

Similar optimizations also apply to StructArray extraction, where a query uses only a sub-column but the query engine still needs to read and decode the entire struct column. SqueezeCache automatically extracts relevant columns, applies cascading encoding, and caches them in memory. Due to space limits, we omit detailed evaluations here. Overall, these experiments demonstrate that SqueezeCache effectively generalizes to complex types (JSON and Datetime), automatically identifying and retaining only the minimal semantic components required by the workload.

8 RELATED WORK

Caching is a well-studied topic in systems and databases. The **eviction policy** determines which data to keep. Recent work has proposed many advanced policies beyond LRU and ARC [44, 49], including machine learning-based approaches such as GL-Cache [67] and Baleen [65], and simplified but efficient algorithms such as S3-FIFO [68] and SIEVE [71]. SqueezeCache is orthogonal to these policies; it improves the *utility* of the cached data itself, allowing any eviction policy to store more effective entries.

The **cache mechanism** determines how data is stored. Traditional systems use buffer pools or OS page caches [8, 22]. Modern tiered memory systems [32, 38, 43] extend capacity using CXL or NVMe. Disaggregated caching systems like Crystal [25] and Alluxio [4] manage data in remote storage, while InfiniCache [64] exploits ephemeral serverless functions to build cost-effective memory caches. Semantic caching [39, 53] leverages query semantics to cache partial results. SqueezeCache extends the idea of semantic caching with *squeezing*, a lossy lineage-aware compression mechanism that bridges the gap between full caching and eviction.

Cloud-native data warehouses such as Snowflake [7, 57], Redshift [6, 60] and BigQuery [45] separate compute from storage. This disaggregation makes I/O the bottleneck [27, 47] and necessitates effective caching (e.g., Photon [12], Velox [50]). Instance-optimized data layouts [24] adapt storage organization to workload patterns, while cloud functions [15, 58] can serve as accelerators for elastic data analytics. Computation pushdown [17, 66, 69] reduces I/O by filtering data at storage. SqueezeCache extends the pushdown paradigm into the cache layer, allowing the cache to serve as a query accelerator by retaining only the data needed for common predicates (e.g., string prefixes or extracted dates).

Efficient data representation is the key to analytic system performance. Columnar formats such as Parquet [62] and Arrow [37] are standard. Compression techniques have evolved from general-purpose (Snappy, Zstd) to specialized encodings: FSST [16] for strings, ALP [2], Chimp [40], and Gorilla [51] for floating-point numbers, and FastLanes [1] and BtrBlocks [36] for integers. MorphStore [23] takes a holistic approach by enabling compression-aware query processing throughout the entire query engine. Beyond compression, data-skipping techniques like Sieve [59] use learned indexes to efficiently skip irrelevant data, and workload-aware column imprints [55] adapt indexes to query patterns. SqueezeCache leverages these lightweight encodings for its "compressed" state but goes further with "squeezing"—using lossy representations (quantization, fingerprints [56]) to trade precision for capacity when memory is scarce.

9 CONCLUSION

We presented SqueezeCache, a novel caching mechanism that challenges the binary decision of traditional eviction policies. By recognizing that queries often require only a fraction of the data they touch, SqueezeCache introduces the concept of "squeezing"—partially retaining critical data segments like string prefixes or extracted date components while evicting the rest. This lineage-aware approach enables SqueezeCache to maintain higher cache effective capacity and serve more queries from memory. Our evaluation demonstrates that SqueezeCache significantly outperforms existing baselines in cache hit ratio and end-to-end latency, proving that fine-grained, semantic-aware data management is key to unlocking the next generation of performance for disaggregated cloud analytics.

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] 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.

[8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces* (1.00 ed.). Arpaci-Dusseau Books.

[9] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2023. *Operating Systems: Three Easy Pieces* (1.10 ed.). Arpaci-Dusseau Books.

[10] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. 2017. Recache: Reactive caching for fast analytics over heterogeneous data. *Proceedings of the VLDB Endowment* 11, 4 (2017), 324–337.

[11] Adrian Garcia Badaracco and Andrew Lamb. 2025. *Dynamic Filters: Passing Information Between Operators During Execution for 25x Faster Queries.* Apache DataFusion. https://datafusion.apache.org/blog/2025/09/10/dynamic-filters/ Blog post.

[12] 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.

[13] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.

[14] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 753–768.

[15] Haoqiong Bian, Tiannan Sha, and Anastasia Ailamaki. 2023. Using Cloud Functions as Accelerator for Elastic Data Analytics. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 161:1–161:27. doi:10.1145/3589306

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

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

[18] Zhichao Cao et al. 2022. Making Cache Monotonic and Consistent. *Proceedings of the VLDB Endowment* 16, 4 (2022), 891–904.

[19] Jianjun Chen, Rui Shi, Heng Chen, Li Zhang, Ruidong Li, Wei Ding, Liya Fan, Hao Wang, Mu Xiong, Yuxiang Chen, Benchao Dong, Kuankuan Guo, Yuanjin Lin, Xiao Liu, Haiyang Shi, Peipei Wang, Zikang Wang, Yemeng Yang, Junda Zhao, Dongyan Zhou, Zhikai Zuo, and Yuming Liang. 2023. Krypton: Real-time Serving and Analytical SQL Engine at ByteDance. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3528–3542. doi:10.14778/3611540.3611545

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

[21] ClickHouse contributors. 2025. JSONBench: a Benchmark For Data Analytics On JSON. https://github.com/ClickHouse/JSONBench. Open-source benchmark comparing native JSON support across analytical databases.

[22] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *CIDR 2022, Conference on Innovative Data Systems Research*.

[23] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *Proceedings of the VLDB Endowment* 13, 11 (2020), 2396–2410.

[24] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 418–431. doi:10.1145/3448016.3457270

[25] 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.

[26] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON tiles: Fast analytics on semi-structured data. In *Proceedings of the 2021 International Conference on Management of Data*. 445–458.

[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] Brad Glasbergen, Kyle Langendoen, Michael Abebe, and Khuzaima Daudjee. 2020. ChronoCache: Predictive and Adaptive Mid-Tier Query Result Caching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2391–2406. doi:10.1145/3318464.3380593

[29] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-tree: A modern read-write-optimized concurrent larger-than-memory range index. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3442–3455.

[30] Xiangpeng Hao and contributors. 2025. Improve "variant_get" performance on a perfect shredding. https://github.com/apache/arrow-rs/pull/8887. Pull request 8887, merged into arrow-rs main branch.

[31] Xiangpeng Hao, Andrew Lamb, Yibo Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2025. LiquidCache: Efficient Pushdown Caching for Cloud-Native Data Analytics. *Proc. VLDB Endow.* 18, 13 (2025), 5662–5675. doi:10.14778/3773731.3773741

[32] 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.

[33] Theodore Johnson and Dennis Shasha. 1994. X3: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th VLDB Conference*. 439–450.

[34] Tao Kong, Hui Li, Yuxuan Zhao, Liping Li, Xiyue Gao, Qilong Wu, and Jiangtao Cui. 2025. STsCache: An Efficient Semantic Caching Scheme for Time-series Data Workloads Based on Hybrid Storage. *Proceedings of the VLDB Endowment* 18, 9 (2025), 2964–2977. doi:10.14778/3746405.3746421

[35] Adarsh Kumar et al. 2025. Linear Elastic Caching via Ski Rental. In *CIDR*.

[36] 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.

[37] 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.

[38] 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.

[39] J Li et al. 2025. STsCache: An Efficient Semantic Caching Scheme for Time-series Data Workloads Based on Hybrid Storage. *Proceedings of the VLDB Endowment* 18 (2025).

[40] 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.

[41] Jian Liu, Kefei Wang, and Feng Chen. 2021. TSCache: An Efficient Flash-based Caching Scheme for Time-series Data Workloads. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3253–3266. doi:10.14778/3484224.3484225

[42] J Lu et al. 2025. DEX: Scalable Range Indexing on Disaggregated Memory. *Proceedings of the VLDB Endowment* 17 (2025), 2603–2616.

[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] Nimrod Megiddo and Dharmendra S Modha. 2003. {ARC}: A {Self-Tuning}, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*.

[45] 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.

[46] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. *Proceedings of the VLDB Endowment* 14, 5 (2021), 771–784.

[47] 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.

[48] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. HetCache: Synergising NVMe Storage and GPU acceleration for Memory-Efficient Analytics. In *Proceedings of CIDR*.

[49] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

[50] 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.

[51] 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.

[52] U Sabarwal et al. 2023. HetCache: Synergising NVMe Storage and GPU Acceleration for Memory-Efficient Analytics. In *CIDR*.

[53] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate caching: Query-driven secondary indexing for cloud data warehouses. In *Companion of the 2024 International Conference on Management of Data*. 347–359.

[54] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. 2023. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 675–691. doi:10.1145/3600006.3613144

[55] Noah Slavitch. 2020. Workload-Aware Column Imprints. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2865–2867. doi:10.1145/3318464.3384411

[56] Mihail Stoian, Johannes Thürauf, Andreas Zimmerer, Alexander van Renen, and Andreas Kipf. 2025. Instance-Optimized String Fingerprints. *arXiv preprint arXiv:2507.10391* (2025).

[57] Jan Vincent Szlang, Sebastian Bress, Sebastian Cattes, Jonathan Dees, Florian Funke, Max Heimel, Michel Oleynik, Ismail Oukid, and Tobias Maltenberger. 2025. Workload Insights from the Snowflake Data Cloud: What Do Production Analytic Queries Really Look Like? *Proceedings of the VLDB Endowment* 18, 12 (2025), 5126–5138.

[58] Jacopo Tagliabue, Tyler Caraza-Harter, and Ciro Greco. 2024. Bauplan: zero-copy, scale-up faas for data pipelines. In *Proceedings of the 10th International Workshop on Serverless Computing*. 31–36.

[59] Yulai Tong, Jiazhen Liu, Hua Wang, Ke Zhou, Rongfeng He, Qin Zhang, and Cheng Wang. 2023. Sieve: A Learned Data-Skipping Index for Data Analytics. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3214–3226. doi:10.14778/3611479.3611520

[60] 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.

[61] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get real: How benchmarks fail to represent the real world. In *Proceedings of the Workshop on Testing Database Systems*. 1–6.

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

[63] 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.

[64] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 267–281.

[65] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. 2024. Baleen: ML Admission & Prefetching for Flash Caches. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 347–371.

[66] 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.

[67] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2023. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 115–130.

[68] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 130–149.

[69] 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.

[70] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. 2022. CompuCache: Remote Computable Caching using Spot VMs. In *Proceedings of CIDR*.

[71] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and KV Rashmi. 2024. {SIEVE} is simpler than {LRU}: an efficient {Turn-Key} eviction algorithm for web caches. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1229–1246.

[72] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2023. Two is better than one: The case for 2-tree for skewed data sets. *memory* 11 (2023), 13.