

White-Box Caching for Cloud Data Systems

By
Xiangpeng Hao

A dissertation submitted in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy
(Computer Sciences)
at the
UNIVERSITY OF WISCONSIN-MADISON
2026

Date of final oral examination: 06/19/2026

The dissertation is approved by the following members of the Final Oral Committee:

Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Ming Liu, Assistant Professor, Computer Sciences
Yiqiao Zhong, Assistant Professor, Statistics

To system builders.

*“Let us first of all and before all be kind, then honest, and then—
let us never forget one another.”*

— Fyodor Dostoevsky, *The Brothers Karamazov*

Acknowledgments

Advisors

I want to thank Andrea and Remzi for their support and guidance, especially in the middle of my PhD, when I had to switch advisors. Who would be my next advisor? That was the question I had to answer by December 2024, or I would have quit and started enjoying life (too soon). Changing advisors is not easy, as it is not clear, going in, that your new advisor is actually any different from your old one. Luckily, they were different. It is not that systems researchers are better than database researchers, but rather that exceptional human beings are better than average ones.

What surprised me most about Andrea and Remzi was their deep belief that academic research should advance human knowledge. They live up to this belief by caring about scientific rigor and the craft of research. Working with them allowed me to see the best of what academia has to offer, and convinced me that NSF actually makes good use of taxpayers' money.

Mentors

I came to know Andrew Lamb when I was lost in database research. I felt that what I was doing was purely imaginary and ungrounded, and I decided that for the rest of my PhD I needed to work on a real system used by real people. But which system? I surveyed a few options, from Postgres to DuckDB, and settled on DataFusion for two reasons. First, it is a Rust project, and it is clear that Rust projects will replace every C/C++ project over time—I did not want to be on the wrong side of history. Second, the DataFusion community is surprisingly active and friendly, and its leader, Andrew Lamb, has a kind of leadership that is rare in open source. So I wrote Andrew an email asking whether he would take me on as a summer intern at InfluxData. At that point I had never contributed a single line of code to DataFusion, and I had little hope that he would say yes. But it worked out, and it became the happiest summer of my PhD—I got to work on a real system, do real research, and meet many wonderful people in the community.

Andrew also made it possible for me to meet many other remarkable people in the DataFusion community, including Jacopo Tagliabue and Will Manning, who run their own companies and were generous enough to sponsor my research, alongside the amazing support of InfluxData. Jacopo Tagliabue is the CTO of Bauplan; his passion for building correct-by-design analytical systems has changed my perspective on what a robust system should be. Will Manning is the CEO of SpiralDB; his vision for the fastest data format—and his ability to execute on it—has inspired me to build the systems I always wanted to build. The three of them not only sponsored my research, but also met with me regularly to discuss ideas and keep me grounded in the real world.

I am grateful to the broader DataFusion/Arrow/Parquet communities, which keep me humble. I was surprised to find so many truly smart people who had brilliant ideas and were able to execute on them in real systems. The only difference is that they are not in a PhD program. I questioned myself: what is the difference between a researcher and an engineer, especially in systems research? That is a question whose answer goes beyond the scope of a PhD dissertation.

Finally, I thank Tianzheng Wang, who first introduced me to data systems research as an undergraduate and who has supported me throughout my PhD. I also thank my mentors Badrish Chandramouli at Microsoft Research and Yixin Luo at Google. My internships under their guidance taught me how research is done outside the university, and significantly improved my financial situation through their generous stipends.

Sponsors

A PhD student often costs less than one-fifth of a full-time software engineer, yet it is still hard to find companies willing to fund us (especially systems-ish researchers). The exceptions are three truly brilliant companies willing to take that risk and trust a PhD student to deliver something that might not even work: **InfluxData**, **Bauplan**, and **SpiralDB**. Let's take a moment to thank them—and to try their products, if you haven't already.

- InfluxData builds InfluxDB, a leading time-series database, and is the home of much of the DataFusion engine this dissertation builds upon.
- Bauplan builds a serverless data platform for analytics and pipelines, closely related to the white-box caching ideas in this work.

- SpiralDB builds Vortex, a “better” Parquet file format, closely related to the format design ideas in this dissertation.

I also thank the University of Wisconsin-Madison and the taxpayers who fund it. My parents never got the chance to go to college, and they never imagined that I would earn a PhD without bankrupting the family.

Life

I thank my friends in the systems and database groups at UW-Madison. Listing all of your names here would make the thesis 200 pages longer; with high probability, if you are reading this, you are acknowledged.

I read books too. *The Brothers Karamazov* has the biggest impact on my life; it changed how I see the world and how I act in it. And of course, *Operating Systems: Three Easy Pieces*[15] not only taught me how to build operating systems, but also how to teach, highly recommended.

Contents

Acknowledgments	iii
Advisors	iii
Mentors	iii
Sponsors	iv
Life	v
Abstract	xxi
1 Introduction	1
1.1 Caching: The Hidden Foundation	2
1.2 The Challenges of Caching in Modern Data Systems	3
1.3 White-Box Caching	4
1.4 Contributions and Outline of Chapters	7
1.4.1 LiquidCache: Pushdown Caching for Cloud-Native Analytics	8
1.4.2 SqueezeCache: Curating Cache Content Through Lineage	9
1.4.3 Efficient I/O and Bf-Tree	10
1.4.4 Engineering a White-Box Cache	10
1.4.5 Related Work and Conclusion	11
2 General Background	12
2.1 Life of a query in cloud-native analytics	13
2.1.1 Data representation on disk and in memory	13
2.1.2 Parsing and logical planning	15
2.1.3 Logical optimization and pushdown	15
2.1.4 Physical planning	16
2.1.5 Scan and decode	16
2.1.6 Vectorized execution	16
2.1.7 Returning results	17
2.2 Caching architectures	17
2.2.1 Private cache	18
2.2.2 Distributed cache	18

2.2.3	Disaggregated cache	19
2.2.4	Semantic and partial caches	19
2.2.5	Eviction policies	20
2.3	Cost model every system researcher should know	20
2.3.1	Decomposing the hierarchy: interconnect × medium	21
2.3.2	Bill of materials and operating cost	23
2.3.3	Reading the numbers	25
2.4	Three layers of opacity, revisited	25
3	The White-Box Cache:	
	A Reference Design	27
3.1	A reference architecture	28
3.2	Information flow: from optimizer to cache	29
3.3	Cache fidelity as a spectrum	30
3.4	Composing the three axes	31
3.5	Non-goals	32
3.6	From the framework to the chapters	32
4	LiquidCache:	
	Adapting Cache Representation to the Workload	34
4.1	Background	36
4.1.1	Filter pushdown on Parquet	37
4.1.2	Target workloads: low-latency analytics	37
4.2	Architecture	37
4.2.1	System components	38
4.2.2	Life of a query	39
4.2.3	What to push down?	41
4.2.4	Local mode	41
4.2.5	Cache mechanisms	41
4.3	Liquid Format	42
4.3.1	Dilemma: Decoding speed vs. memory usage	42
4.3.2	Liquid data representation	43
4.3.2.1	Encode strings.	44
4.3.2.2	Encode integers.	44
4.3.2.3	Encode floating numbers.	45

4.3.2.4	Lossy encoding.	45
4.3.3	Co-design with filter evaluation.	45
4.3.3.1	Selective decoding.	46
4.3.3.2	Late filter materialization.	47
4.3.3.3	Evaluate filters on encoded data.	48
4.3.3.4	Evaluate filters on partially encoded data.	48
4.3.4	Efficient transcoding.	48
4.3.4.1	On demand fine-grained transcoding.	49
4.3.4.2	Background transcoding.	49
4.3.4.3	Deep integration with Parquet reading.	50
4.3.5	Discussion.	50
4.4	Evaluation.	51
4.4.1	Implementation details.	51
4.4.2	Evaluation setup.	52
4.4.3	Baseline implementations.	53
4.4.4	Overall results.	55
4.4.4.1	Latency.	55
4.4.4.2	Network traffic.	55
4.4.4.3	CPU time on cache server.	56
4.4.4.4	Cache memory consumption.	56
4.4.5	TPC-H results.	56
4.4.6	Decoding revisited.	58
4.4.7	Transcoding cost.	59
4.4.8	Cold run latency.	61
4.4.9	Ablation study.	62
4.4.10	Cache dynamics.	63
4.4.11	Cases for non-pushdown.	64
4.5	Related work.	64
4.6	Future work.	65
4.7	Conclusion.	65
5	SqueezeCache:	
	Curating Cache Content Through Lineage.	67
5.1	Background.	70

5.1.1	The nature of strings in real-world data	70
5.1.2	Relationship to semantic-cache mechanisms	71
5.2	Overview	72
5.3	Squeezable Data Layouts	74
5.3.1	Squeezing Strings	74
5.3.1.1	Droppable string buffer.	75
5.3.1.2	Single string buffer.	75
5.3.1.3	Optimistic string length.	75
5.3.1.4	String fingerprint.	75
5.3.1.5	Optional prefix and fingerprint.	76
5.3.2	Squeezing Integers	76
5.3.3	Squeezing Decimals	77
5.3.4	Squeezing Floating Numbers	77
5.3.5	Squeezing DateTimes	78
5.3.6	Squeezing Variants (JSON)	79
5.4	Lineage pushdown	80
5.4.1	Lineage analysis	81
5.4.2	Handling Data Type Mismatches	82
5.4.3	Tracking Squeeze Metadata	83
5.5	Squeeze policy	83
5.5.1	Squeeze Spectrum	84
5.5.2	Selection Policy	85
5.5.3	Retention Policy	85
5.5.4	Hydration Policy	86
5.5.5	Discussion: When Squeeze is Not Beneficial	86
5.6	Evaluation	87
5.6.1	Implementation	88
5.6.2	Baselines	88
5.6.2.1	Non-baselines	89
5.6.3	End-to-end Evaluation	89
5.6.3.1	ClickBench.	89
5.6.3.2	TPC-H and TPC-DS.	90
5.6.4	Cache Dynamics	91

5.6.4.1	Overall results.	91
5.6.4.2	How far are we from optimal?	91
5.6.4.3	How does the cache composition look like?	92
5.6.4.4	I/O is not the only cost	93
5.6.5	Squeeze Helps Equality Predicates	94
5.6.5.1	Is it a good fit for SqueezeCache?	94
5.6.5.2	Cache composition	95
5.6.5.3	Does squeezing help predicate evaluation?	95
5.6.6	Squeeze Helps Substring Predicates	97
5.6.6.1	Fingerprint overhead analysis	98
5.6.7	Squeeze Identifies Sub-Fields	99
5.7	Related work	102
5.8	Conclusion	102
6	Bf-Tree:	
	Decoupling Cache Granularity from Disk	104
6.1	Background	108
6.1.1	B-Tree	108
6.1.2	LSM-Tree	109
6.1.3	Modern NVMe SSD	110
6.2	Bf-Tree Architecture	111
6.2.1	Mini-page	112
6.2.2	Mini and leaf page layout	114
6.2.3	Leaf pages and mapping table	115
6.2.4	Inner nodes	115
6.2.5	Performance optimizations	116
6.3	Buffer pool for mini-pages	117
6.3.1	Memory regions of circular buffer	118
6.3.2	Circular buffer API	118
6.3.3	Performance optimizations	119
6.4	Bf-Tree core operations	120
6.4.1	Get	120
6.4.2	Insert	122
6.4.3	Range scan	123

6.4.4	Delete/Update	123
6.4.5	Mini-page operations	123
6.4.6	Handling negative search	124
6.4.7	Snapshotting, logging, and recovery	125
6.5	Evaluation	126
6.5.1	Experimental setup	126
6.5.2	Baselines	127
6.5.3	Overall performance	128
6.5.4	Scalability	129
6.5.5	Latency	131
6.5.6	Skewness	132
6.5.7	Read write ratio.	133
6.5.8	Cache sensitivity	134
6.5.9	Time-component analysis	135
6.5.10	Copy-on-access region size	136
6.5.11	Impact on promotion rate	136
6.6	Closing the three-axis story	137
7	Engineering Lessons from Building White-Box Caches	138
7.1	A modular core for an invasive technique	139
7.2	Owning the I/O path without imposing it	140
7.2.1	Async I/O in the Rust ecosystem	141
7.2.2	A runtime-agnostic interface	142
7.2.3	The I/O-thread model	142
7.2.4	Reimplementing a slice of the filesystem	143
7.3	Inheriting the OS's correctness burden	144
7.4	Staying equivalent to the format you replaced	145
7.5	The metadata cost of fine granularity	146
7.6	Returning the generic core upstream	146
7.7	Lessons learned	147
8	Related Work	149
8.1	Caching architectures and policies	149
8.2	Pushdown and semantic caching	150
8.3	Data representation and encoding	151

8.4	Resource disaggregation	151
8.5	Larger-than-memory range indexes	152
8.6	Modern hardware and I/O	153
9	Conclusion	154
9.1	Summary	154
9.2	Cross-cutting findings	156
9.3	Limitations	156
9.4	Future work	157
9.5	Closing remarks	158
	Bibliography	160

List of Tables

Table 2.1	Cost components of a cloud-native data system, separated into interconnects (top) and media (bottom). The latency and bandwidth of any deployment are bounded by whichever of an (interconnect, medium) pair is slower or narrower. BOM is the rough bill-of-materials (silicon plus assembly) at retail-OEM volumes; “bundled” components are integrated into the CPU or motherboard and not separately billed.	22
Table 4.1	Characteristics of selected ClickBench queries – Each query is described by its ID, number of columns projected, total data size processed, number of filters applied, selectivity (percentage of rows that pass the filters), and the data types involved. Q20 originally projects all 104 columns but is optimized to a simple count after projection and aggregation pushdown.	52
Table 4.2	Ablation study based on a variation of Q22. Eval on encoded data shows more than 98% improvement over baseline Parquet (pushdown).	63
Table 4.3	LiquidCache latency following different operations. Operations are executed in the order of the table. The bolded filters can not be evaluated on encoded data and must be decoded first.	63
Table 5.1	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.	95
Table 5.2	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.	96
Table 5.3	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.	98

Table 5.4	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.	100
Table 5.5	JSONBench 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.	100
Table 5.6	Cache composition comparison. Number of cached batches, each batch consists 8192 rows.	101
Table 5.7	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).	101
Table 6.1	Four types of records in a mini-page.	125

List of Figures

- Figure 1.1 Three axes of opacity that conventional caches accept and that the White-Box caches in this thesis narrow. Each contribution attacks one axis: Liquid-Cache makes the in-cache *encoding* an adaptive, workload-driven choice; SqueezeCache decides what *content* within each cached unit to keep; and Bf-Tree decouples cache *granularity* from disk granularity. A conventional cache, by contrast, passes data through unchanged and leaves the redundant work to the CPU downstream. 5
- Figure 2.1 Life of a query through a modern cloud-native analytical engine, drawn as a swim-lane diagram across compute, cache, and storage layers. Parsing, planning, optimization, decoding, and execution happen in the compute layer; the scan stage crosses into the cache layer, and missing data is fetched once from object storage. Filter and projection pushdown shift work between layers without changing the overall shape of the pipeline. 13
- Figure 2.2 Three caching architectures for cloud-native data systems. Private caches live inside each compute node and replicate data across the cluster. Distributed caches federate compute-local caches into a single logical store with no replication. Disaggregated caches separate cache servers from compute nodes, letting each scale independently but introducing a network hop on every cache access. 17
- Figure 2.3 Latency for common (interconnect $\times$ medium) pairings on a logarithmic axis spanning nine decades. Italic numbers under each tier translate the latency into human-scale time using the rule “if 1 ns equalled 1 second”. The underlying components and their separate contributions appear in Table 2.1; this figure shows what the application experiences once the two are composed. 21
- Figure 3.1 Reference architecture of a White-Box cache. A conventional cache (left) accepts an opaque blob keyed by an opaque identifier and returns the same bytes on a hit; the workload runs entirely above it. A White-Box cache

	(right) accepts a typed columnar value together with workload context – the schema it belongs to, the predicates that will be evaluated against it, and the lineage of operations the cache is allowed to anticipate – and returns either filtered bytes (representation), a partial representation that still answers the predicate (content), or a record-granular subset of the page it logically belongs to (granularity). The three system chapters of this dissertation instantiate one of the three internal layers (representation, content, granularity) each.	28
Figure 3.2	The fidelity spectrum a White-Box cache manages. A conventional cache moves entries between only the two end-states (Full and Evicted). A White-Box cache exposes the intermediate states and chooses between them per entry, driven by the access trace and the lineage metadata of Section 3.2. Each system chapter inhabits a different subset of the spectrum.	31
Figure 4.1	Time breakdown of filter pushdown for ClickBench Q22. Data decoding and decompression dominate CPU time on cached Parquet; filter evaluation takes less than 10%. This is the bottleneck LiquidCache attacks.	35
Figure 4.2	LiquidCache architecture – LiquidCache runs on a commodity, elastically-scaled cache server, physically close to the compute servers that handle query execution.	38
Figure 4.3	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.	39
Figure 4.4	How LiquidCache encodes string arrays – The leftmost diagram shows the Arrow string representation, it will first be converted to dictionary encoding to eliminate the repeated strings, and then the keys are compressed with bit-packing based on the dictionary size, finally the dictionary is compressed with FSST encoding. Each of the intermediate representations can be used for filter evaluation. The corresponding data size and transcoding time can be found in Section 4.4.7.	43
Figure 4.5	LiquidCache’s selective decoding compared to Parquet’s decoding – Parquet (left) has to decode an entire page even if just one element is needed. LiquidCache (right) allows each element to be decoded independently.	46

- Figure 4.6 LiquidCache’s late filter materialization compared to Parquet’s eager materialization – The first row shows the filter evaluation process of Parquet, and the second row shows LiquidCache. With two filter columns, Parquet eagerly materialize both columns, while LiquidCache only materializes the first column and use the filter mask to selectively decode the second column. 47
- Figure 4.7 ClickBench results comparing LiquidCache with three baselines (lower is better). Parquet (file server) is limited by network traffic, Parquet (pushdown) by CPU usage, and Arrow (pushdown) by memory footprint; LiquidCache avoids all three bottlenecks. 54
- Figure 4.8 TPC-H SF100 (100GB data) with 8GB cache. Data size indicates query scan size, excluding runtime structures. Arrow is best when data fits in cache and worst otherwise. Parquet outperforms ORC due to better query engine integration. Liquid consistently delivers the best performance when the input does not fit in the cache. 57
- Figure 4.9 Decomposed CPU time spent on cache server for ClickBench Q22. LiquidCache spends similar total time as Arrow (pushdown), but primarily on decoding compressed data, while Arrow (pushdown) suffers from memory stalls due to scanning 4× more uncompressed data. 58
- Figure 4.10 Memory usage and transcoding cost of “Title” column of the ClickBench – the largest column in the dataset. The bar height indicates the encoded data size, arrows denote transcoding time. Parquet only allows page-level access, while the rest of the encodings permit random element access. 59
- Figure 4.11 LiquidCache’s latency and memory usage on Q21 – LiquidCache gradually transcodes Arrow into Liquid in background, avoiding latency spikes on the critical path of query processing. 60
- Figure 4.12 Impact of transcoding overhead. Baseline Parquet reads directly from files. LiquidCache(blocking) converts one batch at a time, whereas LiquidCache overlaps transcoding with I/O. Storage devices: S3-far (cross-continent), S3/S3-express (nearest region), MinIO (same cluster), SSD (local), Memory (page cache). 62
- Figure 4.13 Ablation study of LiquidCache’s decoding optimizations vs Parquet (pushdown) as the baseline. 62

Figure 5.1	SqueezeCache caches data by how it is used, not just how frequently it is accessed, closing the gap between cached data size and utilized data size. Numbers shown for ClickBench Q25.	68
Figure 5.2	Overview of SqueezeCache. It has three core components: squeezable data layouts, lineage pushdown, and squeeze policy.	73
Figure 5.3	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.	74
Figure 5.4	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.	76
Figure 5.5	Datetime squeezing. Although physically stored as a single integer, datetime can be squeezed with different semantic components.	78
Figure 5.6	Squeeze a variant column into a set of shredded columns. 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.	79
Figure 5.7	Column lineage example. SqueezeCache analyzes the query to extract the lineage of each column. For <code>CreationDate</code> , the lineage reveals that only the day-of-week component is needed. For <code>Title</code> , 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.	81
Figure 5.8	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.	82
Figure 5.9	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 squeezed, and finally, squeezed data is evicted from the cache.	84
Figure 5.10	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.	90
Figure 5.11	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.	92
Figure 5.12	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. .	93
Figure 5.13	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.	94
Figure 5.14	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.	96
Figure 5.15	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.	97
Figure 5.16	Fingerprint bucket size vs filter effectiveness. The bucket size is the number of buckets used to store the fingerprint.	99
Figure 6.1	Throughput (log-scale) along three dimensions – point lookup, scan, and update. Each triangle is a different system; larger is better. Bf-Tree dominates both B-Tree and LSM-Tree along all three dimensions; Section 6.5.3 discusses this experiment in detail.	106
Figure 6.2	High level architecture of Bf-Tree. Like conventional B-Tree, but pages in the buffer pool are variable lengths.	112
Figure 6.3	Mini-page (var len) / leaf node (4096 bytes) layout	114
Figure 6.4	Circular buffer for variable length mini-pages.	118
Figure 6.5	Bf-Tree's Get and Insert operations.	120

Figure 6.6	Bf-Tree vs. baselines on various workloads – Bf-Tree is the only system performing well on all three representative workloads.	128
Figure 6.7	The scalability of Bf-Tree vs baselines. – Bf-Tree scales well before saturating the SSD bandwidth. RocksDB does not saturate the bandwidth at thread 31.	130
Figure 6.8	In-memory performance compared with Leanstore. – When data fits in memory, Bf-Tree, B-Tree, B δ -Tree fold in a similar design. This is a sanity check that Bf-Tree’s in-memory performance is as good as the state-of-the-art Leanstore.	130
Figure 6.9	Latency distribution of Bf-Tree vs baselines. – Bf-Tree has the best 50th and 99th percentile latency due to its efficient caching and IO handling.	131
Figure 6.10	Impact of workload skewness. –Bf-Tree performs consistently the best across all skew levels.	132
Figure 6.11	Impact of read and write workloads.	133
Figure 6.12	Impact of cache size.	134
Figure 6.13	Time spent on different components of the system – Recreated from the flamegraph [66] for better clarity.	135
Figure 6.14	Impact of copy-on-access region size (% of total circular buffer size).	136
Figure 6.15	Impact of promotion rate: the probability of promoting a record from a disk page to its mini page.	136

Abstract

Modern data systems disaggregate compute, storage, and memory across independent tiers, making data movement more expensive than computation. Yet caches still treat contents as opaque bytes: they accept blobs, index them by key, and evict each blob as a unit, without knowing its structure or which fields a query will access. This dissertation argues that this opacity has become a bottleneck and that caches should instead inspect their data.

We call this approach *White-Box caching* and develop it along three axes – *representation*, *content*, and *granularity* – each recovering information a conventional cache discards. Three systems instantiate one axis each.

LiquidCache attacks representation. It transparently transcodes Parquet into independently decodable dictionary, FastLanes, FSST, ALP, and frame-of-reference encodings, allowing predicates to use the most efficient layer. On ClickBench and TPC-H, it reduces cache CPU by up to 10× over Parquet pushdown and network traffic by two orders of magnitude over file-serving caches, while using memory comparable to compressed Parquet.

SqueezeCache attacks content. Profiling shows that up to 98% of cached bytes – even under Belady-optimal replacement – are never read by the queries that warmed them. Guided by lineage analysis, SqueezeCache replaces each entry with a compact, lossy subset selected per column across a four-state fidelity spectrum. On ClickBench and JSONBench, it improves hit ratios by up to 4×, lowers latency by up to 22×, and cuts I/O by up to 12× over an uncompressed cache; corresponding gains over a compression-only baseline are 2×, 2×, and 8×.

Bf-Tree attacks granularity for transactional workloads. Variable-length mini-pages decouple cache pages from disk pages, yielding one index that is 2× faster than RocksDB on scans, 6× faster than a B-Tree on writes, and 2× faster than both on point lookups.

LiquidCache and SqueezeCache form a 50,000-line open-source Rust artifact, while Bf-Tree is implemented separately. The work also contributes substantially to Apache DataFusion, Arrow, and Parquet.

Chapter 1

Introduction

The way computer systems store and process data has changed more in the last fifteen years than in the preceding fifty. In the 2000s, the dominant model was the integrated database: a single machine bundled compute, memory, and storage, optimized them together, and answered queries from data that lived a few millimeters from the CPU that read it. That assumption – that the unit of provisioning is also the unit of execution – quietly shaped every layer of the stack, from disk formats to buffer pools to query planners.

Cloud computing dismantled it. Storage became a service [5, 13], addressable from any compute instance, priced per request [54], and replicated across data centers. Compute became elastic, ephemeral, and increasingly serverless: a query may run in a function that did not exist a second before it was issued and will not exist a second after it completes [24, 164]. Memory, for a decade considered the inelastic resource bolted to a CPU, is itself being disaggregated; RDMA, CXL, and far-memory tiers are turning DRAM into something more like a network service [49, 73, 104, 120, 140, 147]. The result is an architecture in which every layer scales on its own clock – storage by capacity, compute by demand, memory by working set.

This disaggregation has been an enormous win for cost, elasticity, and operational simplicity, and it underlies every major cloud-native data platform today: Snowflake [170], BigQuery [123], Redshift [5], Databricks [12], and the open lakehouse stack built on Parquet [169], Iceberg, and engines like Apache DataFusion [93, 94]. But it has also stretched a fundamental tension. The further compute moves from storage, the more every byte costs to fetch – in latency, in bandwidth, and in literal dollars per request. The economics of a modern analytical query are increasingly dominated not by the work of computing the answer but by the work of bringing the data within reach.

1.1 Caching: The Hidden Foundation

Whenever the gap between where data lives and where it is used has grown, the answer has been the same: caching. CPU caches absorbed the widening DRAM-to-core latency gap. Operating-system page caches mediated between volatile memory and rotational disks [14]. Database buffer pools turned random-access disk pages into byte-addressable working sets. Content delivery networks pushed static assets to the network edge so that a click in Sydney did not have to wait for Virginia. At every layer of the stack, and in every era of computing, caching has been the unsung discipline that made the next leap in scale economically viable.

The literature reflects this centrality. Decades of work have refined the question of *what* to keep: LRU-K [131], ARC [122], 2Q [82], S3-FIFO [177], SIEVE [189], and a growing body of machine-learning-based admission and eviction policies [18, 62, 174, 176]. A parallel line of work has refined the question of *where* to keep it: tiered memory [73, 120], disaggregated buffer pools, distributed caches like Alluxio [4], elastic caches built on ephemeral serverless functions [171], and production-grade engines like CacheLib [23]. Cloud-native data warehouses have made caching a first-class system component: Snowflake reports it [170], Databricks markets it [41], and lakehouse architectures effectively assume one between object storage and any engine that hopes to be interactive [13, 52].

What unites this body of work is a remarkably stable abstraction. A cache, almost without exception, is an opaque key-value store. Data goes in as a blob, comes out as a blob, and the cache’s job is to decide which blobs to keep. The policy may be sophisticated, the implementation may be distributed and fault-tolerant, but the cache itself does not know what is inside the blob, how it is encoded, which fields a query needs, or which byte ranges of which records are actually hot. The cache treats data as undifferentiated bytes. For decades this opacity was a feature, not a bug: it made caches simple, general, and composable. This dissertation argues that, for modern data systems, that opacity has become the bottleneck.

1.2 The Challenges of Caching in Modern Data Systems

Three forces are exposing the limits of opaque caching, each corresponding to a way that modern data has outgrown the cache’s view of it.

The first is *format*. Data on storage is encoded *for* storage: Parquet [169], ORC [10], and emerging columnar formats [91, 124, 156] pack values with cascading dictionary, run-length, and bit-packing schemes for compactness, not for arithmetic. When a cache stores these bytes, every query that touches them must pay the decoding cost – and our measurements show that decoding and decompression dominate CPU time in filter pushdown on Parquet, leaving the filter evaluation they were meant to accelerate under 10% [117, 180]. An opaque cache cannot do anything about this, because it does not know it is holding Parquet, let alone how to rearrange it.

The second is *usage*. A cached blob carries every byte of every field, but queries rarely use every byte of every field. A 64-bit datetime is held in full when a query needs three bits to extract day-of-week. A 200-byte URL is cached in its entirety when a LIKE predicate needs only a substring fingerprint. A semi-structured JSON value [53] is materialized whole when a query references one of its hundred fields. Our profiling shows that under Belady-optimal [21] replacement, up to 98% of cached bytes are never read by the queries that warmed them. The cache holds the data the storage layer happened to produce, not the data the workload actually consumes.

The third is *granularity*. A cache unit is typically a disk page (4 KB to 64 KB), but the records inside a page are hot at different rates, written at different rates, and worth different amounts to keep. Page-level caching wastes memory on cold records that share a page with hot ones, and pays a full page write for every byte updated. Modern NVMe SSDs, whose bandwidth now approaches memory bandwidth [68], have removed the original justification for tying cache granularity to disk granularity – yet the abstraction has not caught up.

These three challenges – format opacity, usage opacity, and granularity opacity – are not isolated curiosities. They appear, in different guises, in every modern data system that takes caching seriously. Together they motivate a different way of thinking about what a cache is for.

1.3 White-Box Caching

The conventional cache is a black box: it accepts opaque blobs, indexes them by key, scores them by recency or frequency, and evicts them as a unit. The policies vary, the implementations vary, but the contract is the same: the cache must not need to understand its contents. This dissertation proposes an alternative.

White-Box caching is the design philosophy that a cache, in a modern data system, should be allowed to see inside the data it holds, understand the operations being performed against it, and choose its representation, content, and unit accordingly. A White-Box cache is no longer a passive store; it is an active participant in query execution. Its freedoms run along three axes (Figure 1.1), each corresponding directly to one of the opacities identified above:

- A White-Box cache may *adapt its representation* to the workload – choosing an encoding driven by the operators that will run against it, rather than by the storage layer that produced it.
- A White-Box cache may *omit content* within a cached unit, keeping only the fragments that the workload demonstrably uses and demoting or evicting the rest.
- A White-Box cache may *vary the granularity* of cached units, holding individual records, partial pages, or whole pages adaptively, depending on access patterns.

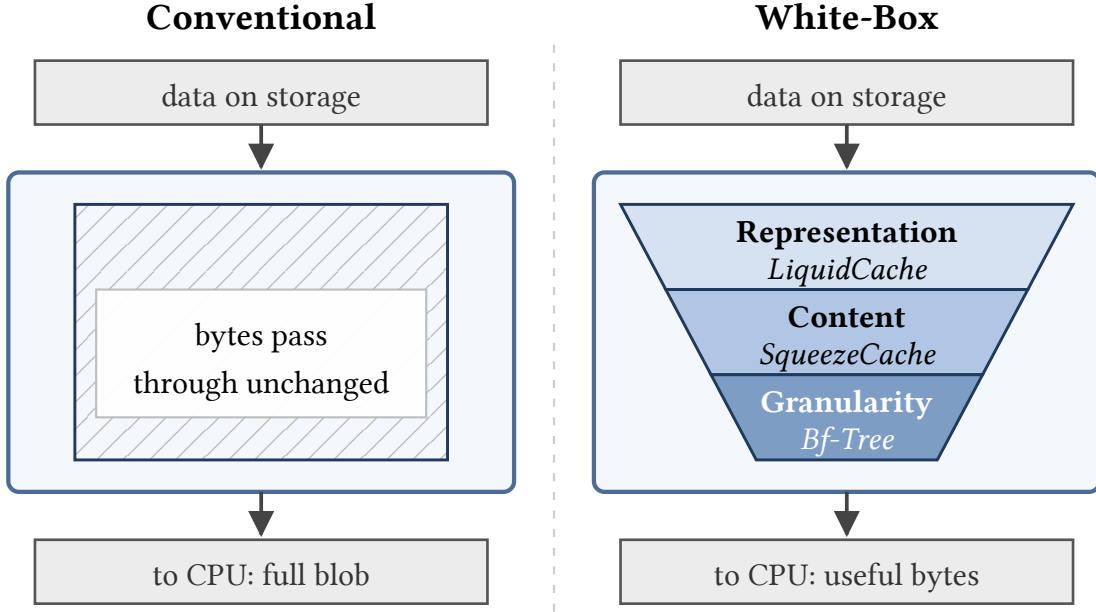


Figure 1.1: Three axes of opacity that conventional caches accept and that the White-Box caches in this thesis narrow. Each contribution attacks one axis: *LiquidCache* makes the in-cache encoding an adaptive, workload-driven choice; *SqueezeCache* decides what content within each cached unit to keep; and *Bf-Tree* decouples cache granularity from disk granularity. A conventional cache, by contrast, passes data through unchanged and leaves the redundant work to the CPU downstream.

This stance has deep roots in storage systems. Two decades ago, *semantically-smart disk systems* [154] showed that a block device – ostensibly the most opaque layer in the stack – could reconstruct the file-system structures above it from the raw block stream and put that knowledge to work: for availability, as in D-GRAID [153]; for liveness-aware reliability [151]; and for storage tuned to the database running atop it [152], with later work formalizing the limits of what such inference can soundly recover [150]. White-Box caching makes the same wager one layer up, but with the labor reversed: instead of reverse-engineering semantics the upper layer never disclosed, a White-Box cache is handed the schema, the query plan, and the access trace outright, and is free to act on them.

Each axis trades simplicity for leverage. The price is that the cache must now consume more than just keys: it must understand schemas, encodings, query plans, and access traces. The reward is that the cache can do work that would otherwise be repeated, redundantly,

by every consumer of its data – work that, in aggregate, dominates the cost of modern analytical workloads.

White-Box caching is not a single technique but a stance, and adopting it forces a series of concrete questions. What should a cache understand about the format of the data it stores, and how does that understanding integrate with engines that were not written with the cache in mind? How should it represent that data internally – losslessly, lossily, or somewhere in between – and how does it decide when to move between representations? How does it learn which fields and operations the workload actually exercises, without coupling itself so tightly to a single query engine that it cannot be reused? How does it balance memory pressure, CPU transcoding cost, and the risk of needing to re-fetch from storage when a lossy approximation turns out to be insufficient? How does it talk to the storage layer beneath it now that NVMe bandwidth, `io_uring` [17], and direct IO have changed the cost of every disk access by orders of magnitude?

These questions span the stack – from data formats to query optimizers to operating-system interfaces – and no single technique answers all of them. The three principal contributions of this dissertation are three answers, each attacking a different axis of opacity, that together demonstrate that White-Box caching is not just an idea but a practical, deployable foundation for modern data systems.

The first contribution opens up the *representation* axis. It asks what a cache could do if it picked its own encoding adaptively, in response to the workload running against it – rather than passively storing the encoding the storage layer produced – and shows that the answer is dramatic: an order of magnitude less CPU, two orders of magnitude less network traffic, and a memory footprint comparable to compressed storage.

The second contribution opens up the *content* axis. It asks how much of a cached unit a workload actually needs, and shows that the honest answer – usually a small fraction – is something a cache can detect and exploit, given enough information about how queries use their inputs.

The third contribution opens up the *granularity* axis. It asks why cache pages have to look like disk pages at all, and shows that decoupling the two yields a storage engine that is simultaneously the best at point lookups, range scans, and writes – a combination that has long been considered a fundamental trilemma in the larger-than-memory literature.

Together, these three systems form a coordinated argument: that the next foundation for caching in modern data systems is not a new policy, a new replacement algorithm, or a

new hardware tier, but a new abstraction – one in which the cache is given permission to look inside the data it holds.

1.4 Contributions and Outline of Chapters

The remainder of this dissertation develops the White-Box caching philosophy through a sequence of substantive chapters. The next chapter (Chapter 2) lays out the technical landscape – query lifecycle, caching architectures, and the cost gradients that make caching worth doing – against which the three contributions are positioned. A short chapter that follows (Chapter 3) develops the shared design vocabulary – the reference architecture, the information-flow channels, and the fidelity spectrum – on which the three system chapters then instantiate one axis each. Each system chapter presents its work essentially as it was published, with a unifying preface that places it within the framing established above and a closing transition that motivates the next. The chapters around them – background, the White-Box reference design, engineering, related work, and conclusion – consolidate the technical foundations, the cross-cutting evidence, the implementation lessons, and the broader implications.

The contributions of this dissertation are as follows:

- A new framing for cache design – *White-Box caching* – articulating three axes (representation, content, granularity) along which conventional caches discard information that modern data systems can no longer afford to lose.
- *LiquidCache*, the first disaggregated pushdown cache for cloud-native analytics, which co-designs an in-cache data format with filter evaluation and demonstrates up to 10× lower CPU usage than Parquet without increasing memory.
- *SqueezeCache*, a memory-efficient caching mechanism that generalizes eviction into a multi-state squeeze spectrum, lineage-aware and workload-adaptive, achieving up to 4× higher cache hit ratios and 22× lower query latency than an uncompressed cache (2× on both over the compression-only state of the art).
- *Bf-Tree*, a read-write-optimized larger-than-memory range index whose mini-page design and variable-length buffer pool decouple cache granularity from disk granularity, outperforming both B-Trees and LSM-Trees on point lookup, scan, and write workloads.
- A production-grade open-source implementation of *LiquidCache* and *SqueezeCache* in over fifty thousand lines of Rust (*Bf-Tree* is a separate implementation), carrying sub-

stantial upstream contributions to Apache DataFusion, Arrow, and Parquet, and adoption by multiple analytical systems.

The remainder of this chapter previews each substantive chapter in turn, before the next chapter establishes the technical background that the three works share.

1.4.1 LiquidCache: Pushdown Caching for Cloud-Native Analytics

Cloud-native analytical systems separate compute from storage, with data living in object stores in Parquet format. To mask the latency and cost of remote reads, the industry has converged on disaggregated caching layers between compute and storage. But existing caches either move whole files over the network, saturating bandwidth, or attempt filter pushdown directly on Parquet and pay a CPU bill that exceeds the savings – because most of the pushdown time is spent decoding Parquet, not evaluating filters. Many production systems have responded by disabling pushdown altogether.

LiquidCache attacks the representation axis by treating the in-cache encoding as an adaptive, workload-driven decision rather than something the storage layer dictates. It transparently transcodes Parquet into “Liquid,” an in-cache format that combines dictionary encoding, FastLanes bit-packing [1], FSST string compression [25], ALP floating-point compression [2], and frame-of-reference integer encoding into a cascading layout where every layer permits independent element access. Because the cache controls the encoding, it can match each predicate to the most compact representation that suffices – encoded equality, dictionary-only substring search, selective decoding, and late filter materialization – so most data is never decoded at all. Liquid lives only inside the cache, so the rest of the ecosystem continues to see Parquet, and the cache is free to evolve its encodings without coordinating with anyone.

We integrate LiquidCache with Apache DataFusion and evaluate it on ClickBench [35] and TPC-H. LiquidCache reduces cache CPU time by up to 10× over Parquet pushdown, reduces network traffic by two orders of magnitude over file-serving caches, and matches Arrow-in-memory caches at a quarter of the memory cost. Many of the underlying techniques, including the StringView representation and the rebuilt Parquet decoding pipeline, have been upstreamed and are now defaults in DataFusion, Arrow, and Parquet.

1.4.2 SqueezeCache: Curating Cache Content Through Lineage

LiquidCache solves the format problem, but our profiling exposes a second one. Across realistic workloads, up to 98% of cached bytes – even under Belady-optimal replacement [21] – are never read by the queries that brought them into memory. The cache holds whole columns when queries use sub-fields, whole strings when predicates check prefixes, whole datetimes when projections extract days of the week. This is the *content* axis, and it is invisible to any cache that does not understand query intent.

The data type most responsible for this waste is also the most pervasive: strings. Real-world analytical data is overwhelmingly stringy – URLs, search phrases, titles, JSON payloads, identifiers – and strings exhibit properties that columnar caching machinery handles awkwardly: variable-length and often long, mostly compared rather than read in full, and dominated by a handful of predicate patterns (equality, prefix, length, substring). The SqueezeCache chapter opens its background with a focused look at these byte-level properties, drawn from ClickBench and JSONBench [37], because they explain why even a representation-aware cache like LiquidCache still over-stores strings – the cache has so far decided what to *encode* but not what to *keep* – and they motivate the squeeze targets of the rest of the chapter.

SqueezeCache then breaks the binary keep-or-evict decision into a spectrum of states – Uncompressed, Compressed, Squeezed, Evicted – and dynamically transitions cached batches through them under memory pressure. Squeezing is lossy by design: a string column can be reduced to a prefix plus a membership fingerprint [157], an integer column to a quantized or clamped approximation, a datetime to a single extracted component, a Variant column to its referenced sub-fields. To decide what to keep, SqueezeCache introduces *lineage pushdown*: an optimizer rule that traces how each column is used in the query plan and communicates the result to the cache through schema metadata, without coupling the two systems together. When a squeezed representation is insufficient for a future query, SqueezeCache reads the missing bytes from disk and re-hydrates.

The result is a cache that holds far more of the working set per byte of memory. On ClickBench and JSONBench, SqueezeCache improves cache hit ratios by up to 4×, reduces query latency by up to 22×, and cuts I/O by up to 12× compared to an uncompressed cache (2×, 2×, and 8× respectively over a compression-only baseline). SqueezeCache is now the default eviction mechanism in the LiquidCache project.

1.4.3 Efficient I/O and Bf-Tree

White-Box caching’s third axis lives at the boundary between memory and disk. Modern NVMe SSDs deliver tens of gigabytes per second of bandwidth – enough that classical assumptions about disk slowness no longer hold, and the abstractions built on those assumptions (kernel page cache, page-sized buffer pools, page-coupled write paths) have become bottlenecks rather than helpers [68].

The shared I/O substrate that underpins this work – an `io_uring`-based, runtime-agnostic I/O subsystem with custom buffer management, direct I/O, and a small-file metadata optimization that consolidates millions of cached batches into a single self-managed file with in-memory indexing, append-only logging, and online garbage collection – is presented alongside the rest of the engineering in Chapter 7. This subsystem is used by both LiquidCache’s disk tier and Bf-Tree.

The chapter’s main contribution is *Bf-Tree*, a read-write-optimized B-Tree for larger-than-memory workloads. Bf-Tree’s central idea is to decouple cache pages from disk pages: in-memory *mini-pages* are variable-length and need not mirror the 4 KB disk pages they correspond to. A mini-page can hold a few hot records, a write buffer of recent updates, or – when needed – the full leaf page for range scans. A novel variable-length buffer pool, organized as a circular log with a copy-on-access second-chance region, manages these mini-pages concurrently and at scale. Bf-Tree is 2× faster than RocksDB [48] on scans, 6× faster than a B-Tree on writes, and 2× faster than both on point lookups, while keeping in-memory performance on par with Leanstore [98]. It demonstrates that the White-Box philosophy is not specific to analytics: granularity opacity is a problem for transactional KV stores too, and the same move – looking inside the page – pays off there as well.

1.4.4 Engineering a White-Box Cache

Ideas of this kind succeed or fail at the level of implementation. A cache that understands data formats, query lineage, and access granularity has many more moving parts than a cache that does not, and is correspondingly easier to get wrong. This chapter draws out the engineering lessons of building and shipping these systems, organized around a single observation: *opacity is a form of delegation*, and every decision a White-Box cache reclaims for performance comes bundled with a burden the opaque cache had delegated away.

It traces those burdens one at a time – a modular codebase that keeps an invasive technique adoptable, a runtime-agnostic `io_uring` substrate that makes the cache’s byte-exact memory accounting credible, formal verification with Verus for the crash-consistency obligation inherited from the filesystem, and differential fuzzing that holds the squeezed string layout indistinguishable from the format it replaced – before distilling them into a handful of lessons for building White-Box caches. Several of these techniques, including the `StringView` default and a rewritten Parquet decoder, have been upstreamed to Apache DataFusion, Arrow, and Parquet.

1.4.5 Related Work and Conclusion

The final two chapters consolidate the related work and the conclusions. The related-work chapter places the three systems within a unified taxonomy of cache designs – opaque-block caches, semi-transparent caches such as result and predicate caching, and the White-Box category this dissertation defines – and identifies the boundary of each. The concluding chapter returns to the three-axis framing introduced here, summarizes the empirical case for it, and outlines the open questions: White-Box caching across consistency boundaries, multi-tenant scheduling of squeeze and transcoding work, and what a fourth axis might look like as data systems continue to evolve.

The dissertation’s central claim is straightforward. Caching has been an opaque, byte-oriented abstraction for too long, and modern data systems can no longer afford the redundancy that opacity costs. By giving the cache permission to see – into formats, into content, into granularity – we recover work that the rest of the stack would otherwise repeat, query after query. White-Box caching is a new foundation for that recovery, and the chapters that follow show how three concrete systems put it into practice.

Chapter 2

General Background

This chapter introduces the technical foundations that the three contributions of this dissertation share. We organize it around three concerns that any modern analytical data system must address: how a query flows through the layers of compute, cache, and storage that constitute such a system; how caches are architected to bridge the widening gap between the two endpoints of that flow; and what the actual costs of moving data between those layers look like today. We summarize the relevant prior work for each so that subsequent chapters can focus on what is new. The cost-model section is a reference for the rest of the dissertation rather than a survey. The chapter closes by restating the three axes of opacity introduced earlier, now equipped with the vocabulary required to address each.

2.1 Life of a query in cloud-native analytics

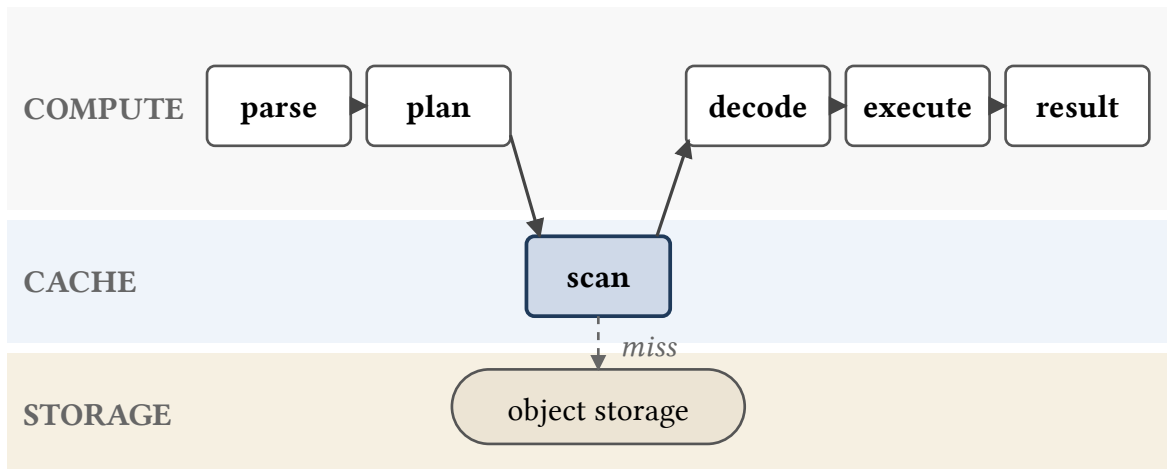


Figure 2.1: Life of a query through a modern cloud-native analytical engine, drawn as a swim-lane diagram across compute, cache, and storage layers. Parsing, planning, optimization, decoding, and execution happen in the compute layer; the scan stage crosses into the cache layer, and missing data is fetched once from object storage. Filter and projection pushdown shift work between layers without changing the overall shape of the pipeline.

Modern analytical query execution has converged on a layered pipeline that all major engines – Spark [183], Photon [20], Velox [133], DuckDB [136], ClickHouse [143], Presto [161], Dremel [123], and Apache DataFusion [93, 94] – recognizably share. This section traces a single query through that pipeline (Figure 2.1) so that subsequent chapters can refer to specific stages without redescribing the whole. Before stepping through the stages, we need a short detour into the data representations that the pipeline operates on, because what each stage costs depends critically on how the bytes are laid out.

2.1.1 Data representation on disk and in memory

Cloud-native analytical systems no longer privately own the file format they read. Where classical OLAP engines paired a proprietary on-disk layout with the engine that read it, contemporary *lakehouse* architectures [12, 13, 128] store data in open direct-access formats that many engines must interpret in compatible ways. Apache Parquet [169], Apache ORC [10], and Apache Arrow [93] have become the de facto representations for analytical

data, deployed at petabyte scale across Snowflake [162, 170], Databricks [12, 41], Redshift [5, 137], BigQuery [123], ClickHouse [143], and the open lakehouse stack built around DataFusion [93, 94], DuckDB [136], Photon [20], and Velox [133].

Parquet [89, 169] is a self-describing columnar format optimized for storage. A Parquet file is organized hierarchically into row groups, column chunks, and pages; each page is independently compressible and decodable. Within a page, values are encoded using a cascade of techniques – dictionary encoding, run-length encoding, delta encoding, and plain encoding – chosen at write time to maximize compression. Pages are further compressed with general-purpose codecs (Snappy, Zstd, LZ4). Parquet’s page index [9] enables data skipping by storing per-page minimum, maximum, and null statistics; more recent extensions add Bloom filters and finer-grained zone maps. The strengths of Parquet – compactness, ecosystem support, and stable governance – are bound to its weaknesses for query processing: its cascading encoding produces excellent compression ratios [89, 184] but couples values within a page so tightly that selective access requires decoding the entire page.

Apache Arrow [93] addresses this by serving as the in-memory companion: a columnar layout optimized not for storage but for vectorized execution. Arrow’s design enables zero-copy sharing across processes and engines and has become a near-universal currency for in-memory analytical data [11, 67, 80, 92, 95, 165]. Together, Parquet and Arrow define a pipeline: data lives in Parquet on storage, becomes Arrow in memory, and the conversion – *decoding* – consumes a disproportionate share of every query that touches Parquet [117, 180, 184].

A line of recent work tries to narrow this decoding cost without sacrificing compactness. FSST [25] compresses string columns by extracting frequent substrings into a small symbol table; FastLanes [1] implements vectorized bit-packing layouts that decode directly into SIMD registers; ALP [2] compresses floating-point columns by recovering integer representations when possible. Earlier schemes – Chimp [109], Gorilla [134], ELF [106] – targeted streaming time-series data with related techniques. Frame-of-reference (FoR) encoding subtracts a column-local minimum to shrink dynamic range; cascading dictionary, FoR, and bit-packing form the basis of BtrBlocks [91] and FastLanes [1] themselves. Beyond compression, lightweight indexes such as column imprints [155] and Sieve [166] summarize per-block contents to skip irrelevant data outright, and instance-optimized layouts [46] adapt storage organization to observed workload patterns. Despite these gains, the schemes

have been slow to enter open file formats [89], and the ecosystem has effectively locked in to the encodings available in 2013 [61, 185]. A second wave of file formats – Nimble [124], Vortex [156], LanceDB [95], and BtrBlocks itself [91] – proposes outright replacements, but compatibility with the existing Parquet ecosystem is a persistent obstacle. A complementary stance, due to MorphStore [40], is that the right place to evolve encoding is not the file format at all but the engine, by making operators compression-aware so that data never needs to be fully decoded. This dissertation takes a related position: the cache is the layer where representation can adapt without breaking the ecosystem.

2.1.2 Parsing and logical planning

A query arrives as SQL or as a declarative API call. The engine parses it into an abstract syntax tree and binds it against a catalog, producing a *logical plan*: a tree of relational operators (scan, filter, project, join, aggregate) whose semantics are well defined but whose execution strategy is not. Catalog metadata is itself usually cached at this point [13]; the cost of resolving a table name should be invisible to the query.

2.1.3 Logical optimization and pushdown

The logical plan undergoes a sequence of rewrites that preserve semantics but reorder operators. Predicate pushdown moves filter operators closer to the scan; projection pushdown drops columns the query never reads; constant folding evaluates compile-time expressions; subquery unnesting flattens correlated structures; common subexpression elimination shares work across the plan. The output is an optimized logical plan whose operators sit closer to the data and farther from the result.

Pushdown is the optimization most relevant to this dissertation. The concept dates to early relational systems [158], but its modern resurgence is driven by the cost of moving data over the network in cloud-native architectures. Filters can now be pushed not just to the storage operator inside the engine, but all the way to the cache server (Chapter 4), to the storage device (Amazon S3 Select [6], Azure Data Lake [125]), to programmable hardware [47, 78, 88, 149], or even cooperatively to several of these in concert, as in FlexPushdownDB [179, 180] and Fusion [117]. Each layer that evaluates a filter shrinks the data that subsequent layers must handle, but each must also pay the work of evaluation – a trade-off that complicates pushdown when the lower layer is CPU-poor [117, 180]. Recent engines also

generate *dynamic filters* at runtime, propagating partial results back into earlier scans to shrink them further [19]; these too can be pushed into the cache.

2.1.4 Physical planning

Logical operators are translated into physical operators that specify *how* each step executes: hash join versus sort-merge join, hash aggregation versus streaming aggregation, parallel versus single-threaded scan, batch size, projection pushdown into the scan operator. Cost-based optimizers [13, 20, 133, 136] choose among physical alternatives by estimating their resource costs from table statistics. The output is a physical plan: a dataflow graph that the runtime will execute.

2.1.5 Scan and decode

Execution begins at the leaves of the physical plan, where scan operators read columnar files from storage. In a cloud-native system the scan operator does not access local files; it reads from a cache server, a private SSD cache, or directly from object storage, depending on which architecture in the next section the deployment uses. Range requests are batched to amortize object-store fees [54], metadata is parsed once and reused, and projection pushdown ensures that only the requested columns are read at all.

The scanned bytes are still in their on-disk encoding. The next step is decoding – transforming compressed Parquet pages into Arrow batches that downstream operators can consume. Studies [117, 180, 184] consistently report that decoding consumes a dominant share of scan-stage CPU on Parquet workloads, exceeding 90% in some configurations. This observation is the foundation of LiquidCache in Chapter 4.

2.1.6 Vectorized execution

Above the scan, the engine processes data in batches: arrays of values are passed between operators rather than tuples one at a time. This *vectorized* execution model, pioneered by MonetDB/X100 and refined by Photon [20], Velox [133], DuckDB [136], and DataFusion [93], amortizes operator-call overhead, enables SIMD, and preserves CPU-cache locality. Within a batch, *late materialization* [108, 158] avoids decoding values that subsequent predicates will discard, and selection vectors [158] track which rows have survived earlier filters without rewriting the underlying columns. Compression-aware execution [40] goes

further by evaluating predicates on encoded data, deferring decoding until it is unavoidable – a principle that Chapter 4 develops in the context of caching.

2.1.7 Returning results

The final operator produces a stream of result batches that are encoded for transmission – in Arrow Flight [11, 80], in JSON, or in some engine-specific protocol – and shipped back to the client. For interactive workloads, the time to the first batch matters as much as the time to the last [113, 123, 161]: a dashboard can render incrementally if results stream, but a half-second hiccup is enough to disrupt interactive exploration [113].

2.2 Caching architectures

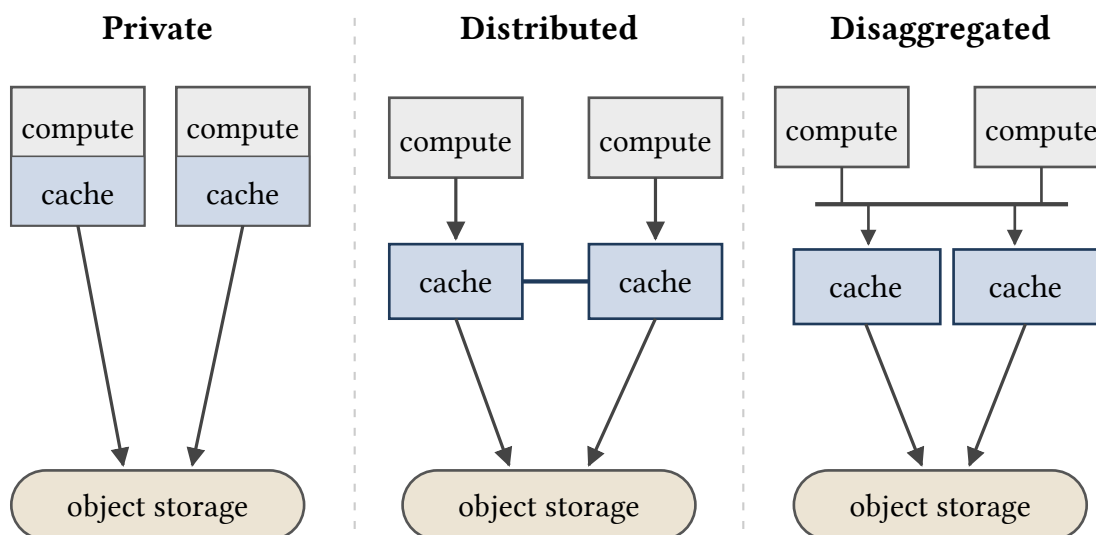


Figure 2.2: Three caching architectures for cloud-native data systems. Private caches live inside each compute node and replicate data across the cluster. Distributed caches federate compute-local caches into a single logical store with no replication. Disaggregated caches separate cache servers from compute nodes, letting each scale independently but introducing a network hop on every cache access.

A cache mediates between two layers that differ in cost or latency. In cloud-native data systems, the relevant gap is between object storage and compute: object stores cost cents per gigabyte for capacity but charge per request and impose hundred-millisecond first-byte latencies [54], while compute is elastic, ephemeral, and expects byte-addressable access at

memory speed. Every major cloud-native engine therefore deploys some form of caching layer between them. Figure 2.2 summarizes the three architectural families that have emerged; this section describes each, together with two orthogonal axes – what a cache caches, and how it evicts.

2.2.1 Private cache

The simplest architecture places the cache inside the compute node itself, drawing on spare memory or attached SSD. Each node maintains its own private cache and never shares state with peers. This is the model used by Crystal [52], by Amazon Redshift’s compute-local SSD cache [5, 137], by Databricks’s Disk Cache [41], and by Bauplan [163, 164]. Private caches are easy to deploy and easy to reason about: no consensus protocol, no consistency to maintain, no shared state between nodes. They scale linearly with the compute fleet because every compute node carries its own cache.

The drawback is equally simple. With N compute nodes accessing the same data, a private-cache fleet caches it N times, wasting both DRAM and SSD. The redundancy grows with cluster size, and ephemeral compute environments – particularly serverless functions [24, 164] – discard their caches between invocations entirely. Private caches are best suited to small clusters with persistent compute, and they are increasingly out of step with the elastic, serverless deployments that dominate modern analytical platforms.

2.2.2 Distributed cache

To eliminate redundancy, distributed caches coordinate state across compute nodes so that each piece of data is cached at most once across the cluster. Snowflake’s local SSD cache [170] uses consistent hashing to partition the cache across compute nodes, and Alluxio [4] similarly federates participating nodes into a single logical cache spanning the cluster. A query running on one node may transparently fetch cached data from another node’s local store; the cache layer makes the placement invisible to the engine above.

Distributed caches achieve higher hit ratios at fixed total memory than private caches do, but the coordination cost is non-trivial. Adding or removing a compute node requires rebalancing cached data across the survivors – typically through consistent hashing or a more elaborate placement scheme – and load balancing during query execution requires the planner to know where each piece of data lives. The cache is no longer invisible to

the engine. Distributed caches are also tightly coupled to compute: a cache outage and a compute outage become the same incident, and elastic resizing of one forces resizing of the other.

2.2.3 Disaggregated cache

The third architecture separates the cache from compute entirely, hosting it on dedicated cache servers shared by multiple compute nodes. This is the model adopted by Google’s Napa [3], by FoundationDB’s storage layer [192], by AWS ElastiCache [7], by Krypton [34], and by managed-Redis offerings used by countless smaller systems. The cache scales independently of compute: a workload that grows in working-set size can add cache capacity without provisioning compute, and vice versa. CompuCache [188] and Ditto [148] explore elastic variants in which spot or serverless capacity is used to expand the cache layer dynamically, and InfiniCache [171] pushes the idea to the extreme by storing cache state in ephemeral cloud functions themselves.

The compute–cache communication, however, moves to the network. For analytical workloads that scan large columns, the network can easily become the new bottleneck. Two responses are visible in the literature. The first is to use the disaggregated cache only for *metadata*, keeping data in private caches or scanning it from storage directly [3, 52, 55, 192]. The second is to push computation into the cache so that less data crosses the network – the *pushdown caching* regime that LiquidCache and SqueezeCache target. A third response, orthogonal to caching, is to relocate memory itself: FaRM [49], AIFM [140], LegoOS [147], Pond [104], TPP [120], and Hao et al. [73] investigate disaggregated DRAM tiers built on RDMA or CXL, while HetCache [130, 141], HPCache [129], and DEX [116] examine heterogeneous tiered caches that span DRAM, CXL-attached memory, and NVMe.

2.2.4 Semantic and partial caches

Orthogonal to topology, caches may also vary in *what* they cache. Result caching stores the output of past queries; if the same SQL arrives again, the cached result is returned without execution at all. Materialized views generalize this by maintaining pre-computed sub-queries. Predicate caching [142] is a more recent variant that records which row ranges have satisfied past predicates, accelerating subsequent scans. STsCache [87, 105] and TSCache [112] push semantic caching further by exploiting properties specific to

time-series workloads, and ReCache [18] and ChronoCache [62] mid-tier-cache analytical query results to anticipate user behavior. Each of these is precise about *what* is cached, but typically rigid about *how much*: an entry is either present in full or not at all. SqueezeCache (Chapter 5) departs from this pattern by treating the fidelity of a cached entry as a spectrum.

2.2.5 Eviction policies

A cache must decide which entries to keep when capacity is exhausted. The literature on eviction is vast and orthogonal to the work in this dissertation; we summarize it only to establish vocabulary. The classical baseline is Belady’s optimal [21], which evicts the entry whose next access is farthest in the future. Among implementable policies, LRU [14], LRU-K [131], 2Q [82], ARC [122], and CLOCK have been the workhorses for decades. Recent work has produced simpler policies that match or exceed these: S3-FIFO [177] replaces LRU’s complexity with three FIFO queues, and SIEVE [189] uses a single FIFO with a moving cursor. A parallel line of work applies machine learning to eviction, including GL-Cache [176] and Baleen [174] for object caches. Engine-level deployments such as CacheLib [23] expose a tunable framework rather than a fixed policy, and recent work on elastic cost-aware caching [28, 90] asks how policies should account for cloud pricing and consistency. The work in this dissertation is orthogonal to all of these: a White-Box cache can run any of these policies and benefits from their improvements. What changes is the unit on which a policy operates, and the choice that the unit expresses.

2.3 Cost model every system researcher should know

Every chapter that follows appeals to the same set of cost figures: the latency, bandwidth, and dollar cost of moving a byte across some particular boundary. This section collects those figures in one place. It is intended as a working reference rather than a survey, so the numbers are deliberately rounded – precise to perhaps half an order of magnitude – and reflect the state of 2024–2025 commodity hardware and major-cloud pricing. The framing follows the tradition of Jeff Dean’s *latency numbers every programmer should know*: a small list of numerical anchors, organized on a single logarithmic spine, with enough commentary to turn the list into intuition.

Numbers in cloud-native data systems span roughly ten orders of magnitude. The CPU sees its own L1 cache in around a nanosecond; the same CPU sees a cross-continent S3

request in around two hundred milliseconds. Most design decisions in this dissertation come down to recognizing which of these tiers a particular piece of work currently sits at, and what it costs to move it to a different tier.

2.3.1 Decomposing the hierarchy: interconnect $\times$ medium

Any latency or bandwidth a query observes is determined by two physical components. The *interconnect* is the wire and protocol the request traverses: the on-chip fabric, the memory bus, the PCIe lane, the SATA cable, the NIC and its network. The *medium* is the storage technology where the data actually lives: SRAM cells in a CPU cache, DRAM in a DIMM, NAND flash in an SSD, magnetic platters in a hard drive, the layered backend of an object store. The achievable latency of any (interconnect, medium) pairing is bounded below by whichever of the two takes longer, and the achievable bandwidth is bounded above by whichever has the lower ceiling. Decomposing the hierarchy this way is more compact than enumerating every pairing, and more educational because it lets the reader reason about combinations the table does not list.

Figure 2.3 visualizes a handful of common pairings on a single logarithmic axis; Table 2.1 lists the underlying components and their cost. Vendor specifications, published benchmark studies [54, 68], and our own measurements agree to within a small factor, so readers chasing a specific deployment should treat these as ballparks rather than commitments.

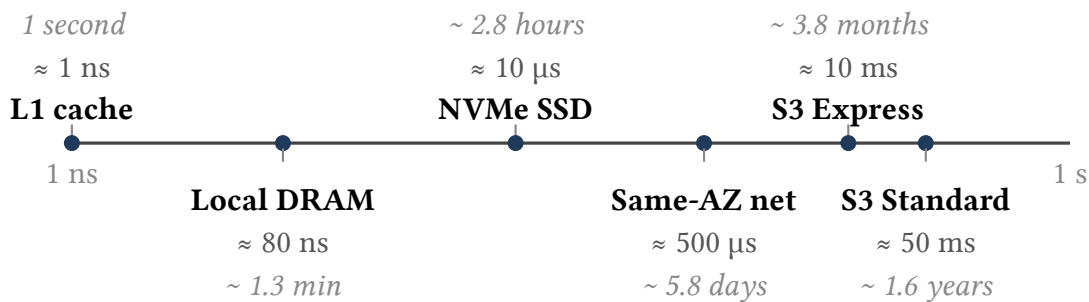


Figure 2.3: Latency for common (interconnect $\times$ medium) pairings on a logarithmic axis spanning nine decades. *Italic numbers under each tier translate the latency into human-scale time using the rule “if 1 ns equalled 1 second”. The underlying components and their separate contributions appear in Table 2.1; this figure shows what the application experiences once the two are composed.*

Component	Latency	Bandwidth	BOM	Operating cost
Interconnects				
On-chip cache fabric	~ 0.5 ns	> 200 GB/s	bundled	—
Local memory channel (DDR5)	~ 5 ns	~ 80 GB/s	bundled	—
NUMA inter-socket hop	~ 70 ns	~ 50 GB/s	bundled	—
CXL 2.0 × 16	~ 150 ns	~ 64 GB/s	bundled	—
PCIe 5.0 × 4	~ 500 ns	~ 16 GB/s	bundled	—
SATA III	~ 50 μs	0.55 GB/s	bundled	—
100 GbE NIC, same rack	~ 5 μs	~ 12 GB/s	~ \$300–1500/NIC	—
Same-AZ network (cloud)	~ 500 μs	~ 10 GB/s	—	free
Cross-AZ network (cloud)	~ 2 ms	~ 5 GB/s	—	\$0.02/GB rt
Cross-region network (cloud)	~ 50–200 ms	varies	—	\$0.02/GB
Internet egress (cloud)	~ 50–300 ms	varies	—	\$0.05–0.09/GB
Media				
SRAM (CPU L1/L2/L3 caches)	< 1 ns	> 200 GB/s	~ \$1000/GB	~ \$17/GB-mo
DDR5 DRAM	~ 80 ns	~ 80 GB/s	~ \$1.50/GB	~ \$0.025/GB-mo
DDR4 DRAM	~ 100 ns	~ 50 GB/s	~ \$0.70/GB	~ \$0.012/GB-mo
NAND (TLC, NVMe-class)	~ 10 μs	~ 14 GB/s	~ \$0.07/GB	~ \$0.0012/GB-mo
NAND (TLC, SATA-class)	~ 100 μs	0.55 GB/s	~ \$0.04/GB	~ \$0.0007/GB-mo
HDD (magnetic)	~ 10 ms	~ 0.25 GB/s	~ \$0.01/GB	~ \$0.0002/GB-mo
S3 Express One Zone backend	~ 5–10 ms	—	opaque	\$0.16/GB-mo
S3 Standard backend	~ 30–100 ms	—	opaque	\$0.023/GB-mo + \$0.40/M GET
S3 Glacier (cold archive)	minutes	—	opaque	\$0.001–0.004/GB-mo

Table 2.1: Cost components of a cloud-native data system, separated into interconnects (top) and media (bottom). The latency and bandwidth of any deployment are bounded by whichever of an (interconnect, medium) pair is slower or narrower. BOM is the rough bill-of-materials (silicon plus assembly) at retail-

OEM volumes; “bundled” components are integrated into the CPU or motherboard and not separately billed.

A few worked compositions ground the table. Local DDR5 over the memory bus runs at ~ 80 ns and ~ 80 GB/s – the bus contributes only 5 ns and matches the chip’s 80 GB/s peak, so DRAM dominates both numbers. NAND on a PCIe 5.0 NVMe drive runs at ~ 10 μ s and ~ 14 GB/s – the bus contributes 500 ns and 16 GB/s, but the NAND medium’s 10 μ s random-read latency dominates the result. S3 Standard at ~ 50 ms is dominated by the network: a same-AZ round trip is ~ 500 μ s to set up, but the backend itself round-trips across several layers of replicated storage and takes tens of milliseconds. Pairings with the same interconnect therefore differ wildly in experienced performance depending on the medium attached, and vice versa – which is exactly what makes “the cache”, a layer that controls *both* sides of the pair, so leverageable.

Two observations matter for the rest of the dissertation.

First, the hierarchy is *exponentially* distributed, not linearly. A useful mental scale is to imagine that 1 ns equalled 1 second: in that scale an L1 hit is a single second, DRAM is a minute and a half, a local NVMe read is about three hours, a same-region S3 request is a year and a half, and a cross-continent request is more than six years. Caching pays off because the gap between adjacent tiers is exponential rather than linear, and crossing two tiers at once (DRAM versus S3, say) is closer to crossing several together than additive intuition would suggest.

Second, latency and bandwidth are independent. S3 has hundred-millisecond latency and multi-gigabyte-per-second aggregate bandwidth; DRAM has nanosecond latency and tens-of-gigabytes-per-second bandwidth. A cache that targets latency and one that targets bandwidth have different break-evens and different evaluation metrics. The three caches in this dissertation pick different points in this plane: LiquidCache and SqueezeCache target the latency–bandwidth-balanced regime of analytical pushdown caching, and Bf-Tree targets the latency-dominated regime of point lookups and range scans.

2.3.2 Bill of materials and operating cost

The last two columns of Table 2.1 decompose dollars into two components. *Bill of materials* (BOM) is the marginal manufacturing cost of one unit of capacity – the silicon, packaging, and assembly that produces a single gigabyte of the medium, or a single device of an

interconnect. BOM is invariant across deployments: a DDR5 DIMM costs roughly the same to manufacture whether it ends up in a personal workstation or in an AWS rack. *Operating cost* is what the operator (or paying user) actually spends per unit time, and has two regimes. For owned hardware, the operating cost is amortized BOM plus energy and datacenter overhead. For cloud-managed services, it is the provider’s published per-byte and per-request pricing.

Industry consensus on the useful lifetime of datacenter hardware has been roughly four years for the last decade, with recent disclosures from major hyperscalers extending this toward five to six years for general-purpose servers. In 2022, Microsoft, Meta, and Google each updated the depreciation life of server hardware in their financial filings – Microsoft from four to six years, Meta from 4.5 to 5, Google from four to six – citing improving hardware durability and rising embodied-carbon cost as the joint motivations. Table 2.1 uses sixty months (five years) as the amortization denominator; deployments holding hardware longer see proportionally lower per-month costs.

The amortized BOM in the table is the *floor* of what a particular medium costs to operate. Cloud-managed offerings sit far above it. DDR5 amortizes to roughly \$0.025/GB-month at BOM; provisioned in a cloud VM, the same memory rents for \$3–4/GB-month – a hundredfold markup that pays for power, cooling, networking, redundancy, support, and provider margin. NAND in cloud-attached NVMe amortizes to ~ \$0.001/GB-month at BOM; cloud-provided NVMe rents for \$0.10–0.20/GB-month, another hundredfold gap. HDD-class storage amortizes to ~ \$0.0002/GB-month; S3 Standard at \$0.023/GB-month is a hundredfold markup but bundles eleven-nines durability, automatic geo-replication, request-level access control, and the network plumbing to serve any client globally. The markups look extreme on paper but reflect real value: every cloud byte arrives ready to use, while every owned byte requires hardware operations, replication design, and physical-plant overhead the table cannot price.

Per-request and per-byte transfer charges are the second source of cloud operating cost, and they sit alongside the per-GB-month figures in the table. S3 GET requests cost \$0.40 per million, PUT requests an order of magnitude more. A workload issuing a million GETs per day spends ~ \$12/month on requests alone, often rivalling the storage bill on small-object workloads – the immediate economic argument for batching range reads, and indirectly for caching, since every cache hit avoids a paid request. Data transfer is highly directional: same-AZ is free, cross-AZ within a region is \$0.02/GB round-trip, cross-region is \$0.02/

GB, public-internet egress starts at \$0.05/GB and rises. A cache server placed in the same AZ as compute is therefore free at the network layer; one in a different region multiplies its bandwidth bill substantially, and offerings such as Cloudflare R2 reshape the equation by charging zero for egress entirely. Finally, *serverless* pricing replaces the GB-month axis with a GB-second axis: state stored in an ephemeral function pays only for active time, a model that favors bursty or sparsely accessed workloads and motivates systems such as CompuCache [188] and InfiniCache [171].

2.3.3 Reading the numbers

Three rules of thumb tie the table together and motivate the design choices in the rest of the dissertation.

The gap between tiers is exponential, not linear. Caching always pays off when the next tier is one full order of magnitude slower. Most adjacent pairs in Table 2.1 are. The interesting design question is rarely “should we cache?” but “at which tier?” – because each tier brings its own bandwidth, BOM, and operating-cost characteristics.

Latency and bandwidth are independent. A cache that targets latency and one that targets bandwidth have different break-evens and different evaluation metrics. LiquidCache and SqueezeCache target latency–bandwidth-balanced analytical pushdown; Bf-Tree targets latency-dominated point and range workloads.

Cloud pricing reshapes the question. Two pairings that look equivalent on latency and bandwidth may differ by a factor of ten in dollar cost. Per-request charges argue for batching; per-byte transfer charges argue for filter pushdown; per-GB storage charges argue for compression and squeezing. The three axes of the White-Box cache – representation, content, and granularity – map almost one-to-one onto these three cost gradients.

2.4 Three layers of opacity, revisited

With the vocabulary of this chapter in place, we can restate the framing of the introduction more precisely. A conventional cache sits at one stage of the query lifecycle – the scan in Figure 2.1 – in one of the architectures in Figure 2.2, and treats its contents as opaque bytes. It does not see that the bytes are Parquet rather than ORC, or that the Parquet pages would decompress an order of magnitude faster if reorganized. It does not see that the query references only the day-of-week field of a datetime column, or only a prefix of a string, or

only one path through a Variant. It does not see that hot records and cold records share a page, or that a single update will force the whole page to be written back. The next chapter (Chapter 3) develops the shared design vocabulary for a cache that is given permission to see; the three chapters after it describe what changes along each axis in turn.

Chapter 3

The White-Box Cache: A Reference Design

The introduction argued that a cache, in a modern data system, should be allowed to see inside the data it holds, and identified three axes – representation, content, granularity – along which that permission pays off. The background chapter then laid out the technical landscape any such cache must inhabit: the query lifecycle, the architectures caches deploy in, the cost gradients that make caching worth doing in the first place, and the three opacities a conventional cache accepts.

What remains, before the three system chapters, is a shared design vocabulary. Liquid-Cache (Chapter 4), SqueezeCache (Chapter 5), and Bf-Tree (Chapter 6) each attack one axis, but they are not three unrelated systems sharing a buzzword. They obey the same reference architecture, communicate with the engine above through the same family of channels, and make the same kind of trade-off between cache fidelity and downstream re-fetch cost. This chapter develops that shared design before the chapters that instantiate it, so that the system chapters can refer to a common contract rather than each re-inventing one.

The chapter is short by design. We describe (1) the reference architecture of a White-Box cache and how it differs from a black-box one (Section 3.1); (2) the channels through which the cache learns what the workload needs and reports back what it can answer (Section 3.2); (3) the fidelity spectrum a White-Box cache exposes, generalizing the binary keep-or-evict decision of conventional caches (Section 3.3); (4) how the three axes compose within a single deployment (Section 3.4); and (5) what we explicitly do *not* try to do, separating White-Box caching from adjacent ideas such as result caching and materialized views (Section 3.5).

3.1 A reference architecture

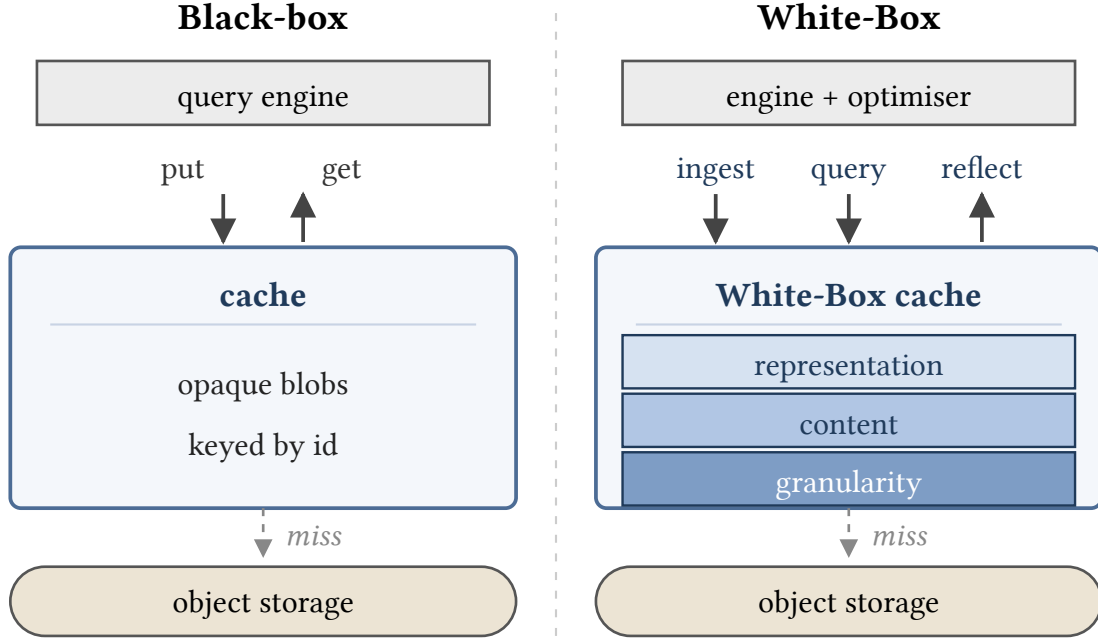


Figure 3.1: Reference architecture of a White-Box cache. A conventional cache (left) accepts an opaque blob keyed by an opaque identifier and returns the same bytes on a hit; the workload runs entirely above it. A White-Box cache (right) accepts a typed columnar value together with workload context – the schema it belongs to, the predicates that will be evaluated against it, and the lineage of operations the cache is allowed to anticipate – and returns either filtered bytes (representation), a partial representation that still answers the predicate (content), or a record-granular subset of the page it logically belongs to (granularity). The three system chapters of this dissertation instantiate one of the three internal layers (representation, content, granularity) each.

A black-box cache has two arrows on its interface, both opaque. A `put(key, blob)` arrow brings bytes in; a `get(key)` arrow takes the same bytes out. Whatever policy the cache runs – LRU, ARC, S3-FIFO – it operates on the keys, never on the blobs. Everything sophisticated happens above the cache, in the engine that knows what the blobs mean.

A White-Box cache replaces those two arrows with three richer ones (Figure 3.1).

Ingest. Data still enters the cache, but accompanied by its schema and (in the disaggregated case) the engine plan that will consume it. The cache may immediately re-encode the data on ingest, as LiquidCache does, or defer the transformation until the workload reveals which encoding is worth paying for.

Query. A query is no longer a key. It is a small dataflow program – predicates, projections, simple aggregations, lineage hints – that the cache executes against its in-cache representation. A White-Box cache may answer the query against any of several internal representations (fully decoded, partially decoded, or fingerprint-only), choosing the most compact one that suffices.

Reflection. A White-Box cache reports back more than “hit” or “miss.” It may return a partial answer (a fingerprint result that requires the engine to refine on disk), a typed result (an extracted day-of-week instead of a full datetime), or a fully materialized batch. The engine is expected to handle the cache as an active participant rather than a passive store.

The three chapters that follow each thicken one of the three internal labels in Figure 3.1. LiquidCache turns the box labeled “representation” into an adaptive cascade. SqueezeCache turns the box labeled “content” into a workload-driven choice of fragments. Bf-Tree turns the box labeled “granularity” into a variable-length mini-page whose size and contents are unrelated to the disk page it derives from.

3.2 Information flow: from optimizer to cache

The most subtle part of a White-Box cache is not what it does internally but how it learns what to do. A cache that asks for too much from the engine becomes engine-coupled and unportable; one that asks for too little falls back to a black-box cache that happens to use modern encodings. We use three communication channels, each instantiated by at least two of the three system chapters.

Schema metadata. The cache reads metadata attached to the columns it holds. LiquidCache uses metadata to remember which Parquet column chunk an in-cache batch derives from and which FSST symbol table to use. SqueezeCache extends this with *lineage metadata*: an optimizer rule attaches per-column usage tags (“only day-of-week extracted”, “only this JSON path referenced”) to the schema, so the cache can squeeze on ingest. Bf-Tree’s reference bits and look-ahead bytes are a degenerate version of the same idea, internal rather than engine-visible.

Pushdown protocol. Predicates, projections, and inexpensive aggregations are pushed into the cache rather than evaluated above it. LiquidCache evaluates filters against partially decoded data; SqueezeCache extends the same protocol with the right to return a *partial result* (a fingerprint pre-filter) that the engine must complete. Bf-Tree pushes negative-

search caching into the cache by storing *phantom records* – a small generalization of the pushdown idea to the KV index regime.

Access traces. Frequency and recency, the inputs that conventional caches use, remain available, but the White-Box cache also records *which fields* of each cached value were consulted. SqueezeCache uses two-byte expression IDs and lock-free CAS-updated headers to record this without slowing scans; Bf-Tree uses per-record reference bits. The trace influences future squeeze, hydration, and eviction decisions.

A White-Box cache is the union of these three channels. A cache that opts into only one is fine – LiquidCache works without lineage and Bf-Tree works without pushdown – but the union is what allows a single cache to attack all three axes at once.

3.3 Cache fidelity as a spectrum

Conventional caches expose a binary state per entry: present or absent. A White-Box cache exposes a continuum. We use four reference states across the three systems (Figure 3.2):

Full. The entry holds an exactly equivalent representation of the original data, possibly re-encoded for efficiency. Liquid arrays are full-fidelity by this definition: the encoding is lossless and any operation that would have worked against the original Parquet data still works against the Liquid layer.

Compressed. The entry is full-fidelity but bit-for-bit smaller, traded for some decoding cost. LiquidCache’s bit-packed Liquid arrays sit here; so do Bf-Tree’s prefix-compressed mini-pages.

Lossy. The entry retains only the fragments the workload demonstrably uses. SqueezeCache’s squeezed strings (prefix + fingerprint), squeezed datetimes (day-of-week), and shredded variants live here. A predicate against a lossy entry either answers definitively from cache or falls back to disk for the missing bytes.

Evicted. The entry has been written to a disk tier and freed from memory. Re-hydration is possible; if the next access path needs the full entry, the cache reads it back.

Each system chapter realizes a subset of this spectrum under its own names: SqueezeCache, for instance, labels the four states *uncompressed*, *compressed*, *squeezed*, and *evicted*, mapping its *squeezed* state onto the *lossy* state above.

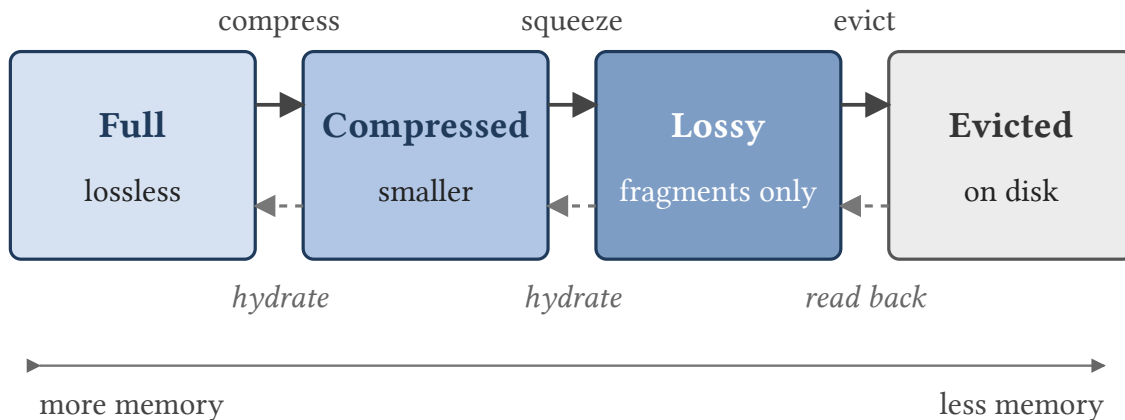


Figure 3.2: The fidelity spectrum a White-Box cache manages. A conventional cache moves entries between only the two end-states (Full and Evicted). A White-Box cache exposes the intermediate states and chooses between them per entry, driven by the access trace and the lineage metadata of Section 3.2. Each system chapter inhabits a different subset of the spectrum.

The choice between states is the cache’s core decision, taking the place that eviction occupies in a conventional cache. Where a black-box cache asks “which entries should I drop?”, a White-Box cache asks “for each entry, which state along the spectrum minimizes the expected total cost of serving future accesses?” SqueezeCache develops this idea most explicitly – its multi-state squeeze policy is exactly a navigation of this spectrum – but the same vocabulary applies to LiquidCache’s compressed-vs-uncompressed choice and to Bf-Tree’s choice between record-cache, write-buffer, and full-page mini-pages.

3.4 Composing the three axes

The three axes are independent. A single cached column may be simultaneously *representation-adaptive* (encoded as Liquid), *content-curated* (squeezed to a prefix and fingerprint), and *granularity-decoupled* (held in a mini-page sized to the hot records rather than to the underlying disk page). The three contributions of this dissertation each demonstrate one axis in isolation, but the framework permits any subset. LiquidCache and SqueezeCache already compose: SqueezeCache is built on LiquidCache and inherits its Liquid encodings as the lossless layer of the fidelity spectrum. Bf-Tree sits in the transactional regime and is

independent of the other two, but its variable-length buffer pool is conceptually the same mechanism a future analytical cache would need to support per-batch fidelity choices that are not aligned to fixed sizes.

The composition matters because the three axes attack three different costs. Representation reduces *CPU time per byte*: a query touches fewer bytes for the same logical work. Content reduces *bytes per entry*: a query holds the same number of entries in less memory. Granularity reduces *entries needed*: a query keeps the right subset of each page hot, freeing memory for other entries. A deployment that takes only one axis recovers one cost gradient. The full White-Box picture is a deployment that takes all three.

3.5 Non-goals

Three adjacent ideas are sometimes confused with White-Box caching, and it is useful to mark the boundaries explicitly.

Result caching stores the literal output of past queries and serves only repeats. A White-Box cache caches *data* rather than results, so a squeezed entry is reusable across queries that share the column but not the predicate – the common case in analytical workloads. Result caching can sit above a White-Box cache without conflict and the two are complementary.

Materialized views pre-compute named sub-queries. They require user or system declaration and are limited to the views chosen at design time. A White-Box cache infers what to retain at runtime, per column, from the workload that arrives.

Predicate caching records which row ranges have satisfied past predicates [142]. It is a degenerate special case of squeezing in which the squeezed representation is a single bitmask. SqueezeCache generalizes this to typed lossy representations chosen per datatype.

These three live above the abstraction this dissertation works on, not against it. A production deployment may run result caching for popular dashboard queries, a small materialized view for joins on the fact table, and a White-Box cache for the scan layer underneath; the three answer different questions.

3.6 From the framework to the chapters

The next three chapters take up the three axes in the order introduced here. Chapter 4 opens the representation axis: it asks what a cache can do once it picks its own encoding,

and demonstrates that an adaptive cascade reduces cache CPU by an order of magnitude on Parquet pushdown without growing memory. Chapter 5 opens the content axis: it asks how much of each cached entry the workload actually needs, builds the lineage machinery sketched in Section 3.2, and shows that retaining only the relevant fragments raises cache hit ratios fourfold. Chapter 6 opens the granularity axis in the transactional regime: it asks why cache pages must look like disk pages at all, and shows that a variable-length buffer pool yields a single index that is simultaneously the best at point lookups, range scans, and writes. Chapter 7 then revisits the cross-cutting engineering – the I/O substrate, the verified disk-management layer, the upstream contributions – and Chapter 8 and Chapter 9 close.

Chapter 4

LiquidCache: Adapting Cache Representation to the Workload

This chapter attacks the first of the three opacity axes from the introduction: *representation*. A conventional cache holds data exactly as the storage layer produced it – compressed Parquet [169] pages, in the file format the lakehouse [13] happens to write – and asks the rest of the stack to make do with whatever bytes that yields. That choice is opaque in two ways at once. The cache cannot decide how its bytes are encoded, so it cannot match the encoding to the operators reading from it; and the engines reading from the cache cannot avoid paying full decoding cost on every byte they touch, even when most of those bytes will be discarded by a filter. LiquidCache, the system described in this chapter, lifts both restrictions by transparently transcoding Parquet into an in-cache format whose encoding is chosen adaptively, in response to the workload running against it.

The opportunity is large because the cost being paid is large. Profiling filter pushdown on Parquet across the open lakehouse stack [20, 93, 133, 136] shows that data decoding – the conversion from Parquet’s storage-optimized layout into the Arrow-shaped [93] layout the query engine wants – consumes the overwhelming majority of the CPU time the cache spends on a typical pushdown query (Figure 4.1); the filter evaluation that decoding was supposed to accelerate occupies less than 10%. Filter evaluation pushed down to S3 Select [6] or Azure Data Lake Storage [125] hits the same wall, and several major cloud-native systems [8, 42] have responded by disabling pushdown for Parquet entirely.

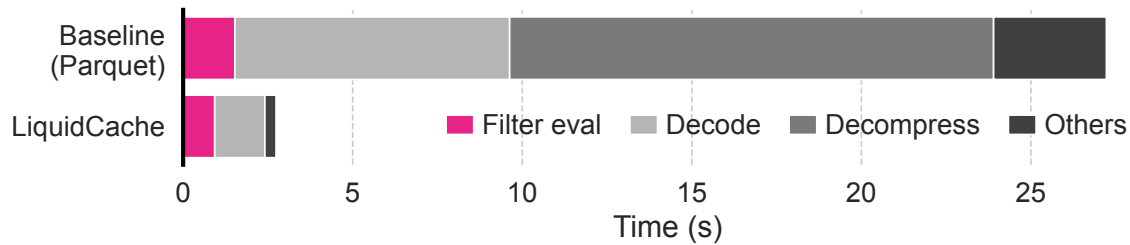


Figure 4.1: Time breakdown of filter pushdown for ClickBench Q22. Data decoding and decompression dominate CPU time on cached Parquet; filter evaluation takes less than 10%. This is the bottleneck LiquidCache attacks.

One response to this gap has been to replace Parquet with a faster file format. BtrBlocks [91], Vortex [156], Nimble [124], LanceDB [95], and others propose alternatives whose encodings are easier to decode. But the file format is a public interface across many engines, and the ecosystem has effectively locked in [61, 89, 185] to the encodings Parquet shipped with in 2013. A second response has been to push filters into specialized hardware – FPGAs [149], SmartSSDs [47], SmartNICs [88], DPUs [78] – but the hardware is bespoke and the deployment story complicates the very disaggregation it was meant to support.

LiquidCache takes a third path: leave the file format alone and change what happens inside the cache. On the first read, the cache transcodes Parquet into *Liquid*, an in-cache format that cascades dictionary encoding, FastLanes bit-packing [1], FSST string compression [25], ALP floating-point encoding [2], and frame-of-reference integer encoding – every layer independently decodable, so that a filter can run against whichever layer is most efficient for the predicate at hand. Filter evaluation can operate on encoded dictionary keys, on dictionary values alone, on bit-packed integers, or on fully decoded Arrow, whichever permits the smallest decoding work. The Liquid format lives only inside the cache layer, so the rest of the ecosystem continues to see Parquet, and the cache is free to evolve its encodings without coordinating with anyone. Transcoding overhead is hidden through fine-grained, batch-level conversion that runs asynchronously in the background, exploiting the idle CPU cycles that analytical workloads leave between operator stages.

We implemented LiquidCache as a disaggregated cache server integrated with Apache DataFusion [93] and evaluated it on ClickBench [35, 36] and TPC-H. LiquidCache reduces cache CPU time by up to 10× compared to Parquet pushdown, cuts network traffic by two orders of magnitude compared to non-pushdown caches, and matches Arrow-in-memory caches in latency at a quarter of the memory footprint. Several techniques developed for

LiquidCache – a rewritten Parquet decoding pipeline and a new Arrow StringView representation – have been upstreamed and are now defaults in DataFusion, Arrow, and Parquet.

The contributions of this chapter are:

1. The first disaggregated pushdown cache for cloud-native analytics, built on production-grade systems (Section 4.2).
2. The Liquid in-cache format, co-designed with filter evaluation so that predicates can be evaluated against whichever encoded layer suffices (Section 4.3).
3. A demonstration that a cache can adopt state-of-the-art encodings without breaking the open file-format ecosystem.
4. An evaluation that explores the design space of disaggregated caches on ClickBench and TPC-H and quantifies their trade-offs (Section 4.4).
5. Upstream contributions to Apache DataFusion, Arrow, and Parquet that propagate the underlying techniques into systems beyond this dissertation (Section 7.6).

The chapter opens with the LiquidCache-specific background that Chapter 2 and Chapter 3 did not cover, then presents the architecture (Section 4.2), the Liquid format and its co-design with filter evaluation (Section 4.3), the empirical evaluation (Section 4.4), related work, and a transition to the next chapter.

4.1 Background

Chapter 2 introduced the broad context that LiquidCache assumes: data lives in Parquet on object storage; caches sit between compute and storage in private, distributed, or disaggregated topologies (Figure 2.2); queries flow through parse, plan, optimize, scan, decode, and execute stages (Figure 2.1); filter pushdown is an established optimization; and decoding consumes the bulk of scan-stage CPU on Parquet workloads (Section 4.1.1 briefly recapitulates the four-step pushdown pipeline). Chapter 3 then developed the cache-design vocabulary – the reference architecture, the information-flow channels, and the fidelity spectrum – against which LiquidCache instantiates the representation axis. LiquidCache deploys as a disaggregated pushdown cache and inherits the latency/bandwidth characteristics of that topology – a same-region network hop on every access (Table 2.1), traded for independent scaling of cache and compute. This section adds the two pieces of LiquidCache-specific background neither of those chapters covered: a slightly tighter look at the

Parquet-pushdown bottleneck Figure 4.1 targets, and the latency regime LiquidCache is designed to serve.

4.1.1 Filter pushdown on Parquet

Evaluating filters against a Parquet file involves four steps: (1) decode the Parquet metadata to locate the relevant data pages; (2) decompress those pages with a general-purpose codec such as LZ4 or Zstd; (3) decode the Parquet-encoded columnar data into an in-memory representation; (4) evaluate the predicates against the decoded data. As shown in Figure 4.1, the dominant cost is decoding – the third step – not the fourth. Because filters cannot be evaluated directly on Parquet’s compressed format, decoding is unavoidable and must be performed regardless of how selective the predicate turns out to be. Several cloud storage systems either do not support filter pushdown on Parquet files [8, 42, 125] or see degraded performance when attempting it [180].

4.1.2 Target workloads: low-latency analytics

LiquidCache targets low-latency workloads – dashboards, anomaly-detection pipelines, interactive queries, LLM knowledge retrieval – that demand sub-second response time. User studies show that response times above roughly 500 ms significantly hinder interactive exploration [113]. Fetching data from object storage already incurs more than 100 ms of first-byte latency [54], and Parquet metadata often requires two network round-trips before execution can even begin. Large-scale deployments of systems such as Dremel [123] and Presto [161] report orders-of-magnitude slowdowns when data is accessed over the network. These observations motivate caching techniques that minimize both remote I/O and decoding overhead.

4.2 Architecture

LiquidCache is a disaggregated pushdown cache – the third architecture in Figure 2.2 – hosted on commodity VMs and sitting between compute nodes and object storage. Cache servers are provisioned for memory rather than CPU, the inverse of compute, so each can serve many compute nodes that scale independently. Filter pushdown into the cache is what allows this asymmetry to pay off: by evaluating predicates close to the data, the

cache emits only the filtered result, sparing the inter-tier network the bandwidth a non-pushdown cache would consume.

This section presents the detailed design of LiquidCache. 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, and finally discuss caching mechanisms and policies.

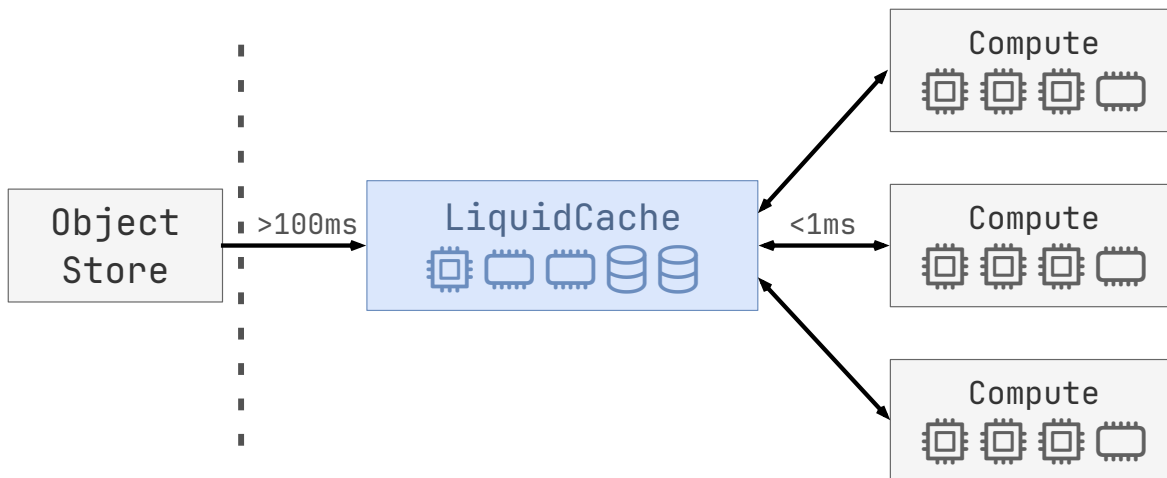


Figure 4.2: LiquidCache architecture – LiquidCache runs on a commodity, elastically-scaled cache server, physically close to the compute servers that handle query execution.

4.2.1 System components

As shown in Figure 4.2, LiquidCache is a cache server that sits between compute nodes and object storage. Object storage is slow, with first-byte latencies over 100ms [54]. 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 [125] or are slower with pushdown enabled [180]. LiquidCache instead uses common object storage APIs and supports any storage provider.

The cache server provisions SSD-based elastic storage to store Parquet data retrieved from object storage. The cache server can seamlessly provision more elastic storage without downtime when additional storage capacity is required. The bundled CPU on the cache server manages network communication between compute nodes and object storage, and

evaluates filters received from compute nodes. However, unlike compute nodes performing computationally intensive operations, the cache server’s CPU requirements are minimal and cost-efficient.

Communication between the cache and compute servers occurs via Arrow Flight, a high-performance network protocol built on gRPC. Arrow Flight enables zero-copy data transfer by allowing compute nodes to directly interpret data from network buffers without de/serialization – a major cost in data-intensive systems [183]. Compute nodes send SQL queries containing pushdown filters to the cache, which evaluates the filters and returns matching results.

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

4.2.2 Life of a query

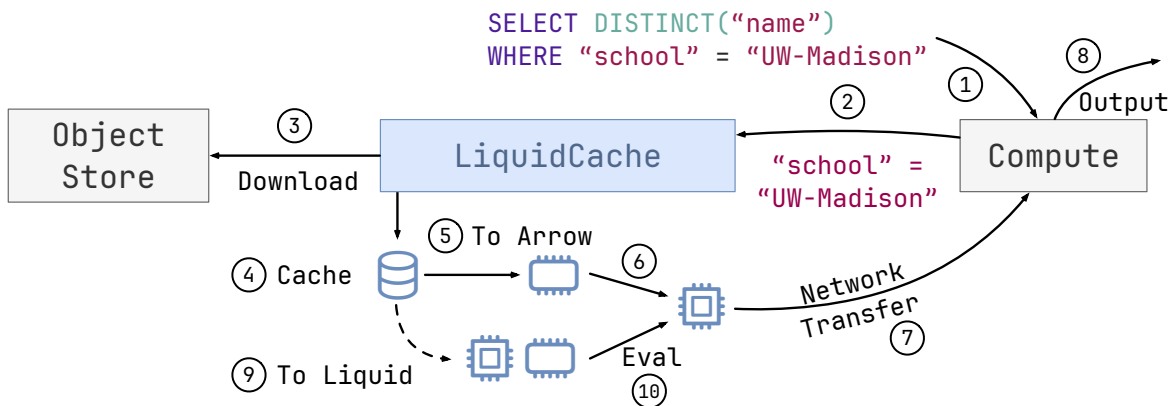


Figure 4.3: The life of a query in LiquidCache – Step 9 transcodes the Parquet into Liquid format in the background, allowing efficient predicate evaluation on a cache hit.

With a disaggregated cache, the cache server and compute server cooperate to execute a query. As in data lake architecture [13], the catalog server sends a pair of SQL queries and file locations (on object storage) to the compute server. A conventional compute server will download the data from object storage and process the query on its own hardware. This section discusses the life of a query of LiquidCache, as shown in Figure 4.3, 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 4.1 and discussed in Section 4.1.1, data decoding and decompression dominate 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 4.3, the data scanning operator corresponds to scanning the table with the filter `"school = UW-Madison"`, and the compute operators correspond to the `"DISTINCT"` aggregation.

LiquidCache implements optimizer rules that decide which operators should be pushed down to the cache server, and encodes the query plan into protobuf messages and sends to the cache server for execution.

4. Execute the query.

The cache server processes the SQL query with an embedded query engine, first checking for liquid-encoded data in its in-memory cache. The Liquid format (Section 4.3) provides high compression ratios and efficient predicate evaluation. If not in memory, the cache server checks its local disk cache. As a last resort, it downloads the data from object storage. For data not already in Liquid format, the cache server submits a background task

to transcode it, enabling efficient predicate evaluation on subsequent cache hits. Lastly, the cache server streams the filtered results to the compute server via Arrow Flight, processing and sending data batches in parallel.

5. Return results.

Finally, the filtered data is streamed back to the compute server, which continues processing the remainder of the query plan. Step 9 in Figure 4.3 shows that the cache server concurrently transcodes newly fetched Parquet data into Liquid format so that subsequent queries can evaluate predicates without repeating the conversion.

4.2.3 What to push down?

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

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

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

Cost-based pushdown. Currently, LiquidCache always pushes down the operators above, and more sophisticated cost-based policies [117, 180] are complementary and left for future work.

4.2.4 Local mode

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

4.2.5 Cache mechanisms

Cache Parquet bytes on cache-local disk. On cache misses, the server downloads the requested data range from object storage based on the query and Parquet metadata. It merges multiple small ranges into larger ones to minimize object store requests, which are billed

per request rather than by volume [54]. 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 the scope of this dissertation; practical deployments can choose any caching/eviction policy, or simply provision more elastic storage.

Cache Liquid data in memory. LiquidCache caches each column in batches of 8192 rows so that queries load only the required pieces. Entries are identified by file name, row group, column index, and row number. A column-level LRU policy [14, 15] exploits the common pattern that batches within a column are accessed together while different columns are independent. Evicted batches are written to disk using a FlatBuffers-style [63] representation to avoid future deserialization overhead.

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

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

4.3 Liquid Format

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

4.3.1 Dilemma: Decoding speed vs. memory usage

Decoding consumes significant CPU cycles [91], while DRAM already accounts for nearly half of data-center cost [104]. Highly compressed formats like Parquet decode slowly

but use little memory, whereas in-memory layouts such as Arrow decode instantly but quadruple space usage. LiquidCache resolves this tension by co-designing its encoding with the filter evaluation process. All encodings permit independent decoding of each element so that values that fail early filters are never materialized. Techniques such as dictionary encoding and bit packing are combined in a cascade so that only the minimal representation necessary for the current filter stage is produced (Section 4.3.3).

4.3.2 Liquid data representation

LiquidCache combines several state-of-the-art data encoding techniques for high compression ratios and efficient filter evaluation. These include FSST [25] for string compression, FastLanes [1] for integer bit-packing, and standard encoding techniques like dictionary encoding and FoR (frame-of-reference) encoding.

This subsection presents LiquidCache’s current encoding strategy. While we focus on string and integer encodings, LiquidCache can incorporate any appropriate columnar encoding technique that preserves independent element decodability, such as those in [91]. The system’s flexible architecture allows dynamic updates to its encoding chain as new techniques emerge.

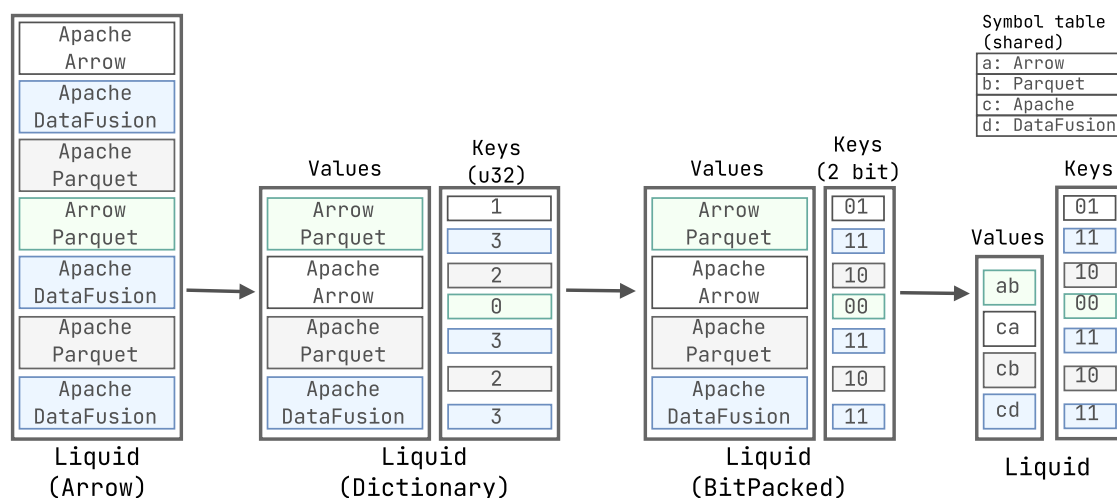


Figure 4.4: How LiquidCache encodes string arrays – The leftmost diagram shows the Arrow string representation, it will first be converted to dictionary encoding to eliminate the repeated strings, and then the keys are compressed with bit-packing based on the dictionary size, finally the dictionary is compressed with FSST encoding. Each of the intermediate representations can be used for filter evaluation. The corresponding data size and transcoding time can be found in Section 4.4.7.

4.3.2.1 Encode strings.

LiquidCache encodes string arrays (Figure 4.4) in three cascading steps: first use dictionary encoding to deduplicate the strings, then use bit-packing to compress the keys, and finally use FSST encoding to compress the dictionary values.

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, $\text{ceil}(\log_2(n))$ bits where n is the number of unique strings. For example, with 10 unique strings, each key requires $\text{ceil}(\log_2(10)) = 4$ bits. LiquidCache leverages FastLanes [1] encoding to efficiently bit-pack these keys using the minimal required bit width.

For dictionary value compression, LiquidCache employs FSST encoding [25], 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.

4.3.2.2 Encode integers.

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

4.3.2.3 Encode floating numbers.

For floating-point columns, LiquidCache adopts the ALP scheme [2]. When values are whole numbers, they are converted to integers using PseudoDecimal [91]; otherwise ALP applies vectorized compression to the significant bits.

4.3.2.4 Lossy encoding.

Lossy schemes [110, 134, 159] can further reduce size when applications tolerate small errors. Because LiquidCache caches the logical equivalent of Parquet files, we currently store only lossless encodings. Users may, however, apply lossy compression when generating the Parquet data itself.

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

4.3.3 Co-design with filter evaluation

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

4.3.3.1 Selective decoding.

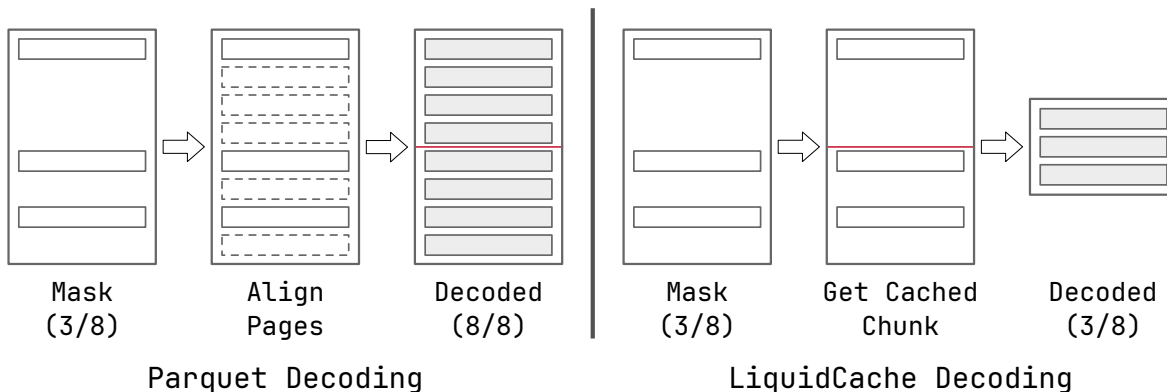


Figure 4.5: LiquidCache’s selective decoding compared to Parquet’s decoding – Parquet (left) has to decode an entire page even if just one element is needed. LiquidCache (right) allows each element to be decoded independently.

Filter pushdown has two stages: (1) build filter mask over predicate columns, and (2) build output over projection columns. Selectively decoding applies the filter mask from stage 1 to selectively decode the projection columns where the corresponding bit in the filter mask is true [158]. While selective decoding has been applied to Parquet in many systems [27, 93], its benefits are limited by Parquet’s page-based compression scheme. As illustrated in Figure 4.5, even when the filter mask selects 3 out of 8 elements, a Parquet decoder must decode the entire page containing all 8 elements since it cannot decode individual elements within a page. Therefore, selective decoding only benefits Parquet in cases where entire pages can be skipped. In contrast, LiquidCache pushes the selective decoding to its extreme, where an element is decoded if and only if it passes the previous filters.

4.3.3.2 Late filter materialization.

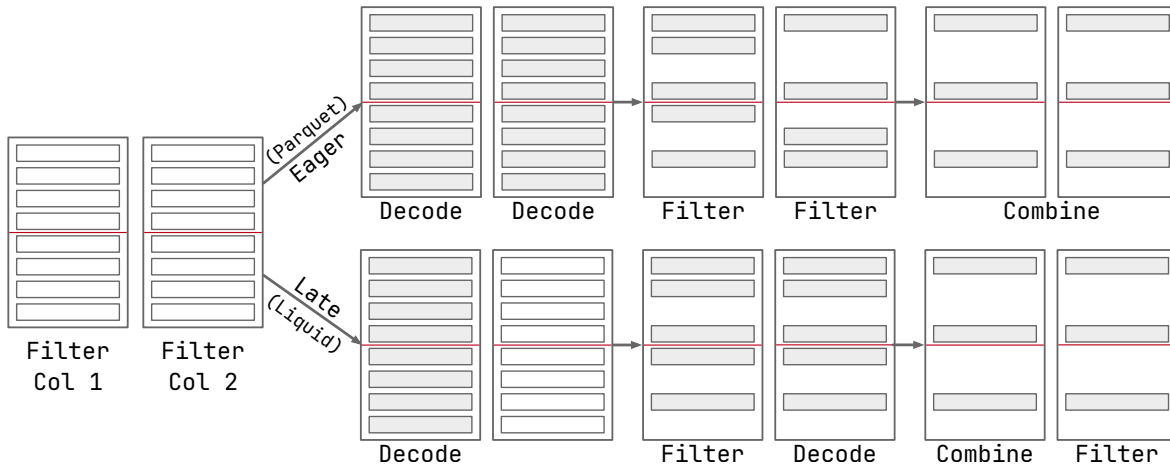


Figure 4.6: LiquidCache’s late filter materialization compared to Parquet’s eager materialization – The first row shows the filter evaluation process of Parquet, and the second row shows LiquidCache. With two filter columns, Parquet eagerly materialize both columns, while LiquidCache only materializes the first column and use the filter mask to selectively decode the second column.

Late filter materialization takes one step further by applying selective decoding to the filter evaluation itself (i.e., stage 1). It targets scenarios with multiple chained filters across different columns – a pattern commonly found in analytical queries. Figure 4.6 illustrates late filter materialization in action. In the eager approach without late filter materialization (first row), the system first decodes all filter columns, then evaluates predicates on each column independently, and finally combines the resulting boolean masks to determine which rows satisfy all conditions. With late filter materialization enabled (second row), the system processes columns sequentially – it decodes the first filter column, evaluates its filter to produce a boolean mask. Then, this mask is used to selectively decode only the qualifying elements from the second filter column. This cascading process means each subsequent column needs to decode progressively fewer elements as more filters are applied. Late filter materialization reduces both decoding work and filter evaluations by processing only data that passed previous filters.

LiquidCache leverages bit deposit operations [108] to accelerate the bit mask manipulations required for late filter materialization.

4.3.3.3 Evaluate filters on encoded data.

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

4.3.3.4 Evaluate filters on partially encoded data.

Not all filter operations can be evaluated on encoded data. Substring pattern matching, for instance, requires access to the fully decoded strings. However, LiquidCache's cascading encoding strategy (Section 4.3.2) allows decoding to proceed only as far as necessary for each filter type. For example, decoding stops after dictionary values are available for substring searches, avoiding the overhead of key decoding and dictionary materialization. For prefix matching, only the initial bytes of each string need to be decoded. In addition to filter evaluation, LiquidCache also transmits the partially encoded data (when applicable) over the network to reduce the network traffic.

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

4.3.4 Efficient transcoding

When LiquidCache reads Parquet files from object storage, it must transcode them into Liquid format to leverage the abovementioned optimizations. While transcoding between

formats incurs overhead, LiquidCache employs several techniques to make this process efficient. This subsection examines three 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.

4.3.4.1 On demand fine-grained transcoding.

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

LiquidCache takes a more fine-grained approach by transcoding data on-demand at column granularity. Only those columns are transcoded into liquid format when a query requests specific columns. This selective transcoding is further optimized through predicate pushdown – LiquidCache only transcodes data batches that pass filter predicates, avoiding unnecessary work on rows that will be filtered out. While transcoding does incur an upfront cost, it is a one-time investment that benefits all subsequent queries accessing the same data. This aligns well with typical analytical workload patterns where queries are highly repetitive [137], allowing the transcoding cost to be amortized across many query executions.

4.3.4.2 Background transcoding.

Rather than blocking query processing while transcoding data from Arrow to Liquid format, LiquidCache performs this conversion asynchronously in the background. By scheduling transcoding during periods of lower CPU utilization, LiquidCache minimizes 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. We will evaluate this in Section 4.4.8.

Third, analytical systems typically exhibit spiky workload patterns [137] requiring CPU over-provisioning to handle peak loads. LiquidCache exploits these quieter periods to perform transcoding work efficiently using otherwise idle CPU cycles.

4.3.4.3 Deep integration with Parquet reading.

In addition to the design optimizations discussed above, LiquidCache also employs several engineering optimizations that directly optimize the transcoding process. LiquidCache rewrites the core of the Parquet reader such that the decoding process is aware of a later transcoding. For example, LiquidCache implements a new Arrow string representation called `StringView`, which allows LiquidCache to reuse the Parquet decoding buffer as the string buffer for the dictionary encoding, which saves multiple memory copies for large string arrays. Our `StringView` implementation has been upstreamed to Parquet and is now the default string representation for DataFusion [70, 71]. 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.

4.3.5 Discussion

Comparison with other modern data formats. Modern file formats such as BtrBlocks [91], LanceDB [95], Nimble [124], and Vortex [156] 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.

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 [89].

4.4 Evaluation

Our evaluation answers the following questions:

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

4.4.1 Implementation details

We implement LiquidCache on Apache DataFusion [93], a high-performance analytical engine consistently ranking among top performers for Parquet workloads [94]. Our implementation consists of approximately 22k lines of Rust code, with an additional 5k lines contributed to upstream DataFusion, Arrow, and Parquet. LiquidCache implements a physical optimizer rule that replaces the Parquet reader with LiquidCache’s reader, enabling integration with existing systems [11, 67, 80, 92, 95, 165] through minimal code changes – typically under 10 lines.

4.4.2 Evaluation setup

We evaluate LiquidCache using ClickBench [35, 36], an industry-standard analytical benchmark with 15GB (100M rows) of real-world web analytics data. This benchmark includes many short, selective queries typical of low-latency workloads. We focus on queries with complex filter patterns and variable-length fields [138].

Table 4.1 details characteristics of our selected query subset: query ID, columns projected, data size processed, filters applied, selectivity (percentage of rows passing filters), and data types. The queries span diverse scenarios from single-column filters to complex multicolumn predicates, with selectivities ranging from highly selective ($<0.01\%$) to broad (13.2%), and data sizes from 0.2GB to 14.8GB.

Experiments run on CloudLab [51] 6525 machines with 16 cores (32 threads) x86_64, 128GB RAM and SATA SSD, using 10Gbps network to simulate typical cloud environments. We disabled TLS encryption and used the default Arrow Flight without compression to minimize overhead. Each query was executed five times, with results averaged across the final three runs for warm-up. All latency measurements represent end-to-end execution time from SQL parsing to result retrieval. Unless otherwise specified, we report LiquidCache benchmarks on fully transcoded format.

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

Table 4.1: Characteristics of selected ClickBench queries – Each query is described by its ID, number of columns projected, total data size processed, number of filters applied, selectivity (percentage of rows that pass the filters), and the data types involved. Q20 originally projects all 104 columns but is optimized to a simple count after projection and aggregation pushdown.

4.4.3 Baseline implementations

We implement three representative baselines matching the disaggregated caching topology of Figure 2.2, representing key industry approaches: file serving, Parquet filter pushdown, and Arrow filter pushdown.

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

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

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

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

Other approaches exist between Arrow (pushdown) and Parquet (pushdown), such as pushdown systems caching uncompressed (or lightly compressed) Parquet data [13]. We do not evaluate these separately: by construction they trade memory for compute somewhere between the two extremes, so their results would fall between the Arrow (pushdown) and Parquet (pushdown) data points we already report and would add no qualitative insight beyond the bracket those two baselines establish.

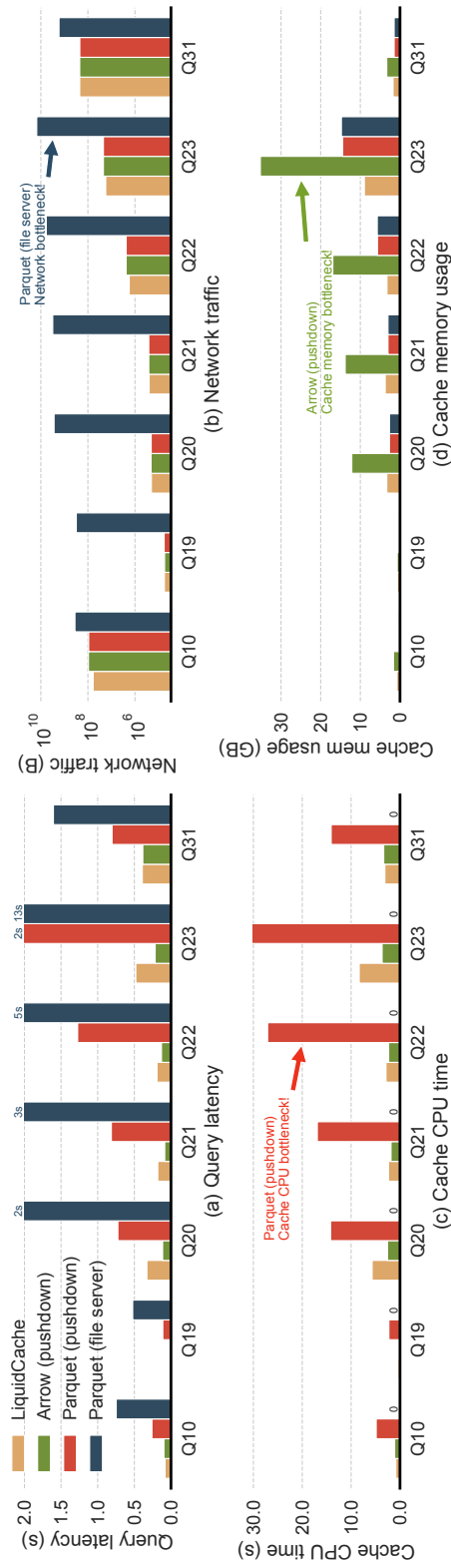


Figure 4.7: ClickBench results comparing LiquidCache with three baselines (lower is better). Parquet (file server) is limited by network traffic, Parquet (pushdown) by CPU usage, and Arrow (pushdown) by memory footprint; LiquidCache avoids all three bottlenecks.

4.4.4 Overall results

Our first experiment (Figure 4.7) compares LiquidCache with the baselines on four important metrics: latency, network traffic, cache CPU time, and cache memory usage. The x-axis shows the query id as described in Table 4.1, and the y-axis shows the corresponding metrics; lower is better.

4.4.4.1 Latency

Figure 4.7(a) shows Arrow (pushdown) and LiquidCache perform similarly, both significantly outperforming Parquet (pushdown) and Parquet (file server). Parquet (file server) consistently shows worst performance, with latencies up to 13 seconds. Parquet (pushdown) performs better but still reaches 2 seconds, while LiquidCache and Arrow (pushdown) never exceed 0.5 seconds. LiquidCache achieves latency comparable to Arrow (pushdown) on most queries but shows higher latency on Q20 and Q23, which involve filters on large string columns requiring LiquidCache to decode data before applying filters, while Arrow (pushdown) applies filters directly on in-memory Arrow data.

4.4.4.2 Network traffic

Figure 4.7(b) shows corresponding network traffic on a logarithmic scale (lower is better).

The three pushdown-enabled systems (LiquidCache, Arrow (pushdown), and Parquet (pushdown)) exhibit similar network traffic patterns since they transfer only records matching filter predicates. LiquidCache achieves slightly lower traffic by transferring partially-encoded data (e.g., dictionary-encoded Arrow) over the network. In contrast, Parquet (file server) transfers orders of magnitude more data by sending entire unfiltered Parquet chunks. This gap is particularly pronounced for highly selective queries (Q19-Q23), where pushdown systems transfer only small fractions of records. The difference narrows for less selective queries (Q10, Q31), where most data passes filters. Excessive network traffic saturates bandwidth and incurs substantial CPU overhead in cache and compute nodes' network stacks for packet processing.

4.4.4.3 CPU time on cache server

Cache servers with limited CPU power struggle with computationally intensive Parquet decoding, as noted in Section 4.1.1. We instrumented our implementation to measure CPU time consumed by pushdown operations, including decompression/decoding and filter/aggregation evaluation.

Figure 4.7(c) compares the total CPU core time consumed by each system on the cache server. Parquet (pushdown) spends significantly more CPU time than other baselines since it must fully decode Parquet data on the cache server. In contrast, Parquet (file server) uses minimal CPU, simply serving files without processing. Among pushdown-supporting systems, Arrow (pushdown) is most efficient since it operates directly on in-memory data without decoding. LiquidCache achieves comparable CPU efficiency to Arrow (pushdown) despite working with compressed data. This efficiency stems from Liquid’s faster decoding scheme (Figure 4.10) and LiquidCache’s ability to skip unnecessary decoding through tight filter pushdown integration (Figure 4.13).

4.4.4.4 Cache memory consumption

For fair comparison, each experiment begins with an empty cache and reports memory used for caching, excluding runtime data structures.

As shown in Figure 4.7(d), Arrow (pushdown) has the highest memory usage, while LiquidCache maintains a memory footprint comparable to Parquet. For Query 23 alone, Arrow consumes more than 30 GB of cache memory. Parquet variants (pushdown and file server) show identical memory usage as they cache the same compressed Parquet data. LiquidCache achieves comparable or better memory efficiency than Parquet, thanks to its advanced cascading encoding scheme. Despite its high compression ratio, LiquidCache preserves random access capabilities to individual data elements, as demonstrated in Figure 4.10.

4.4.5 TPC-H results

Although LiquidCache targets short, selective queries, we also benchmark TPC-H to evaluate larger-than-memory workloads. We run the scan-heavy queries (Q4,6,11,12,14,15,16,20) at scale factor 100, totaling 100 GB data, with only 24GB max system memory, including 8GB cache and 16GB for runtime data structures (e.g., join hash tables). We additionally

compare to Apache ORC [10]. Converting the dataset using the same Snappy compression yields a slightly better ratio for ORC (32%) than Parquet (34%), confirming prior observations [111, 184].

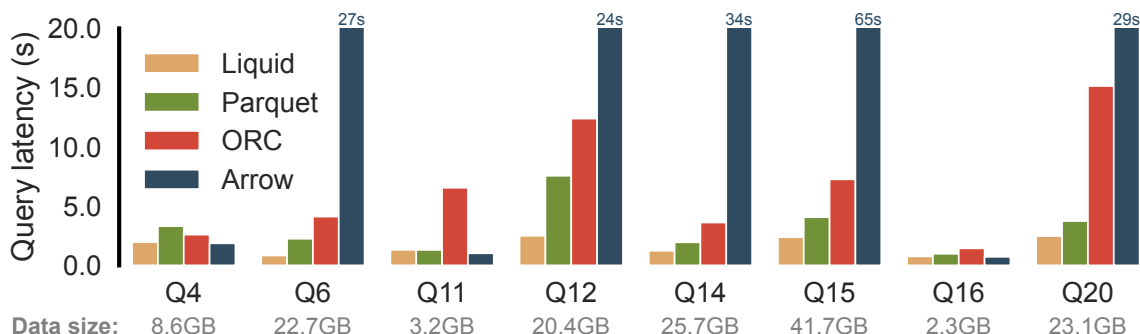


Figure 4.8: TPC-H SF100 (100GB data) with 8GB cache. Data size indicates query scan size, excluding runtime structures. Arrow is best when data fits in cache and worst otherwise. Parquet outperforms ORC due to better query engine integration. Liquid consistently delivers the best performance when the input does not fit in the cache.

Figure 4.8 shows the TPC-H SF100 results benchmarked with 8GB data cache, the y-axis shows the query latency, and data sizes indicate the corresponding query’s data size, excluding runtime structures. Queries have varying data sizes, from 2.3GB to 41.7GB. When the data size fits in cache, Arrow performs the best (similar to Liquid) and performs the worst otherwise. Although Parquet and ORC have similar compression ratio, Parquet generally outperforms ORC by a large margin due to its better query engine integration – both DataFusion and DuckDB best support Parquet while the former has minimal ORC support and the latter has no ORC support – highlighting the importance of query engine and data format co-design. Liquid consistently performs the best, thanks to its efficient encoding that reduces data size and filter-aware decoding that skips unnecessary decoding and file reads.

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

4.4.6 Decoding revisited

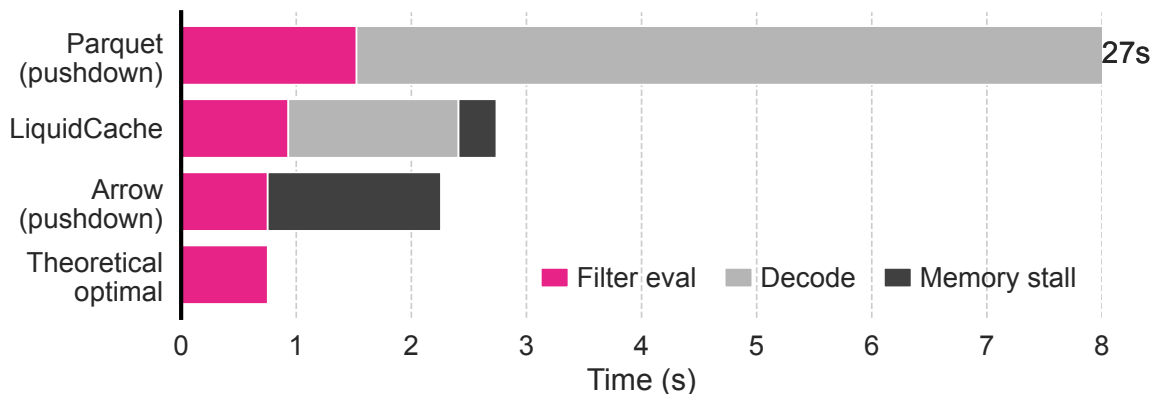


Figure 4.9: Decomposed CPU time spent on cache server for ClickBench Q22. LiquidCache spends similar total time as Arrow (pushdown), but primarily on decoding compressed data, while Arrow (pushdown) suffers from memory stalls due to scanning 4× more uncompressed data.

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

As shown in Figure 4.9, LiquidCache significantly reduces overall CPU time from 27s to 2.7s, reaching a similar overall time as Arrow (pushdown). While LiquidCache and Arrow (pushdown) spend comparable total execution time, they exhibit markedly different performance characteristics. LiquidCache dedicates most processing time to efficiently decoding compressed data. Thanks to its filter-pushdown co-design (Section 4.4.9), it only decodes the subset of data that passes filters. In contrast, Arrow (pushdown) experiences significant memory stalls due to operating on uncompressed data that is 4× larger. Both systems achieve similar filter evaluation times, approaching the theoretical optimum – which would evaluate all filters directly on encoded data, eliminating decoding overhead while maintaining a compact memory footprint.

4.4.7 Transcoding cost

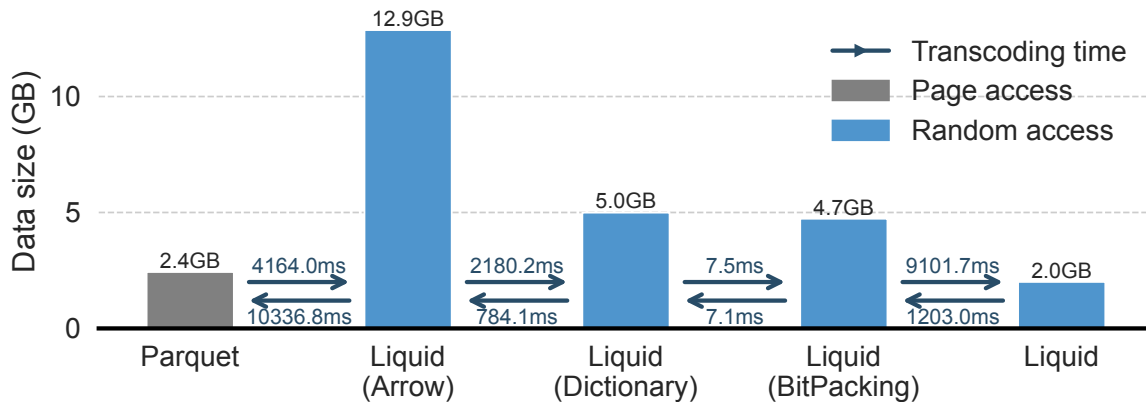


Figure 4.10: Memory usage and transcoding cost of “Title” column of the ClickBench – the largest column in the dataset. The bar height indicates the encoded data size, arrows denote transcoding time. Parquet only allows page-level access, while the rest of the encodings permit random element access.

Instead of requiring all data sources to produce Liquid-encoded data, LiquidCache progressively transcodes upstream Parquet data into Liquid on the fly, facilitating integration into existing systems. This section analyzes direct transcoding costs among Liquid’s different data representations, then demonstrates how LiquidCache effectively hides these costs from the critical query execution path through background processing.

Figure 4.10 presents a detailed analysis using the “Title” column from ClickBench – the dataset’s largest column. The figure shows memory consumption (y-axis) for different encoding formats (x-axis), with arrows indicating transcoding times between formats. While Parquet restricts access to page-level granularity, all other encodings support random access to individual elements. Our evaluation reveals several key findings:

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

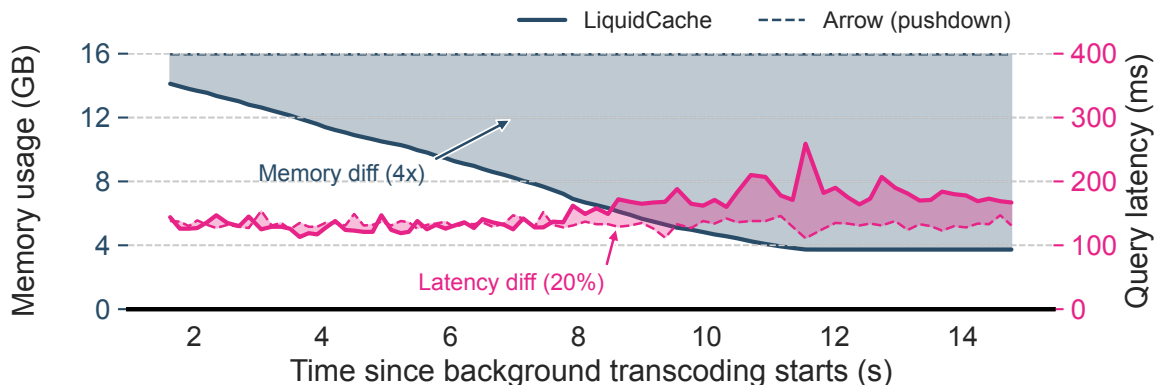


Figure 4.11: *LiquidCache’s latency and memory usage on Q21 – LiquidCache gradually transcodes Arrow into Liquid in background, avoiding latency spikes on the critical path of query processing.*

We analyze how LiquidCache handles transcoding costs during query execution using Q21 from ClickBench as a case study. This query contains multiple complex filters and demonstrates typical transcoding patterns observed across other queries. We restrict background processing to only four threads to simulate realistic cache server conditions with limited CPU resources. Real systems can use all idle cores for transcoding when under-loaded – a typical pattern in cloud analytical systems [137].

Figure 4.11 illustrates the relationship between transcoding progress, memory usage, and query latency for both LiquidCache and Arrow (pushdown). The x-axis represents time elapsed since transcoding begins, while the left and right y-axes show memory consumption and query latency, respectively. The shaded area represents the difference between LiquidCache and Arrow (pushdown).

On cache misses, LiquidCache first decodes Parquet data into Arrow format, then queues Arrow-to-Liquid transcoding as a background task while immediately proceeding with query execution. As background transcoding progresses, memory usage steadily decreases, achieving a 4× reduction compared to Arrow format, while query latency increases by only 20%. As discussed in Section 4.3.4, LiquidCache’s design ensures transcoding never blocks the critical query execution path, as queries can operate on any intermediate data without waiting for transcoding to complete. The near-identical initial query performance between LiquidCache and Arrow (pushdown) confirms this non-blocking behavior.

4.4.8 Cold run latency

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

Figure 4.12 compares execution latency between LiquidCache and baseline Parquet across different storage devices. The left region displays cold run latency, while the right shows stabilized warm run latency. Storage devices range from remote (S3-far: cross-continent us-west to eu-central) to local (Memory: kernel page cache). Intermediate options include S3/S3-express (nearest region: Nevada to Oregon), MinIO (same cluster), and SSD (locally attached storage). The y-axis shows latency with a horizontal line indicating transcoding time. LiquidCache(blocking) disables background transcoding, sequentially processing one batch at a time.

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

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

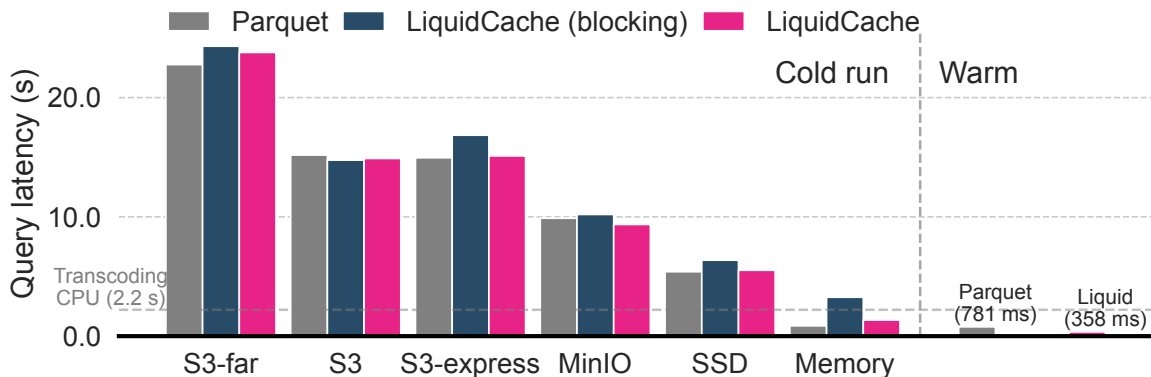


Figure 4.12: Impact of transcoding overhead. Baseline Parquet reads directly from files. LiquidCache(blocking) converts one batch at a time, whereas LiquidCache overlaps transcoding with I/O. Storage devices: S3-far (cross-continent), S3/S3-express (nearest region), MinIO (same cluster), SSD (local), Memory (page cache).

4.4.9 Ablation study

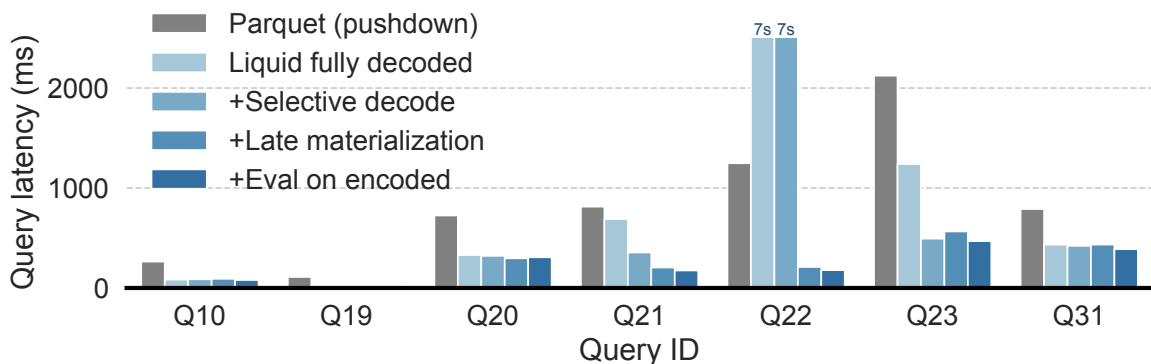


Figure 4.13: Ablation study of LiquidCache’s decoding optimizations vs Parquet (pushdown) as the baseline.

We analyze how each decoding optimization described in Section 4.3.3 contributes to overall performance. Using the same setup, we selectively enable optimizations one at a time to measure their impact.

“Liquid fully decoded” decodes entire arrays into Arrow format even when accessing single elements - a coarser granularity than Parquet’s page-level decoding (Figure 4.13). It outperforms Parquet (pushdown) on single-filter queries like Q20, where full-column

decoding is necessary because Liquid data decodes faster than Parquet (Figure 4.10). Selective decoding provides significant benefits when constructing output data using filter evaluation masks to decode only relevant data. This is particularly effective for queries with selective filters (e.g., Q21, Q23). However, even for selective queries like Q22, it sees limited benefit as they are bottlenecked by the filter evaluation phase, highlighting the importance of late filter materialization.

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

Table 4.2: Ablation study based on a variation of Q22. Eval on encoded data shows more than 98% improvement over baseline Parquet (pushdown).

Filtering directly on encoded data requires filter expressions to be compatible with the encoding scheme (e.g., equality filters on strings). While this optimization shows modest benefits in our evaluation, we demonstrate its potential impact through an additional experiment on a variation of Q22 with compatible predicates on large columns. As shown in Table 4.2, this achieves an 11× improvement over late filter materialization alone, suggesting significant performance gains when applying this technique to larger columns.

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

Table 4.3: LiquidCache latency following different operations. Operations are executed in the order of the table. The bolded filters can not be evaluated on encoded data and must be decoded first.

4.4.10 Cache dynamics

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

Table 4.3 demonstrates LiquidCache’s cache behavior across different operations. Operation 1 represents a cache miss requiring SSD data loading and Liquid transcoding. Operation 2 hits cache despite different filters, enabling encoded data evaluation without decoding. Operation 3 introduces a new filter type requiring prior decoding. After disk flush, operation 5 loads data in Liquid format, allowing encoded evaluation. Operation 6 loads a new column with latency similar to operation 1. Operation 10 loads and transcodes both columns A and B following their deletion.

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

4.4.11 Cases for non-pushdown

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

4.5 Related work

A consolidated discussion of related work across all three contributions of this dissertation appears in Chapter 8. Threads especially relevant to LiquidCache include *pushdown and semantic caching* (Section 8.2), where LiquidCache differs from FlexPushdownDB and Fusion by attacking the decoding cost that makes dynamic placement necessary; *data representation and encoding* (Section 8.3), where LiquidCache adopts modern encodings inside the cache without asking the ecosystem to replace Parquet; and *resource disaggregation* (Section 8.4), where LiquidCache achieves logical disaggregation in software rather than the physical disaggregation requiring specialized RDMA or CXL hardware.

4.6 Future work

LiquidCache addresses one of three opacity axes. Some of the directions it leaves open are taken up by the next two chapters; others remain genuinely open.

The most pressing limitation is *cache memory management*. LiquidCache uses simple LRU at the granularity of a transcoded Liquid array, but our profiling shows that most cached bytes – up to 98% under Belady-optimal replacement [21] – are never read by the queries that warmed them. This is a *content-axis* problem rather than a representation-axis one, and the next chapter (Chapter 5) attacks it directly with the SqueezeCache eviction and partial-retention mechanism. A related limitation is *disk-tier I/O efficiency*: LiquidCache’s overflow-to-disk path relies on conventional file-per-batch storage and the kernel page cache, both of which become bottlenecks under modern NVMe. Chapter 6 addresses this with a granularity-aware storage engine that decouples in-memory caching from on-disk pages.

Three further directions remain genuinely open:

- **Substrait query plans.** LiquidCache currently relies on DataFusion’s internal plan format. Supporting Substrait [160] would allow Spark and DuckDB to share the same cache.
- **Joins and sideways information passing.** Beyond projection, filtering, and simple aggregation, future versions could push joins or sideways information passing into the cache to reduce traffic further.
- **Adaptive pushdown.** Pushdown can increase network traffic for non-selective predicates (Section 4.4.11); a cost-based policy could choose at runtime whether to push, using cardinality estimates from the optimizer.

4.7 Conclusion

This chapter presented LiquidCache, the first contribution of this dissertation and its attack on the representation axis of White-Box caching. The key observation is that data decoding – not filter evaluation – is the bottleneck in pushdown systems on Parquet; the response is to transparently transcode Parquet into Liquid, an in-cache format whose cascading encodings let the cache evaluate predicates against whichever layer is most efficient. Because Liquid lives only inside the cache, the rest of the ecosystem continues to see Parquet, and

the cache is free to evolve its encodings without coordinating with anyone. Evaluation on ClickBench and TPC-H showed up to 10× lower cache CPU than Parquet pushdown and two orders of magnitude lower network traffic than file-serving caches, while keeping memory consumption comparable to compressed Parquet – exhausting much of the value the representation axis alone can provide.

The next chapter takes up where this one stops, attacking the second opacity axis: *content*. Even with LiquidCache’s adaptive encoding, the cache still holds every byte of every cached column, when queries often need only a fraction of those bytes. SqueezeCache (Chapter 5) breaks the binary keep-or-evict decision into a spectrum of partial-retention states and uses lineage analysis to decide which fragments of each column to keep.

Chapter 5

SqueezeCache: Curating Cache Content Through Lineage

This chapter attacks the second of the three opacity axes from the introduction: *content*. The previous chapter showed that a cache that takes control of its own *representation* can decode and filter cached bytes an order of magnitude faster than a conventional Parquet-on-disk cache. What that work did not change is the cache’s *contents*: LiquidCache (Chapter 4) still stores every byte of every cached column, in the encoded form that some future predicate might require. For workloads dominated by selective filters, that conservative choice leaves enormous capacity on the table – most of the bytes a cache holds are never read by the queries that brought them into memory.

Our profiling of pushdown caches on real-world workloads makes the gap concrete. Across ClickBench queries on a LiquidCache deployment, *up to 98% of cached bytes are never accessed* – even under Belady’s [21] optimal replacement policy, which gives the cache perfect foresight about which entries to evict. The problem is not *which* entries the cache holds but *how much* of each entry it holds. Figure 5.1 illustrates this with ClickBench Q25: the CPU consumes only 0.03 GB of information to answer the query, while the OS page cache holds 1.7 GB of raw data, and even an application-managed compressed cache holds 0.5 GB. The cache is operating at the granularity of whole columns, when the query operates at the granularity of fragments – a string prefix, a fingerprint, a single component of a datetime, one path through a JSON value.

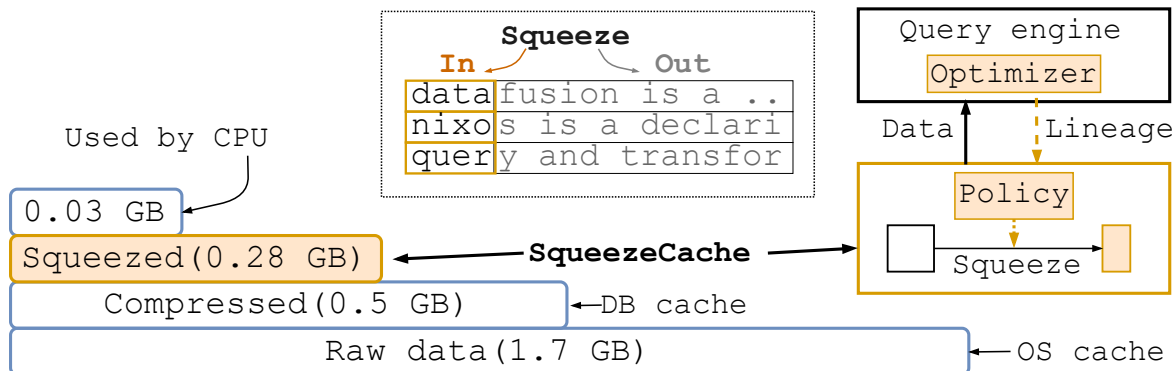


Figure 5.1: SqueezeCache caches data by how it is used, not just how frequently it is accessed, closing the gap between cached data size and utilized data size. Numbers shown for ClickBench Q25.

The natural response – pre-shred each column into smaller sub-columns at write time – has two well-known drawbacks. It requires a full-stack rewrite from file format through memory layout to query operator, slowing adoption to the same pace that has hobbled the Parquet-replacement formats discussed in Chapter 4. And it fixes the shredding at write time, well before the optimizer knows what fragments any future query will need. A datetime pre-shredded into year, month, and day costs *more* memory than the original when the query asks for day-of-week; a JSON shredded for one field is wasted memory when the next query wants another.

SqueezeCache takes a different approach: it moves the layout transformation into the cache layer, where it can be both dynamic and workload-aware. The cache *squeezes* each wide column at runtime into a compact subset of the original values – a string prefix, a 32-bit fingerprint, an extracted day-of-week, a shredded JSON sub-field – small enough to fit in memory but rich enough to answer the predicates the workload is actually running. Squeezing is lossy by design: when a future query needs information the squeezed entry no longer retains, SqueezeCache reads the missing bytes from disk and re-hydrates the column. Both drawbacks of pre-shredding disappear: the on-disk file format is untouched, and the squeeze decision is made when the query plan is known.

Squeezing requires a holistic design across the system to unlock. We summarize three challenges. First, the cached representation must be *squeezable* for each datatype: given the operator pushed down, the format must transform the data to save memory while still answering the predicate. Second, the squeezed data must be *evaluable*: the predicate must run on the partial representation, falling back to full data only when essential. Third, the

cache must decide *how much* to squeeze each entry – a richer choice than the binary keep-or-evict decision of conventional cache policy.

SqueezeCache addresses these three challenges with three corresponding mechanisms. *Squeezable Data Layouts* (Section 5.3) for common datatypes (strings, integers, decimals, floats, datetimes, JSON variants) structurally decompose each datatype to allow partial retention. *Lineage Pushdown* (Section 5.4) extracts the per-column usage pattern from the query plan and communicates it to the cache through schema metadata, without coupling the two systems. A *Multi-State Squeeze Policy* (Section 5.5) treats data fidelity as a spectrum, transitioning each cached entry through uncompressed, compressed, squeezed, and evicted states based on memory pressure and observed access patterns.

We implemented SqueezeCache as an extension to LiquidCache and evaluated it on ClickBench and JSONBench. SqueezeCache improves cache hit ratios by up to 2× over a compression-only baseline (4× over an uncompressed cache), reduces end-to-end query latency by up to 2× (22× over uncompressed), and cuts I/O by up to 8× (12× over uncompressed) in memory-constrained environments. SqueezeCache is now the default eviction mechanism in the open-source LiquidCache project.

The contributions of this chapter are:

1. Squeezable data layouts for strings, integers, decimals, floats, datetimes, and JSON variants that admit predicate evaluation on partial data (Section 5.3).
2. Lineage pushdown, an optimizer-to-cache communication mechanism that extracts the per-column squeeze target from the query plan (Section 5.4).
3. A multi-state squeeze policy that generalizes eviction from a binary decision to a fidelity spectrum (Section 5.5).
4. An empirical evaluation on ClickBench and JSONBench that quantifies the contribution of each mechanism (Section 5.6).

The chapter is organized as follows. We first give the SqueezeCache-specific background – the nature of string workloads that motivate the content-axis framing and the relationship to semantic caches (Section 5.1) – and then step through the SqueezeCache mechanisms (overview in Section 5.2, then squeezable layouts, lineage pushdown, and squeeze policy), the evaluation (Section 5.6), related work, and a transition to the next chapter.

5.1 Background

Chapter 2 introduced the foundations SqueezeCache builds on: data lives in Parquet on object storage; caches sit between compute and storage in a disaggregated topology (Figure 2.2); queries flow through scan, decode, and execute stages (Figure 2.1); and conventional eviction policies – LRU-K [131], ARC [122], 2Q [82], S3-FIFO [177], SIEVE [189], GL-Cache [176], Baleen [174] – all treat their cached units as opaque, undifferentiated blocks. Chapter 3 developed the design vocabulary – the reference architecture, the information-flow channels (schema metadata, pushdown, access traces), and the fidelity spectrum – on which SqueezeCache instantiates the content axis. Chapter 4 added the immediate host environment: LiquidCache, a disaggregated pushdown cache that transcodes Parquet into a query-friendly format and exposes the lineage-aware hooks SqueezeCache uses to intercept eviction and inspect column usage. This section adds the two pieces of background none of those chapters cover: the byte-level properties of real-world strings – the data type most responsible for the content-axis gap – and the relationship to the family of *semantic* caching mechanisms that also try to cache less than full data.

5.1.1 The nature of strings in real-world data

The data type most responsible for the content-axis gap is strings. Real-world analytical workloads are saturated with them – URLs, search phrases, page titles, JSON payloads, user identifiers, log lines. ClickBench, drawn from production web analytics [35], contains roughly 40 string columns out of 105 total, and the largest single column (Title) accounts for over half of the dataset’s on-disk size [138]. JSONBench [37], a more recent benchmark over public-Bluesky social-network events, takes this further: every record is a deeply nested JSON value where a typical query references only one or two paths. These workloads expose three properties of strings that conventional columnar caching machinery handles awkwardly.

Strings are variable-length, and often long. Unlike fixed-width numeric columns, strings carry the cost of length-prefix metadata, offset tables, and inline-versus-indirect storage decisions. In ClickBench, the URL and Title columns have median lengths in the dozens of bytes but maximums in the kilobytes; the long tail forces caches to provision for worst-case storage even when the typical entry is small.

Most string operations need only a fragment of each value. Empirically, equality and prefix predicates dominate string filter workloads, followed by length checks and substring (LIKE) matches. Equality and prefix can be decided from the first few bytes; length needs only metadata; substring can be quickly *pruned* by a small fingerprint [157] that summarizes which bytes appear in the string. Even when the predicate semantically requires the full string, the fraction of strings that survive prior filters and require full materialization is typically small – a few percent of the column.

The byte-level layouts the industry has converged on already exploit this. Umbra-style and Arrow StringView layouts inline a short prefix in a fixed-size header so that the most common predicates – equality on short keys, length comparisons – can be decided without dereferencing the variable-length buffer. FSST [25] trains a byte-level symbol table per column so that decompressing a single value is cheap, allowing the variable-length buffer to be held compressed. SqueezeCache builds on both: under memory pressure, it retains only the StringView header and a small substring-fingerprint bit array, dropping the FSST-compressed buffer to disk entirely and reading it back only when a predicate requires full materialization.

These three properties explain why the cache-utilization gap is widest on strings, and why the squeezable string layout introduced in Section 5.3.1 is the most aggressive layout in the chapter. The same principles – *decompose so that common predicates can be answered from a fragment* – carry over to integers (where most filters need only a few bits of precision), datetimes (where most queries extract a single component), and JSON variants (where most queries reference a handful of paths). The rest of the chapter generalizes the string analysis to these other types.

5.1.2 Relationship to semantic-cache mechanisms

Semantic caches exploit query structure to cache fragments rather than full data. *Result caches* store the literal output of past queries and serve only exact repeats. *Materialized views* store the output of named sub-queries; they require user or system declaration and are limited to the views the operator chose to maintain. *Predicate caches* [142] record which row ranges have satisfied past predicates, accelerating subsequent scans with the same predicate. *Time-series-specific caches* such as STsCache [87, 105] and TSCache [112] push the idea further by exploiting properties unique to time-series workloads.

SqueezeCache differs from all of these along two axes. First, it works at the granularity of *cached column data* rather than cached results, so a squeezed entry is reusable across queries that share the column but not the predicate – the common case in analytical workloads. Second, it treats lineage as a *continuous* variable rather than a binary one: instead of choosing between “cache the result” and “cache nothing”, a squeeze policy picks how much of each column to retain along the fidelity spectrum, with the choice driven by the lineage analysis in Section 5.4. Predicate caching can be viewed as a degenerate special case of squeezing where the squeezed representation is a single bitmask; the framework in this chapter generalizes this to lossy data formats for each datatype.

5.2 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:

```
“sql
SELECT EXTRACT(DOW FROM Date) AS day_of_week, COUNT(*)
FROM table WHERE Title LIKE "
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 5.2.

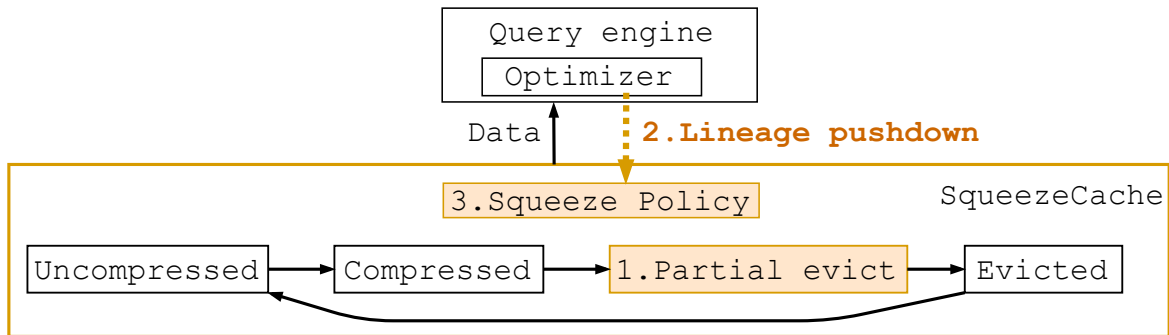


Figure 5.2: Overview of SqueezeCache. It has three core components: squeezable data layouts, lineage pushdown, and squeeze policy.

1. Squeezable Data Layouts (Section 5.3).

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 5.4).

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 5.5).

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.

5.3 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 [72], BtrBlocks [91], 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.

5.3.1 Squeezing Strings

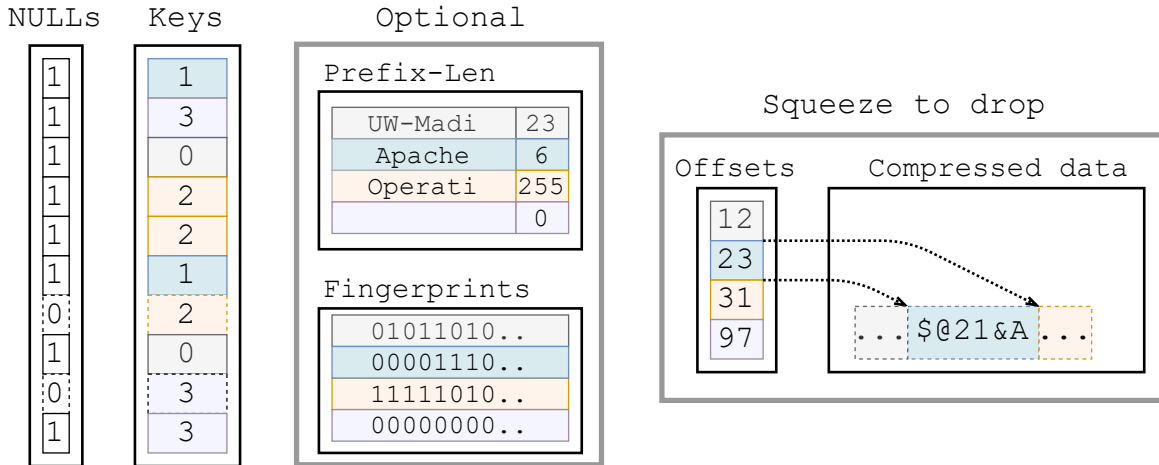


Figure 5.3: Overview of string layout. The byte-view of the string is similar to previous work, where each string has a header (which encodes string length, offset to the buffer, and a prefix), and a string buffer which contains the actual string data.

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 5.3 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 [25]. We store an additional offset array (4 bytes per unique string) to help locate the strings we need to decompress (selective decompression).

5.3.1.1 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.

5.3.1.2 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.

5.3.1.3 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.

5.3.1.4 String fingerprint.

String fingerprints [157] 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.

5.3.1.5 Optional prefix and fingerprint.

Both prefix and fingerprint have storage costs. If lineage analysis (Section 5.4.1) 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.

5.3.2 Squeezing Integers

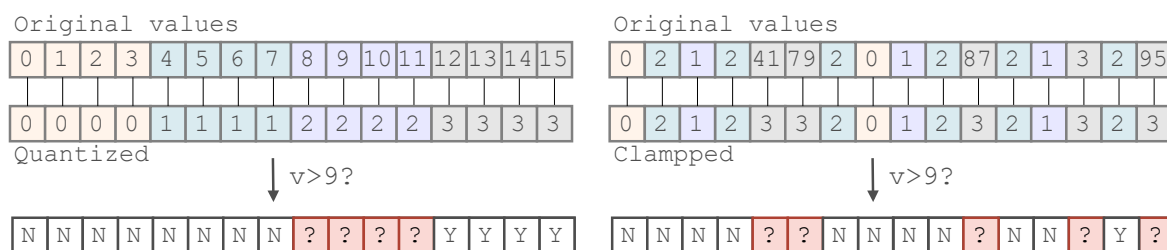


Figure 5.4: Overview of squeezable integer. Integers can be squeezed by quantizing (left) or clamping (right) to a fixed range. Squeezed integers are smaller but can still answer many filters.

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 5.4 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 5.4, 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 5.4). 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.

5.3.3 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 [72].

5.3.4 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 5.3.2. Since the percentage of exceptional values is expected to be small, our implementation doesn't apply squeezing to these values.

5.3.5 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 5.5 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 5.4.1).

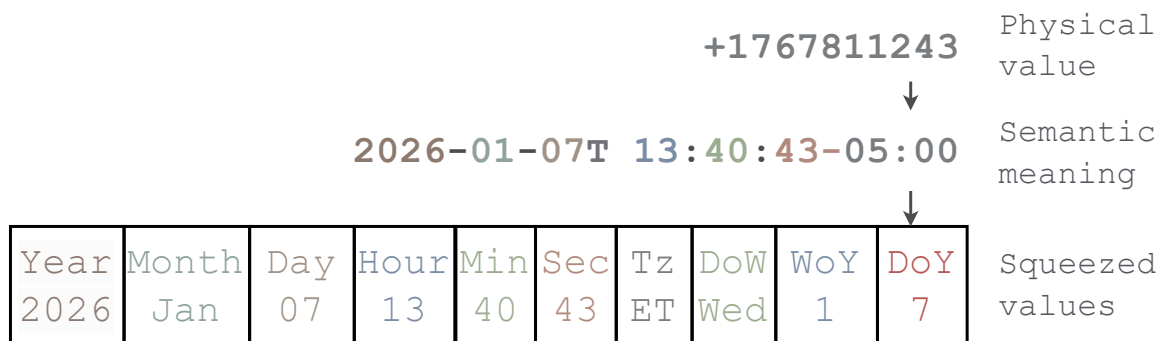


Figure 5.5: Datetime squeezing. Although physically stored as a single integer, datetime can be squeezed with different semantic components.

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).

5.3.6 Squeezing Variants (JSON)

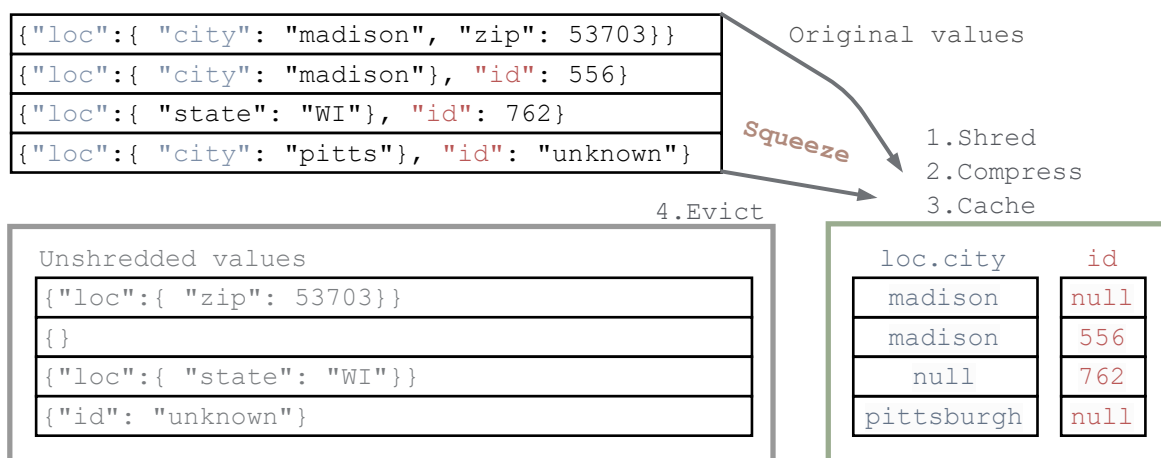


Figure 5.6: Squeeze a variant column into a set of shredded columns. We can shred the variant column into a set of shredded columns, and only cache the shredded columns in memory. When a variant get operation is called, we can reconstruct the variant object from the shredded columns.

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 5.6, 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.

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.

5.4 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.

5.4.1 Lineage analysis

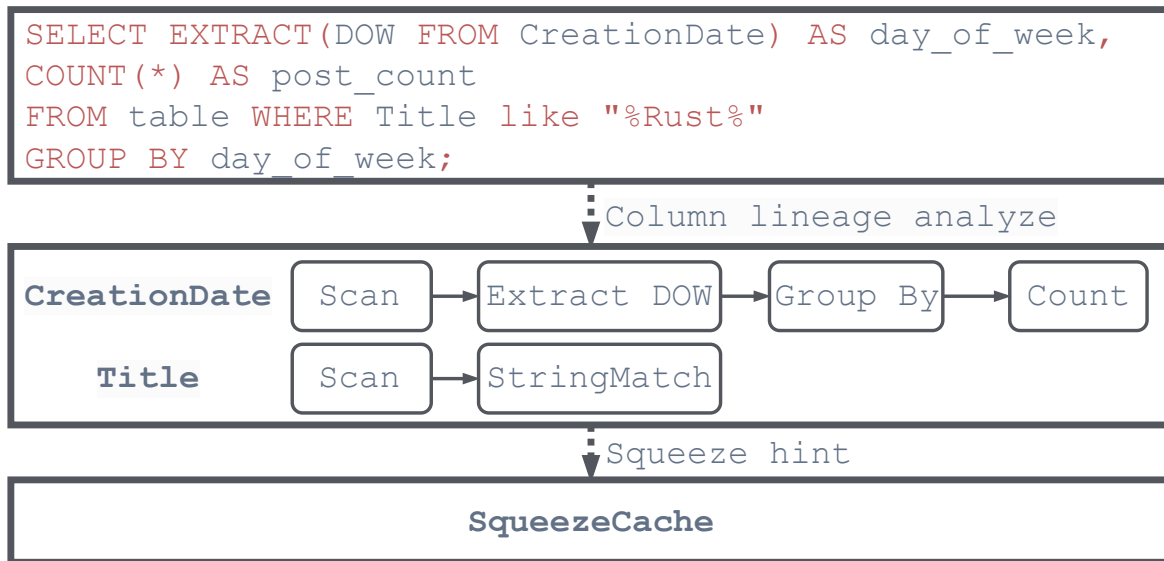


Figure 5.7: Column lineage example. SqueezeCache analyzes the query to extract the lineage of each column. For *CreationDate*, the lineage reveals that only the day-of-week component is needed. For *Title*, the lineage indicates a substring match, suggesting a fingerprint is sufficient. This information guides the cache to squeeze the data into compact, query-specific representations.

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.

As shown in Figure 5.7, 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.

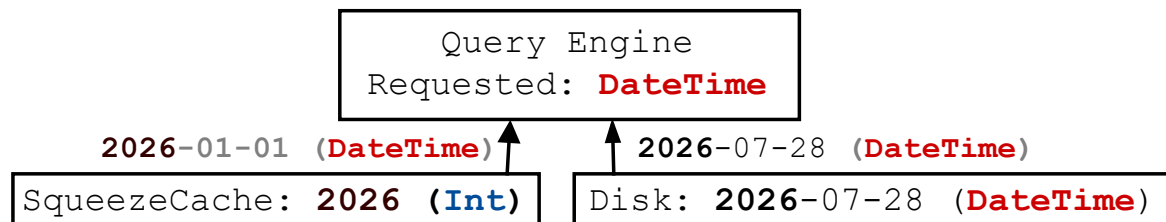


Figure 5.8: SqueezeCache creates a synthetic value when the cached data type does not match the expected type. The lineage analysis ensures that the unused components are not needed by the query.

5.4.2 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 5.8, 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.

5.4.3 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.

5.5 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 5.9 shows an array’s life cycle in SqueezeCache. Arrays transition through four states: uncompressed, compressed, squeezed, and evicted. Inserted as uncompressed, they are compressed, then squeezed, and finally evicted as the cache fills. Specifically, squeeze targets the uncompressed queue, then compressed, then squeezed.

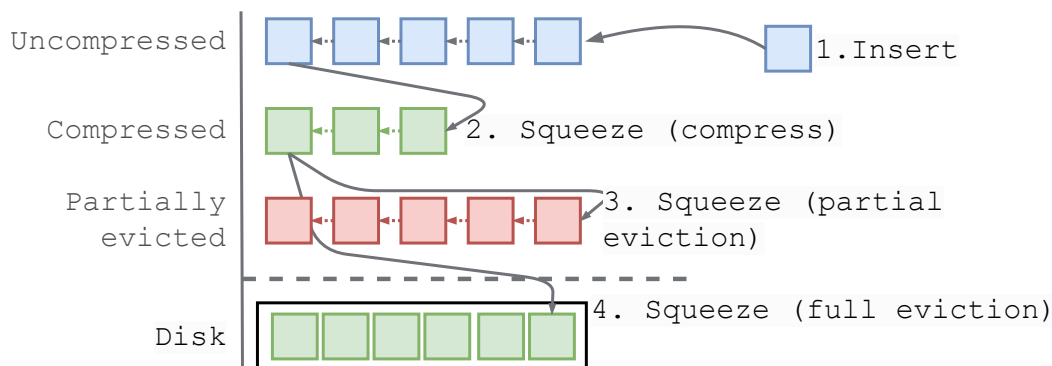


Figure 5.9: Overview of the life of an array in SqueezeCache. Arrays are inserted as uncompressed data. When the cache fills, squeeze operations progressively transition entries: uncompressed data is compressed, compressed data is squeezed, and finally, squeezed data is evicted from the cache.

5.5.1 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 squeezed (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.

5.5.2 Selection Policy

When the system is under memory pressure, it must pick a cached entry to squeeze, similar to an eviction policy. Figure 5.9 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), which prevents large scans from polluting the cache by immediately evicting recently inserted data.

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 squeezed entry, we also write the full data to disk and record its location. This allows later hydration (Section 5.5.4) to read it back. By default, SqueezeCache writes the compressed data (LiquidArray [72]) to disk, because it is highly compressed, and requires no deserialization on read.

5.5.3 Retention Policy

Given lineage information (Section 5.4.1), 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.

5.5.4 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 squeezed 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.

5.5.5 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 [137, 162, 168].

- 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.

5.6 Evaluation

Our evaluation is organized around two groups of questions: two on whether squeezing pays off in aggregate, and three on why it works and whether one mechanism generalizes. The first group asks whether squeezing pays off on realistic workloads:

1. **Does SqueezeCache improve end-to-end performance?** On ClickBench’s filter-heavy queries it cuts query latency by up to 22× and I/O by up to 12× versus an uncompressed cache, with no slowdown on the queries that do not benefit (Section 5.6.3).
2. **How does cache efficiency scale with the memory budget, and how close is it to optimal?** Across budgets, SqueezeCache sustains up to 2× higher cache hit ratios than a compression-only baseline (4× over uncompressed); we also quantify the remaining gap to a row-granular optimum (Section 5.6.4).

The second group asks *why* squeezing works and whether one mechanism generalizes across query patterns. We isolate a single workload per experiment so that the squeeze strategy lineage analysis selects, and its effect, are visible in isolation:

1. **Do retained prefixes resolve selective equality and range predicates without disk I/O?** On ClickBench Q25, squeezed prefixes answer all but 2.1% of squeezed batches in memory, leaving only a handful of genuinely ambiguous rows to fetch from disk (Section 5.6.5).

2. **Does squeezing adapt when prefixes do not work, such as substring search?** For LIKE queries, lineage analysis switches the squeeze target to string fingerprints, halving decode time at essentially unchanged I/O (Section 5.6.6).
3. **Does squeeze generalize beyond strings to sub-field extraction?** For JSON fields, datetime components, and struct sub-columns, SqueezeCache caches only the referenced fields, shrinking entries by up to 46× and serving the query entirely from memory (Section 5.6.7).

We first describe the implementation (Section 5.6.1) and baselines (Section 5.6.2), then take each question in turn.

5.6.1 Implementation

We implemented SqueezeCache on top of LiquidCache [72] 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.

5.6.2 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 [163]. LiquidCache: LiquidCache [72] compresses data but does not perform partial eviction. This baseline represents the state-of-the-art compression-only cache.

5.6.2.1 Non-baselines

Eviction policies like LRU-K [131], SIEVE [189], and S3-FIFO [177] 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.

5.6.3 End-to-end Evaluation

We measure end-to-end gains on two benchmark families that probe opposite ends of squeezing’s applicability: ClickBench, whose scan-intensive, low-latency queries over real-world data are exactly the workload SqueezeCache targets; and the synthetic TPC-H and TPC-DS suites, where we expect little benefit and instead check that squeezing falls back gracefully to compression without introducing overhead.

5.6.3.1 ClickBench.

ClickBench [35] 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 5.10 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 14× vs. Arrow in this experiment. 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 5.6.6), 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).

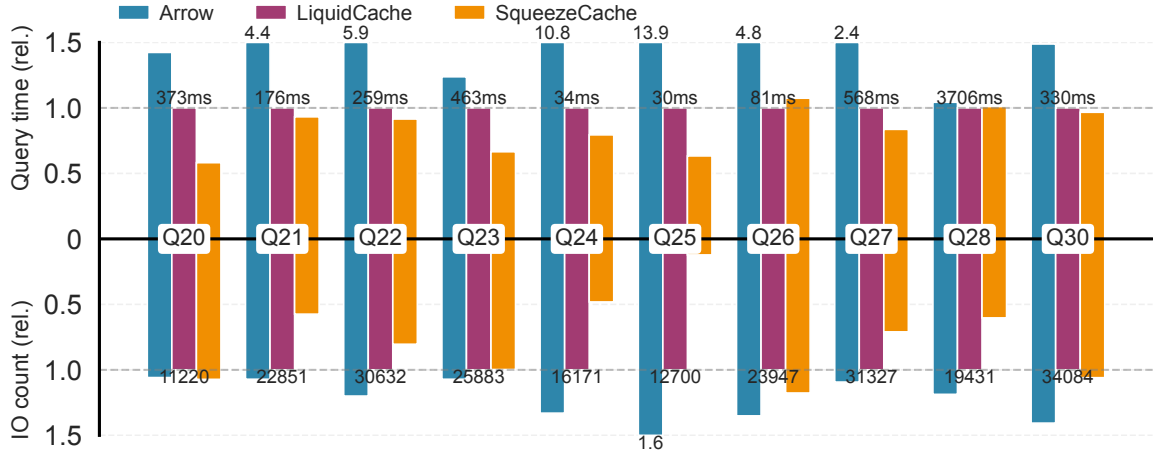


Figure 5.10: ClickBench overall performance comparison. Query time and I/O count comparison between SqueezeCache and the baselines. Values are normalized to the LiquidCache baseline. The I/O device is kernel page cache, so end-to-end query time mainly reflects in-memory processing time.

5.6.3.2 TPC-H and TPC-DS.

As discussed in Section 5.5.5, 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 [72], and multiple studies show it does not reflect real-world workloads [137, 162, 168]. 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.

5.6.4 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 5.11 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).

5.6.4.1 Overall results.

SqueezeCache consistently outperforms LiquidCache by $2\times$ in terms of cache hit ratio, and $4\times$ 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.

5.6.4.2 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— $60\times$ 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.

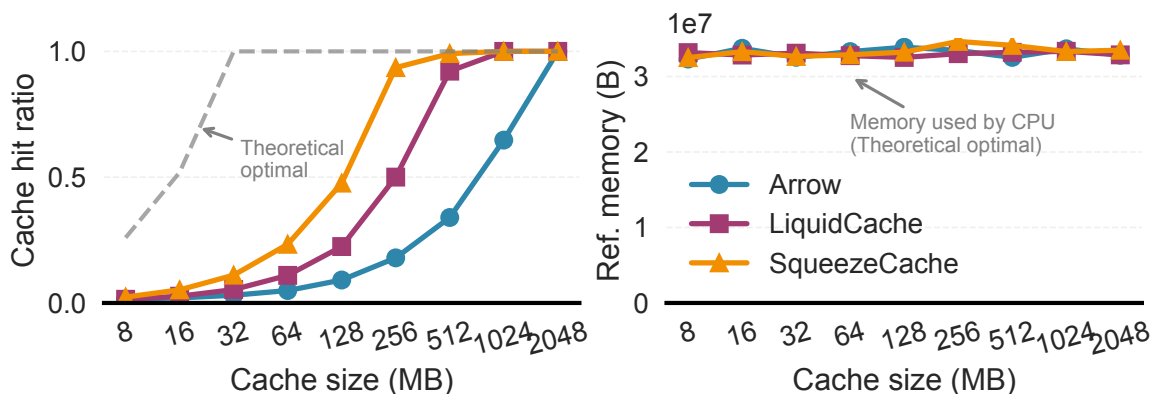


Figure 5.11: Cache size vs cache hit ratio (left) and actual referenced memory (right). Cache hit ratio is the number of batches cached in memory divided by the total number of batches. Actual referenced memory is LLC (last level cache) references cacheline size. It approximates total memory traffic from memory to CPU.

5.6.4.3 How does the cache composition look like?

In SqueezeCache, cached entries are not fixed: they can be Arrow, Compressed, or Squeezed. Figure 5.12 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.

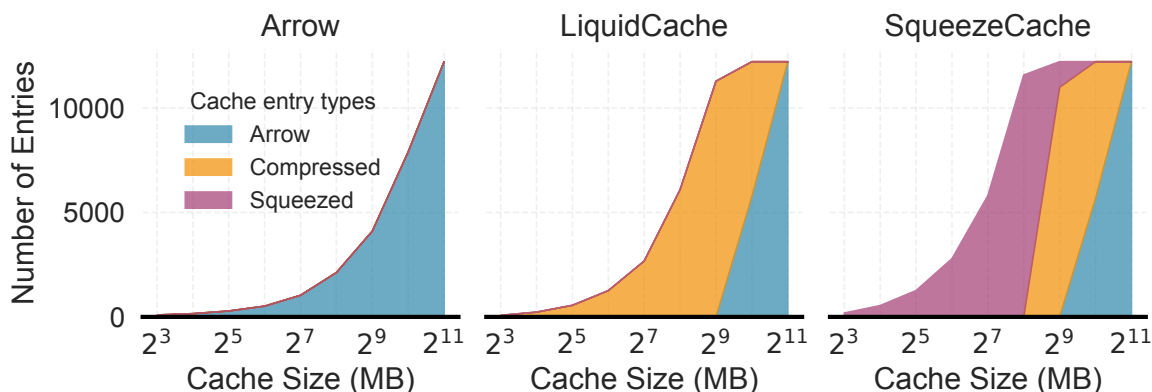


Figure 5.12: Cache entry composition comparison. Number of different types of entries in the cache. Arrow has only one type of entry, LiquidCache additionally has compressed entries, and SqueezeCache additionally has squeezed entries.

5.6.4.4 I/O is not the only cost

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 [72], the in-memory data processing time is also a significant portion of the query time. Figure 5.13 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.

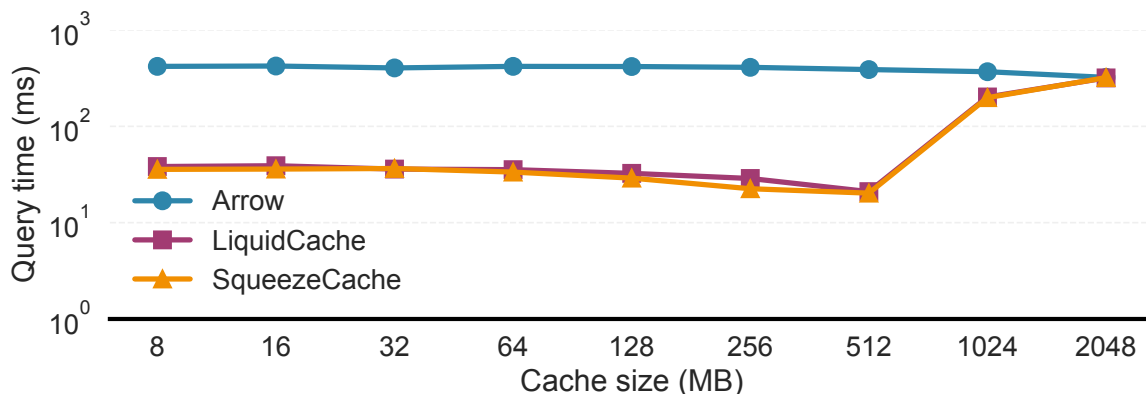


Figure 5.13: Query time comparison with kernel page cache as the I/O backend. Query time comparison between SqueezeCache and the baselines, with kernel page cache as the I/O device. Kernel page cache is a transparent cache mechanism provided by Linux kernel, a cache hit only costs a few microseconds.

5.6.5 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.

```

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

```

5.6.5.1 Is it a good fit for SqueezeCache?

The predicate `SearchPhrase <> "` works well with squeezable strings (Section 5.3.1): the prefix alone often decides the predicate. Although selectivity is only 13.1%, DataFusion also adds a dynamic filter [19] 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.

5.6.5.2 Cache composition

The on-disk data for this query is about 565MB; we set cache size to 256MB. Table 5.1 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 [72]), Squeezed (using techniques from this chapter), or Evicted (on disk). Entries start as Arrow and are progressively squeezed as memory pressure increases (Section 5.5.1).

	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 5.1: 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.

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.

5.6.5.3 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 5.2 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.

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

Table 5.2: 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.

Figure 5.14 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.

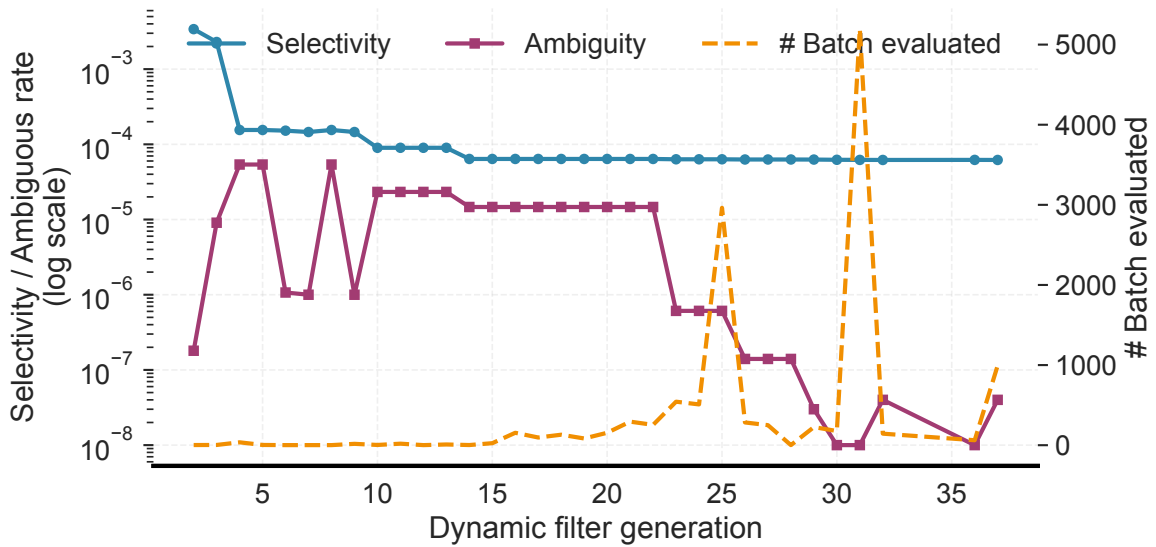


Figure 5.14: Dynamic filter generations, with their selectivity and ambiguous rate. Selectivity is the percentage of rows that pass the filter. Ambiguity shows the percentage of rows that can not be pruned with squeezed data. As query progresses, more selective filters are generated and ambiguity drops.

As shown in Figure 5.14, 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 5.15 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.

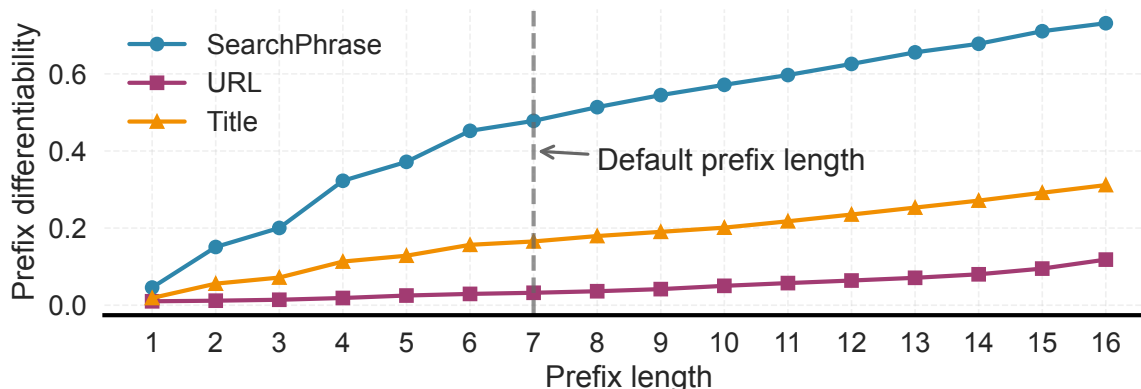


Figure 5.15: Prefix length sensitivity analysis. Differentiability is computed as the number of unique values captured by the prefix, divided by the total number of unique values in the batch.

5.6.6 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'

```

"sql
SELECT COUNT(*) FROM hits WHERE "URL" LIKE '

```

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 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 5.3. The I/O counts are similar (fingerprints do not reduce I/O), but SqueezeCache requires slightly more IO due to fingerprint overhead. When reading from fast NVMe drives, decoding often dominates query time [72] 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 of decode time), roughly halving overall query time (a 50% reduction). 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 [157], which we plan to pursue in future work, but even the simple round-robin assignment is effective.

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

Table 5.3: 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.

5.6.6.1 Fingerprint overhead analysis

Figure 5.16 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.

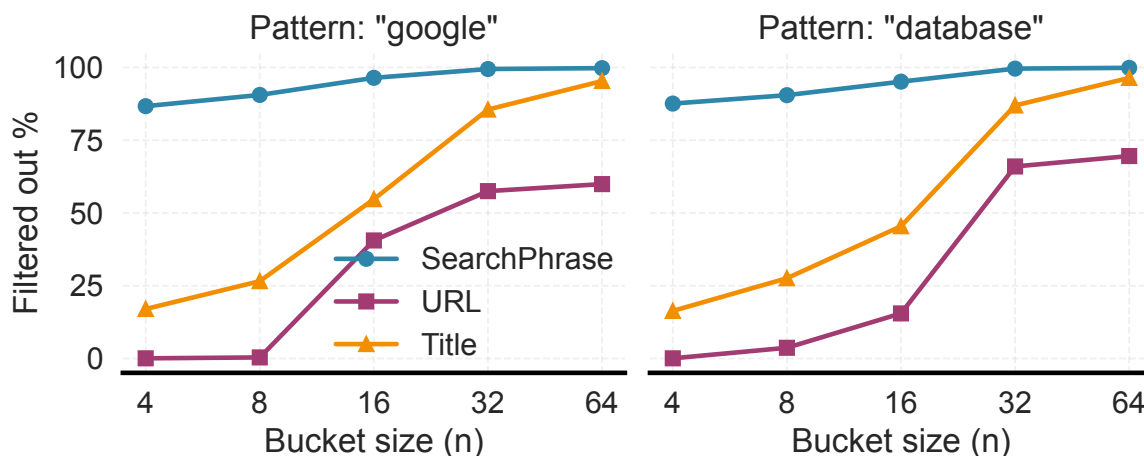


Figure 5.16: Fingerprint bucket size vs filter effectiveness. The bucket size is the number of buckets used to store the fingerprint.

5.6.7 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 [53]. 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 [69]. All data is pre-shredded to isolate caching effects from runtime extraction overhead. We analyze JSONBench Q1 [37], which counts the number of events in the “bluesky” table, grouped by the “event” field.

```
“sql
```

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

```
”
```

Table 5.4 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 [72], yielding 0.09 MB per batch—a $46\times$ reduction in size.

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

Table 5.4: 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.

Table 5.5 shows runtime performance: the baseline caches only 2.5% of data, while SqueezeCache caches 100% in memory with 0 I/O, yielding $3.2\times$ lower query latency. Notably, SqueezeCache generalizes JSON-specific optimizations—it supports efficient semi-structured queries automatically without special handling.

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

Table 5.5: JSONBench 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.

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 roughly 12GB after compression, again, we set the memory budget to roughly half of the actual memory usage. Table 5.6 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 495 batches in memory, with no IO required.

```
“sql
SELECT EXTRACT(DOW FROM "CreationDate") AS day_of_week,
```

```

COUNT(*) AS post_count
FROM "Posts" GROUP BY day_of_week;
"

```

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

Table 5.6: Cache composition comparison. Number of cached batches, each batch consists 8192 rows.

Table 5.7 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 $\times$, 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 $\times$, 3.0 $\times$, 2.4 $\times$, and 3.9 $\times$, respectively.

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

Table 5.7: 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).

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. The mechanism and the resulting savings mirror the datetime-component case above: each unused sub-column is squeezed away, and memory usage scales down in proportion to the fraction of the struct the workload actually touches. 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.

5.7 Related work

A consolidated discussion of related work across all three contributions of this dissertation appears in Chapter 8. Threads especially relevant to SqueezeCache include *caching architectures and policies* (Section 8.1), where SqueezeCache is orthogonal to eviction-policy design and benefits any choice; *pushdown and semantic caching* (Section 8.2), where SqueezeCache generalizes predicate caching by treating fidelity as a spectrum rather than a binary; and *data representation and encoding* (Section 8.3), where SqueezeCache extends compression-aware execution from lossless to lossy regimes and uses lightweight summaries such as string fingerprints as squeeze targets.

5.8 Conclusion

This chapter presented SqueezeCache, the second contribution of this dissertation and its attack on the content axis of White-Box caching. The starting observation is that, even under Belady-optimal replacement, most cached bytes – up to 98% on real-world analytical workloads – are never read by the queries that warmed them. The response is *squeezing*: replacing each cached entry with a compact, lossy subset that still answers the predicates the workload is running, decided per column by lineage analysis and managed through a multi-state policy that treats fidelity as a spectrum rather than a binary. Evaluation on ClickBench and JSONBench showed up to 4× higher cache hit ratios and up to 22× lower query latency than an uncompressed cache (2× on both over a compression-only baseline) at the same memory budget, with no end-user changes required to existing analytical engines. SqueezeCache is the default eviction mechanism of the open-source LiquidCache project.

The third opacity axis – *granularity* – remains. LiquidCache and SqueezeCache both inherit the cache unit they manage from the query engine above (typically an 8192-row batch of a single column); neither changes the relationship between cache pages and disk pages, or between the cache and the records inside it. On disk, both rely on a conventional file-per-batch layout backed by the kernel page cache – a model that breaks down on modern NVMe SSDs whose bandwidth and IOPS approach DRAM territory (Table 2.1). The next chapter (Chapter 6) takes up this third axis, building a storage engine whose

in-memory cache pages are no longer mirrors of on-disk pages and whose I/O path is structured around `io_uring` rather than the OS page cache.

Chapter 6

Bf-Tree:

Decoupling Cache Granularity from Disk

This chapter attacks the third and final opacity axis from the introduction: *granularity*. The previous two chapters showed how a cache can take control of its in-memory *representation* (Chapter 4) and the *content* it retains within each entry (Chapter 5). What both chapters left untouched is the relationship between the cache’s in-memory unit and the on-disk unit it spills to: an 8192-row Liquid array in memory becomes an 8192-row file on disk; a squeezed batch in memory becomes a squeezed file on disk. The cache’s *granularity*, in other words, remains coupled to whatever the disk produces – a coupling that worked fine when disks were millisecond-class but breaks down under modern NVMe SSDs whose latency and bandwidth approach DRAM territory (Table 2.1).

Bf-Tree attacks this axis at its most foundational layer: the larger-than-memory range index itself. A conventional B-Tree organizes records into 4 KB pages aligned with disk I/O, so each disk read narrows the search space by a full page. The page-aligned design comes at a price. Updating one record requires writing the whole page back – a write-amplification factor of tens to hundreds depending on record size. Caching a page in memory because one of its records is hot also pulls in the cold records on the same page, wasting cache capacity that could hold something hotter. These costs were tolerable when disks were the binding constraint, but modern NVMe SSDs have made the trade-offs visible: an SSD that can almost saturate a memory channel does not need to be coddled with page-aligned writes, and the unused records on a cached page now occupy expensive DRAM.

Many enhancements have been proposed. LSM-Trees [33, 43, 48, 79, 96, 118] replace pages with logs to reduce write amplification, at the cost of slower reads and expensive compaction. Bw-Trees [102, 173] and Be-Trees [22, 38] chain delta records onto pages

to batch updates, but long chains slow search. Two-Tree [74, 193], Anti-Caching [45], Siberia [56], and TreeLine [182] introduce separate *record caches* on top of page caches to improve point-lookup efficiency, at the cost of dual memory management and no benefit to range scans. Each enhancement addresses one symptom of page-aligned caching without addressing the underlying coupling between cache page and disk page.

The central observation of this chapter is that the disk's page-aligned nature does not require the cache to be page-aligned too. If we let cached pages have *variable length*, the cache's memory layout no longer needs to mirror the disk's: it can hold a few hot records when that is enough, buffer recent updates when that is useful, mirror the whole page when range scans demand it, or even grow beyond the disk page to cache a key-range gap [65]. We crystallize this idea into a new data structure – the *Bf-Tree* – whose memory component is a variable-length buffer pool of *mini-pages*, each holding a judiciously chosen subset of the corresponding disk page.

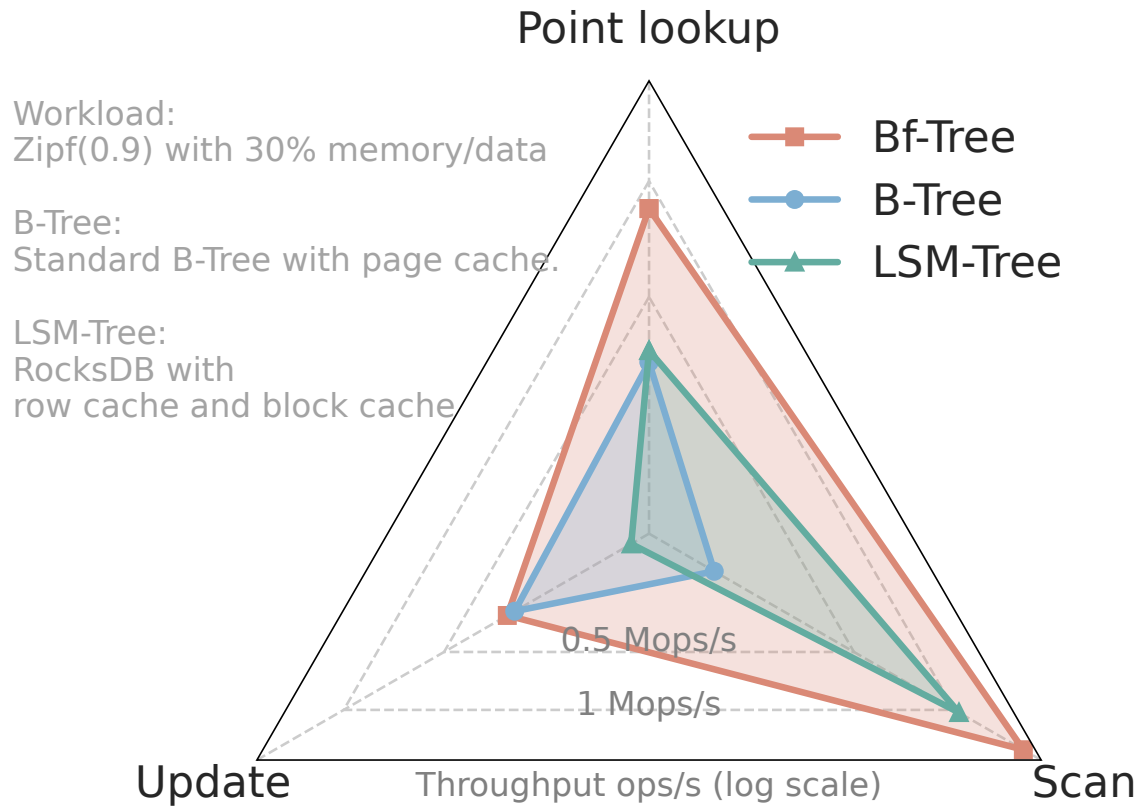


Figure 6.1: Throughput (log-scale) along three dimensions – point lookup, scan, and update. Each triangle is a different system; larger is better. Bf-Tree dominates both B-Tree and LSM-Tree along all three dimensions; Section 6.5.3 discusses this experiment in detail.

Mini-pages serve three purposes simultaneously. First, they *cache frequently accessed records* at record granularity, so the cache holds hot records without dragging along cold ones. Second, they *buffer recent updates* at variable size, batching them into a single page write when the page is evicted – recovering most of the write-amplification savings of an LSM-Tree. Third, they *cache range gaps* when range scans warrant: a mini-page can grow to full page size and act as a conventional page cache for that key range. A single mechanism therefore answers what required three different mechanisms in prior work (record caches,

delta chains, page caches), and the variable-length buffer pool allows the mix to be set *dynamically* as the workload changes.

Variable-length cache entries introduce four engineering challenges absent in fixed-size buffer pools. The buffer pool must (1) constrain total memory consumption to a configured value; (2) track which mini-pages are in use and which are free, despite the irregular sizes; (3) identify hot mini-pages worth growing and cold mini-pages worth shrinking or evicting; and (4) coordinate all of this concurrently across many threads, on hardware fast enough to make every microsecond visible. We meet these constraints with a novel circular-buffer buffer pool, inspired by FASTER’s hybrid log [31, 32] but generalized from fixed-size to variable-size allocations.

Bf-Tree is the most distant of the three contributions from this dissertation’s analytical-caching starting point. Where LiquidCache and SqueezeCache live in the lakehouse stack and serve filtered columns to analytical queries, Bf-Tree is a transactional key–value index – the kind of structure that backs an OLTP database, a metadata service, or any system that needs ordered lookups and updates at scale. The shift is deliberate: it shows that the White-Box philosophy is not specific to analytics. The same move – *looking inside the cached unit so that the cache can hold something smaller and more useful than the disk produced* – pays off in a KV store too, on a different set of operations (point lookup, range scan, update) than the analytical-pushdown regime of the earlier chapters.

We implemented Bf-Tree in 13k lines of Rust on top of the `io_uring`-based I/O substrate described in Chapter 7. On a YCSB-like benchmark with 200 million 32-byte records and 2 GB of cache, Bf-Tree is 2× faster than RocksDB [48] on range scans, 6× faster than a conventional B-Tree on writes, and 2× faster than both on point lookups – the only system among our baselines to perform well on all three operations simultaneously. In-memory performance matches state-of-the-art in-memory B-Trees [98], and tail-latency curves track the cache hit ratio (~ 75% with Bf-Tree’s mini-pages, versus ~ 50% with page-level caching at the same memory budget).

The contributions of this chapter are:

1. The mini-page abstraction, a variable-length cache entry that decouples cache granularity from disk granularity.
2. A novel variable-length buffer pool over a circular log that manages mini-pages concurrently and at scale.

3. A full implementation of Bf-Tree in Rust with strong correctness guarantees, evaluated against RocksDB, conventional B-Trees, modern B-Tree variants, and Leanstore.
4. A demonstration that the White-Box philosophy generalizes beyond analytical caches to transactional key-value indexes.

The chapter is organized as follows. We first cover the background on B-Trees, LSM-Trees, and modern NVMe SSDs that frames Bf-Tree’s design choices. The remaining sections present the architecture, the variable-length buffer pool, the core operations, the evaluation, and related work. A closing section completes the three-axis White-Box story.

6.1 Background

We start by discussing recent advancements in the design of B-Trees and LSM-Trees, the two most important data structures for larger-than-memory data systems. Although B-Trees and LSM-Trees are traditionally optimized for read and write-intensive workloads respectively, practical use cases must handle both workloads simultaneously. Therefore, they have each been extended to manage the opposing workload type efficiently.

6.1.1 B-Tree

A larger-than-memory B-Tree organizes data in pages and places hot pages in memory while cold pages reside on disk. It assigns each page with a unique ID and maintains a mapping from the page ID to the physical location of the page (on disk or in memory). This allows a B-Tree to grow larger than memory (by evicting cold pages to disk) but still allows efficient access to data on disk.

Point lookup. A B-Tree excels in efficient point lookups by traversing the tree from the root to the leaf node and then binary searching the leaf node for the target key-value pair. By caching inner nodes in memory [45], a B-Tree can complete a point query with a single IO operation involving a leaf node retrieval from disk. This design significantly enhances lookup speed for larger-than-memory data.

Despite their efficiency in disk lookups, B-Trees face challenges in caching hot records, especially when they are interspersed with cold ones on the same page. B-Tree’s page-based caching brings the entire page in memory – caching hot records along with cold ones – leading to low cache efficiency.

To mitigate this problem, multiple systems – such as Two-Tree [193], Siberia [56], Anti-caching [45], and Tree-line [182] – have incorporated record-level caching (i.e., cache individual records rather than entire pages) in their system to improve caching efficiency. While effective for point lookups, record-level caching cannot operate alone, as it does not benefit range scans, necessitating page-level caching just for range queries. Further, as a separate component, record caches are not helpful for writes. With two caching components, the system needs a delicate balance in memory allocation between the two types of caches.

Write. A write operation in a B-Tree often faces significant write amplification: a single update to a key-value pair requires a write to the entire page. This is because the record size (often less than 100 bytes [29]) is much smaller than the page size (ranging from 4KB to 64KB). A typical write operation on a B-Tree leaf page often involves reading the entire page from disk, modifying the page in memory, and writing the entire page back to disk. Such inefficiency is problematic in write-intensive workloads, prompting a preference for LSM-Trees, which are better suited to these scenarios.

To mitigate write amplification, researchers have proposed various modifications [22, 38, 102, 173] to a conventional B-Tree. Among these, the Bw-Tree’s [102] delta chain approach represents a notable advancement. A write to a leaf page is “blindly” appended to an in-memory delta chain without touching the leaf page. As the delta chains grow, they eventually merge to the leaf page in a batched manner, thereby reducing write amplification. However, long delta chains slow down search during both read and write operations, as they need to pointer-chase the delta chain to find the target key-value pair.

Range scan. B-Trees support efficient range scans by chaining the leaf nodes for efficient forward and backward scans.

Moreover, a B-Tree’s page-based caching is ideal for range scans, as the entire page is cached in memory, preserving spatial localities of neighboring records.

6.1.2 LSM-Tree

The Log-Structured Merge-Tree (LSM-Tree [48, 64, 132]) complements the B-Tree, especially for write-intensive workloads. LSM-Trees allow efficient data ingestion but incur higher costs for the read workloads. Despite this trade-off, LSM-Trees are widely used in industry, e.g., RocksDB [48], TiDB [79], and Bigtable [33].

Write. LSM-Trees append incoming writes to an in-memory buffer and periodically flush the buffer to disk. When writing to disk, records are sorted and merged with existing data, a process known as compaction. LSM-Trees employ a multi-level file organization to manage compaction efficiently.

Despite the optimizations, compaction remains resource-intensive, often leading to IO amplification and significant read latencies at the higher percentiles. Recent works have proposed to mitigate compaction overheads by offloading it from the critical path [191] and reducing data writes [118].

Point lookup. During a point lookup, an LSM-Tree needs to binary-search multiple levels of potentially large files. LSM-Trees employ many techniques to accelerate point lookups. Bloom filters [44, 58] serve as file-level filters that can terminate unnecessary file searches. A block cache [59, 119] caches data blocks from the SST file, reducing the number of disk IOs. A row cache [139] is a record-level cache that caches individual records in memory; it is more fine-grained than a block cache but does not apply to range scans.

Range scan. LSM-Trees are inefficient at range scans, as they need to search all levels of the tree to find all records in the given range and merge them to get the final result. Moreover, an LSM-Tree’s row cache does not apply to range scans, leading to low cache efficiency for scan-intensive workloads.

6.1.3 Modern NVMe SSD

Disks have improved significantly in the past few decades, especially with the advent of modern NVMe SSDs. Although conventional systems continue to work with new hardware, many of their assumptions have changed, particularly those designed around the limitations of conventional HDDs. As a result, the operational trade-offs of B-Trees and LSM-Trees warrant a reevaluation in the context of new hardware capabilities.

Random vs. sequential writes. Conventional HDDs use spinning disks to locate data, so sequential writes are much faster than random writes. Log-structured write is a good fit for such hardware as it only requires sequential writes. Modern NVMe SSDs, on the other hand, have no mechanical moving parts, use flash memory for efficient parallel access, and incorporate fast hardware garbage collection schemes [175, 178]. This allows modern NVMe SSDs to have almost as fast random write as sequential write; recent studies [68, 182] suggest that 4KB random writes can almost saturate an SSD’s bandwidth, making log-structured writes less attractive.

Kernel-bypass I/O. As discussed in Chapter 7, modern NVMe SSDs have I/O latency comparable to a context switch and bandwidth that is a significant fraction of memory bandwidth, making the conventional kernel-mediated I/O path (with its context switches and dual copies) a real bottleneck. Bf-Tree uses `io_uring` through the substrate of Chapter 7, sharing that infrastructure with LiquidCache; see that section for the broader trade-off between `mmap`, `SPDK`, and `io_uring`.

B-Tree page size. Smaller page sizes are preferred to reduce amplification, especially on modern NVMe SSDs that support finer-grained access. Bf-Tree therefore by default uses 4KB page size instead of more common 8-32 KB page sizes [60, 81, 98, 127]. Recent evaluation [68] on B-Tree page size shows that modern SSD favors 4KB page size, smaller page size results in worse IOPS and latency due to excessive overhead in the flash translation layer and not optimized 512-byte page access. This is consistent with our preliminary experiment that 4KB page size is the sweet point on a PCIe 4.0 modern NVMe SSD [144].

6.2 Bf-Tree Architecture

Two major problems of conventional B-Trees – write amplification and inefficient caching – stem from the fact that disk pages are much larger than individual records. This granularity mismatch fundamentally limits the performance of B-Trees, forcing them to either cache cold records or write the entire page for a single record update.

A B-Tree’s page organization is suitable for block devices and naturally implements page-based caching. However, the cache does not have to be page-based, as memory is byte-addressable. Bf-Tree starts with this thought experiment: *What if cached pages can have variable lengths?* By having variable-length pages, memory caches no longer need to align to disk pages. Instead, they have the freedom only to cache the data that is worth caching. In Bf-Tree, this means it can cache (1) only the “read hot” part of a page to serve reads efficiently and (2) the “write hot” part of a page to absorb updates (often in-place) as much as possible before batch writes to disk. It can also grow to cache the entire range gap if needed.

This section presents the high-level design of Bf-Tree, a modern B-Tree re-imagined with a native variable-length buffer pool for memory. Bf-Tree can leverage variable-length cache entries, called *mini-pages*, to support efficient point lookup, range scan, and write operations. While this design can help address the aforementioned problems, it comes with

new concurrency, memory management, and resource utilization challenges. Section 6.3 discusses how such a buffer pool can be efficiently implemented for a Bf-Tree.

Figure 6.2 shows the high-level architecture of the Bf-Tree. It consists of four parts: (1) the inner nodes; (2) the buffer pool that caches mini-pages; (3) the on-disk leaf pages; and (4) the mapping table for leaf and mini pages. At a high level, the Bf-Tree's architecture is not much different from the conventional B-Tree, except that its buffer pool supports variable length pages.

The rest of this section discusses the design of each component in detail, specifically, where we store them in memory or on disk and how they interact with each other. The last sub-section discusses the optimizations we implemented to improve Bf-Tree's performance.

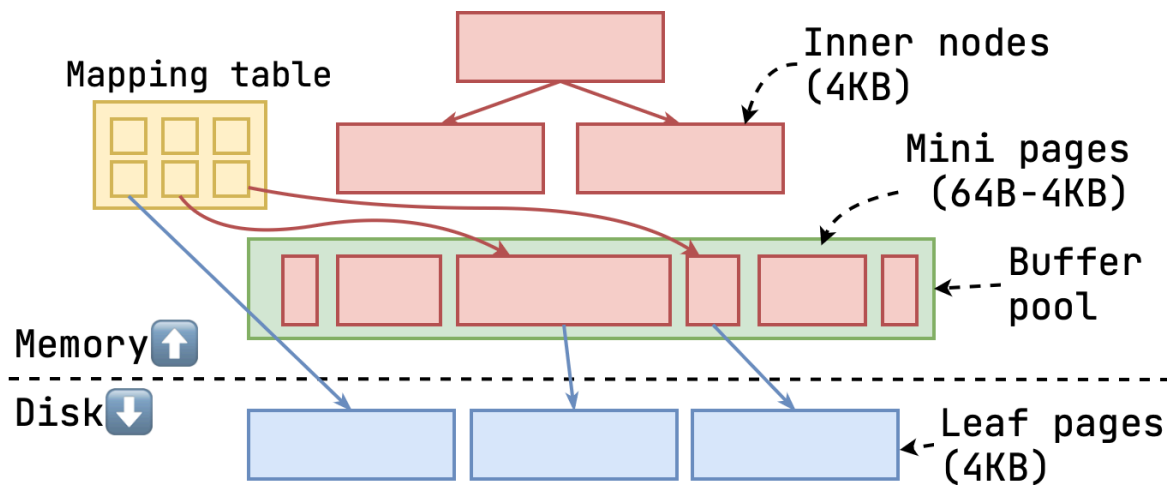


Figure 6.2: High level architecture of Bf-Tree. Like conventional B-Tree, but pages in the buffer pool are variable lengths.

6.2.1 Mini-page

A mini-page is an in-memory slim version of the corresponding leaf page. It serves two purposes: (1) to buffer recent updates and (2) to cache frequently accessed records. Mini-pages are for leaf pages only (i.e., not for inner nodes), and each leaf page may have *at most one* corresponding mini-page.

Records in a mini-page are maintained as sorted, preserving spatial locality. This allows records to be efficiently searched using binary search, unlike the delta chain approach that requires a sequence of pointer chases to find the target record. We next discuss the high-

level functionality of a mini-page. Section 6.4 discusses in detail how mini-pages are used in Bf-Tree's core operations.

Absorbing write operations. A write operation tries to insert to the mini-page of the destination leaf page. If the leaf page does not already have a mini-page, it creates a minimal-sized (e.g., 64 bytes to align with a cache line) mini-page that can contain the new record. If the mini-page is full, it grows to accommodate the new records. Each time, the mini-page doubles its size until it can accommodate the new record. Ultimately, the mini-page can grow too large (up to 4KB), which can cause the insertion/search performance to degrade. Then, or when it needs to be evicted from memory, we batch-write and merge the mini-page into the base leaf page.

We discuss new mechanisms for a fast concurrent buffer pool implementation for mini-pages in Section 6.3.

Caching hot records. Before reading the leaf page from disk, a read operation first (binary) searches the mini-page for the desired record and terminates early if the record is found. Searching the mini-page is efficient as the records are sorted and in-memory. If the record is not found in the mini-page, we load the corresponding leaf page from disk.

After reading the record from disk, we can cache it by inserting it into the mini-page. This allows future read operations to terminate the search early. Note that the mini-page will cache the individual records, not the entire page, avoiding the inefficient page-level caching of conventional B-Trees. To avoid flooding the mini-page with cold records, we cache the records from the leaf page only with a tunable probability (20% by default; studied in Section 6.5.11). Caching hot records is implemented as an in-memory insert operation to the mini-page, which may trigger the mini-page to grow as needed.

Caching range gaps. Record caching does not help with range scans because the range query has to look at the leaf page for the full set of records; in other words, record caching breaks the spatial locality needed for range scans. On the other hand, a page-level cache is ideal for range scans, as it preserves the spatial locality of records. Mini-pages support caching range gaps: when a mini-page grows to the full size, we merge (if necessary) and convert it into a full leaf page mirroring disk. Unlike systems such as RocksDB, where static memory partitioning is required for row-cache and block-cache, mini-pages in Bf-Tree automatically adapt to workload changes (e.g., point or range query intensive). This allows the same memory budget to be used for both range and point queries.

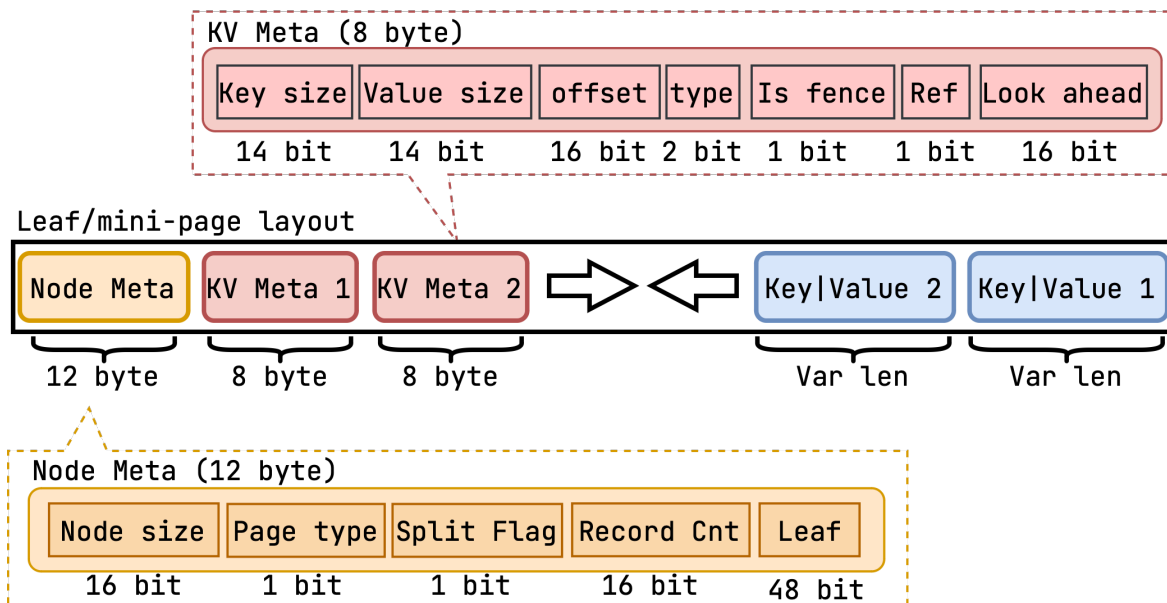


Figure 6.3: Mini-page (var len) / leaf node (4096 bytes) layout

6.2.2 Mini and leaf page layout

Mini-pages and leaf pages share the same layout, storing key-value pairs in sorted order and allowing efficient lookups. In Bf-Tree, they share the same implementation, except that mini-pages can have varying lengths. This significantly reduces the complexity of the system and allows us to implement optimizations once and apply them to both mini-pages and leaf pages.

As shown in Figure 6.3, the page layout starts with a 12-byte Node Meta, which encodes the node size, page type (mini or leaf page), split flag (whether the page is full), and value count (the number of records on the page).

The Node Meta is followed by an array of KV Meta, which stores the metadata of the key-value pair. The metadata and key-value data are stored separately to support variable-length keys and values. The KV Meta is stored from the beginning of the page, and the actual key-value data is stored from the end of the page. The node is full when the KV Meta and the key-value data meet in the middle. Separating the metadata and the actual data also allows efficient insertion, as we only need to shift the metadata when inserting a new record instead of shifting the entire node.

Each KV Meta is 8 bytes, which stores the length of the key and value and the offset of the key and value in the page. It also stores the type of the key and value (more detail on Table 6.1), whether the key is a fence key (Section 6.2.5), the reference flag of the key-value pair (Section 6.3.1), and finally, the look-ahead bytes of the key (Section 6.2.5). The key-value data is stored sequentially on the page, starting at the specified offset in the KV Meta.

6.2.3 Leaf pages and mapping table

Leaf pages are always on disk, and may have at most one associated mini-page in memory. To locate the leaf page and its mini-page, the last level inner node stores a page ID, which references a mini-page or a leaf page. Bf-Tree maintains a mapping table that maps the page ID to the actual location. If the page ID points to a mini-page, its leaf page address can be found in the leaf field of the page header.

As shown in Figure 6.2, the mapping table is an in-memory container that maps the logical page ID to the physical location of the page (i.e., memory address or disk offset). The mapping table can be implemented using a hash table/indirection array [60, 86, 127], relying on the OS's page table [39, 76], or using pointer swizzling [98]. Bf-Tree uses an indirection array for its simplicity and performance. However, our design can work with any of the above approaches, as the mapping table is decoupled from the rest of the system.

In addition to the address translation, the mapping table stores the reader-writer lock for each page.

The reader-writer lock (16 bits) is co-located with the page address (48 bits); they are together packed into a 64-bit word for efficiency. As the mini-page and its leaf page share the same page ID, locking the mini-page will also lock the leaf page, simplifying the locking mechanism and reducing overhead.

6.2.4 Inner nodes

Conventional B-Tree systems treat inner nodes and leaf nodes indistinguishably and use the same mapping table to translate an inner node page ID to a physical address. This creates a high overhead of inner node access (due to translation) and a contention hotspot on the mapping table. Inner nodes of practical B-Tree systems usually take less than 1% of the total B-Tree size [45, 65], and they are much more frequently accessed than leaf pages (e.g., every leaf page access will involve multiple inner node accesses).

Bf-Tree by default pins the inner nodes in memory and uses direct pointer addresses to reference them. This allows a simpler inner node implementation, efficient node access, and reduced contention on the mapping table. As inner nodes are pinned to memory, the buffer pool of Bf-Tree only needs to cache leaf pages. Pinning inner nodes to memory is not a design requirement, when deployed with constrained memory budget, users can choose to disable inner node pinning, or only pin the first few levels of inner nodes to memory. When an inner node is not pinned to memory, it will be accessed through the mapping table using a page ID instead of a direct memory address (similar to leaf pages).

Bf-Tree implements optimistic latch-coupling [100] on inner nodes to reduce their contention – based on the observation that although highly contended, inner nodes are rarely modified (only modified on nodes split/merge).

Specifically, we use an 8-byte version lock to track the state of the inner node. A read operation compares the version number before and after the operation and only proceeds when the version number matches. A write operation will acquire an exclusive lock on the inner node and bump the version lock after the modification. Optimistic latch coupling for inner nodes allows Bf-Tree to scale to high concurrency, as the read operation does not pollute cache lines, thus avoiding cache coherence traffic.

6.2.5 Performance optimizations

Now we discuss the optimizations made to Bf-Tree’s node layout, which apply to inner nodes, leaf nodes, and mini-pages.

Fence keys. Fence keys guard the key range of a page and determine the neighbor nodes for range queries. The first key of a mini/leaf page is the low fence key, and the second is the high fence, followed by the actual records. The low fence points to the node’s left neighbor, and the high fence points to its right neighbor. Alternative approaches are possible, e.g., chained pointers. Bf-Tree uses fence keys for their simplicity.

Prefix compression. Keys of Bf-Tree can have long prefixes, e.g., URL keys and names. To reduce memory usage and accelerate the key search, Bf-Tree implements prefix compression based on the fence keys. The node’s prefix is implicitly stored in the fence keys as they tell the node’s key range; i.e., the common prefix of the low and high fence keys is the node’s prefix. When inserting records into a node, the common prefix is skipped, and only the suffix is stored in the node; this reduces space consumption and allows higher fan out. To read the full key, we assemble the key’s prefix and suffix stored in the node.

Look-ahead bytes. Figure 6.2 shows that Bf-Tree stores the KV pair’s metadata and actual data apart.

While this design has the benefits mentioned earlier, it can incur higher random memory access for key comparison: we first load the record’s metadata, then use it to load the actual key. To accelerate this two-step pointer chasing, we store the first 2 bytes of the actual key (called look-ahead bytes) in the metadata and compare the look-ahead bytes first. Thanks to prefix compression, the first 2 bytes of the keys are usually different, and we can terminate the search early without loading the full key. We only need to load and compare the full key if the look-ahead bytes are the same.

6.3 Buffer pool for mini-pages

So far, we have described how mini-pages can significantly reshape the design of a B-Tree, and how Bf-Tree leverages mini-pages to achieve its efficiency goals. We next discuss how to manage the memory of mini-pages, as mini-pages can be of different sizes and grow and shrink dynamically.

The variable-length buffer pool has three challenges: (1) manages the memory of all mini-pages, i.e., the exact memory location of each mini-page, (2) tracks the hotness of each mini-page and evicts the cold mini-pages when needed. (3) concurrency challenge of evicting and allocating by many threads while maintaining memory safety and parallelism. The first challenge has to deal with memory fragmentation (like most allocators [57, 97]): When a large chunk of continuous memory is broken into smaller chunks of variable sizes, it is difficult to assemble them back to form a large chunk. The second challenge is to systematically decide which mini-page should be evicted and which should be kept in memory. The third challenge is safely issuing enough parallel IO requests to saturate SSD bandwidth.

Bf-Tree solves the problems above by designing a novel circular buffer to manage mini-pages: we store the mini-pages in the circular buffer, adding to the tail until the buffer is full. When it is full, the mini-pages close to the head address are evicted to disk, allowing the tail address to advance. This design is inspired by FASTER’s hybrid log design [31, 32], where a similar goal is achieved. We next discuss how the circular buffer is designed and implemented to work in a concurrent setting.

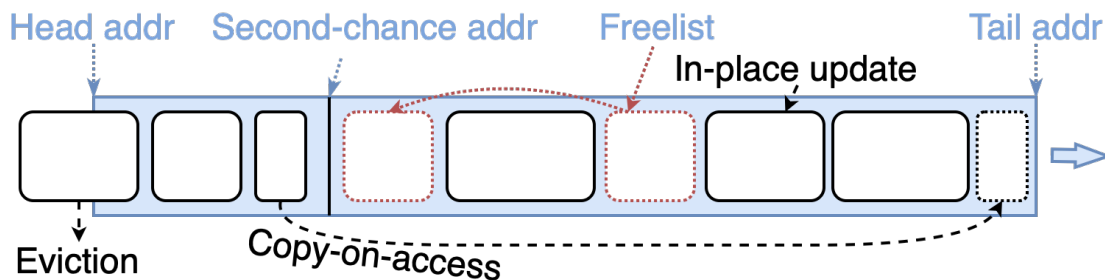


Figure 6.4: Circular buffer for variable length mini-pages.

6.3.1 Memory regions of circular buffer

The nature of the circular buffer evicts all mini-pages that reach the head address. This is undesirable behavior for hot mini-pages. As shown in Figure 6.4, the circular buffer has three addresses: the head, tail, and second-chance addresses. These three addresses divide the memory into 90% in-place-update region (between tail address and second-chance address) and 10% copy-on-access region (the rest). Mini-pages on the in-place-update region can be modified in place, while mini-pages on the copy-on-access region will be copied to the tail address on access (the copy mini-page to tail section). This prevents hot mini-pages from reaching the head address and getting evicted.

The circular buffer also maintains multiple *free lists*, each with a different size class, to track recently de-allocated memory. A mini-page de-allocation happens when it tries to grow or shrink. This involves first allocating a new mini-page, copying the content of the old mini-page to the new one, and finally de-allocating the old mini-page. The de-allocated memory is added to a free list and reused for future allocation.

6.3.2 Circular buffer API

The circular buffer provides a succinct yet efficient API to support various operations on mini-pages.

Alloc. Memory for a mini-page can be allocated from two places: the free list of the requested size category and the tail address. When the free list has no memory, we allocate from the tail address by advancing the tail address by the requested size. When the circular buffer is full, i.e., the physical locations of tail and head addresses are close to each other; the circular buffer returns an error. The caller will then call eviction to make room for new allocations.

Eviction. Eviction is the process of making room for new allocations. Eviction starts from the mini-page closest to the head address. A callback function is invoked to merge dirty records in the mini-page to the leaf page on disk. Then, the mapping table is updated to point to the leaf page, and the head address is advanced. Eviction may happen simultaneously from multiple threads (more below).

Dealloc. De-allocation simply adds the memory region to the corresponding free list, for reuse during a future allocation.

6.3.3 Performance optimizations

The circular buffer is central to Bf-Tree's design and can easily become a performance bottleneck if not optimized sufficiently.

Memory fragmentation. There's no paging concept in the circular buffer, i.e., mini-pages are allocated with no spaces between them. Each allocated mini-page has an 8-byte metadata that stores its size and state (e.g., ready or free-listed). The meta-data is stored right before the mini-page. This design minimizes fragmentation as mini-pages do not need to align to specific memory boundaries.

Memory alignments. Without paging, each mini-page is only aligned to an 8-byte boundary; this means that some mini-pages may cross the physical 4KB paging boundary, incurring additional page table lookup while accessing the mini-page. This is mitigated using the Huge Pages (HugeTLB) mechanism provided by the Linux kernel, where the entire physical memory of the circular buffer is backed by huge pages of 2MB or 1GB, heavily reducing the likelihood of mini-pages crossing the page boundary.

Concurrent evictions. Eviction ensures that cold mini-pages are evicted to disk, making room for hotter mini-pages. Eviction bumps the head address so that the tail address can be advanced. This is a sequential operation because the head address can only increase linearly. To parallelize the eviction process, we allow each thread to start evicting a mini-page concurrently but require all threads to finish the eviction in order, i.e., the head address can only be advanced when all threads have finished their eviction.

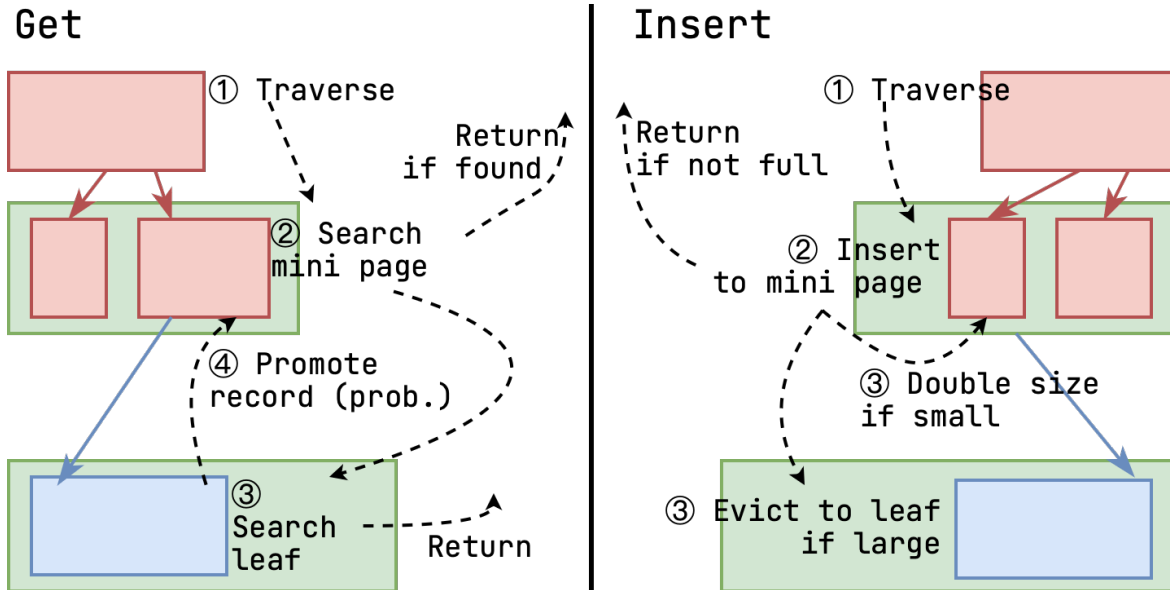


Figure 6.5: Bf-Tree's Get and Insert operations.

6.4 Bf-Tree core operations

So far, we have discussed the core mechanisms of Bf-Tree and the mini-pages. This section discusses how Bf-Tree connects the components to support efficient read, write, and range scans; Figure 6.5 illustrates the Get and Insert paths.

6.4.1 Get

The get operation starts with traversing the tree to the mini-page that may contain the target key-value pair and searching the mini-page for the key. If the record is found in the mini-page (cached), we will return the record and terminate the operation early. If the record is not found (or no mini-page exists), we search the corresponding leaf page on disk. We load the leaf page using the offset stored in the mini-page header (or the mapping table) and search the page for the target key-value pair. With a tunable probability (20% by default), we cache the record by inserting it into the mini-page, creating a new mini-page if one does not already exist. This allows subsequent searches for that record to complete in memory. Listing 6.1 summarizes the get operation in pseudo-code.

```
def get(key):
    (mini_page, leaf_page) = traverse(key)
    if mini_page:
        result = mini_page.binary_search(key)
        if result:
            # early terminate if found
            return result

    # otherwise, read from the leaf page
    result = leaf_page.binary_search(key)
    if rand() < 0.20:
        # with the default promotion probability, cache it in the
mini-page
        insert_or_create_mini_page(leaf_page, result)
    return result
```

Listing 6.1: Get operation

6.4.2 Insert

```
def insert(key, value):
    (mini_page, _leaf_page) = traverse(key)
    ok = mini_page.insert(key, value)
    if ok:
        # early terminate if insert succeeds
        return

    new_size = mini_page.next_size()
    if new_size == 0:
        # need to merge to the base page
        mini_page.merge()
        mini_page.insert(key, value)
        return
    else:
        # need to grow the mini-page
        mini_page.resize(new_size)
        mini_page.insert(key, value)
        return
```

Listing 6.2: Insert operation

Like the get operation, the insert operation starts with traversing the tree to the mini-page, creating a new mini-page if it does not already exist. The size of the new mini-page is just enough to fit the new record so that we have minimal write amplification. Then, we try to insert the record into the mini-page. If success, i.e., the mini-page has enough space to absorb the record, we will terminate the operation early. No IO (i.e., write to the leaf page) is needed in this case. If the mini-page is full, we will resize it (doubling its size) so it can fit the new record.

The resize process will allocate a new memory chunk and copy-initialize the mini-page to the new location. Once the mini-page is resized, we insert the record into the new mini-page. If the current mini-page is already large (e.g., 4KB) – the mini-page has already absorbed enough records – we merge it to the leaf page. The merging process will evict all dirty (modified) and cold records from the mini-page.

6.4.3 Range scan

Range scan involves scanning a range of records in the mini-page and leaf page and then merging the records to get the final result. This means the mini-page will not help reduce the IO for the range scan because it needs to load the leaf page anyway.

To mitigate this, Bf-Tree allows a frequently scanned mini-page to grow to the full page size and cache the entire page in the circular buffer. Thus, the buffer pool can also behave as a page-level cache. Page-level caching fundamentally differs from record-level caching because it caches both individual records and entire range gaps. Caching the entire leaf page provides us a simple way to handle not only efficient range scan but also gap locking [65], negative lookup, etc.

6.4.4 Delete/Update

Deleting a record is essentially inserting a tombstone record into the mini-page. When a read operation reads the tombstone in the mini-page, it returns a not-found result without touching the leaf page. When the mini-page is merged to the leaf page, the tombstone record is removed from the leaf page, if it exists.

Like Delete, an update operation inserts the record into the mini-page. Future reads can directly read from the mini-page. The record is updated on the leaf page when the mini-page is merged.

6.4.5 Mini-page operations

Mini-page merge. There are two cases where a mini-page might be merged: (1) the mini-page is too large, and (2) the mini-page is cold. A mini-page grows in size to cache/buffer as many records as possible. But it can not grow arbitrarily large, as records in the mini-page are sorted; a large mini-page incurs high insertion overhead. When a mini-page grows to the full page size (4KB), it is merged with the leaf page on disk to become a 4KB mini-page that mirrors the leaf page.

A mini-page will be evicted (merged to the leaf page) if it is not accessed while in the second-chance region, as discussed in Section 6.3.1.

To merge a mini-page with its leaf page, we first locate its leaf page using the `leaf` field in the mini-page header. Then, we calculate the space needed for the leaf page to accommodate the mini-page; if the leaf page does not have enough space, we first split the

leaf page and then insert the records of the mini-page into the corresponding leaf pages. Once all records are merged into the leaf, we can discard the mini-page and reuse the memory.

Copy mini-page to tail. When a mini-page is accessed while it is in the second-chance region, it is copied to the tail address. This prevents a frequently accessed mini-page from reaching the head address and, thus, being evicted to disk. We simply allocate a new memory chunk from the circular buffer and copy the mini-page to the new location. The old address is marked as a tombstone, and eviction is not triggered. While copying a mini-page to the tail, we also remove cold records from the mini-page (discussed below). This means that a record in the mini-page is evicted to disk if it is not accessed while it is in the in-place update region, and the entire mini-page is evicted to disk if it is not accessed while in the second-chance region.

Evicting cold records. Mini-pages cache hot records, which can become cold over time. Cold records are evicted to disk when a mini page is copy-on-accessed in the second-chance region. Bf-Tree takes this opportunity to examine all records and only keep hot ones on the new mini page. Hot/cold records are differentiated by their *reference bit* in the meta data; if set, the record is kept, otherwise evicted. The reference bit is set when a record is accessed. When eviction starts, we evict all records whose reference bit is 0 and clear all other reference bits. If a cold record is a cache record (read cache or phantom record), we can directly discard it without writing back to the leaf page. If the record is dirty (insert or tombstone), we merge the entire mini-page to the leaf page (maximizing the utility of the disk write) while retaining the hot records in the mini-page.

Leaf page split. Like a conventional B-Tree, a leaf node splits into two nodes when it is full. For Bf-Tree, an insert always tries to insert to the mini-page first. A full (4KB) mini-page that can no longer absorb new records triggers an eviction, followed by a split operation of the leaf page. When the split happens, each of the remaining records in the mini-page is compared with the split key to determine which leaf page should be inserted.

6.4.6 Handling negative search

A common problem with existing record-caching systems is that they do not handle negative searches well. A negative search is an operation that tries to find a key-value pair that does not exist in the system. Existing record-caching systems only assert the existence of a key-value pair; if a record is not found in the cache, it either means the record is not

cached or means the record does not exist. Those systems then need to look up the leaf page to confirm the record’s existence, which is inefficient.

Bf-Tree solves this problem as follows: we cache the negative search result by inserting a phantom record into the mini-page. The observation is that the negative record is like any frequently searched record; we cache them in the mini-page. We note that advanced techniques such as Bloom filters [58] can improve the negativity test, but we leave this as future work as they may add significant design complexity.

Record type	Dirty?	Existence?
Insert	True	True
Cache	False	True
Tombstone	True	False
Phantom	False	False

Table 6.1: Four types of records in a mini-page.

So far we have discussed four types of records (Table 6.1) in a mini-page: insert, cache, tombstone, and phantom. Each of the types indicates the dirty and existence property of a record – a dirty record must be written back to the leaf page when the mini-page is merged, and the existence tells whether a record exists in the system.

6.4.7 Snapshotting, logging, and recovery

Bf-Tree is compatible with standard snapshotting, logging, and recovery mechanisms. Bf-Tree currently implements a simple ARIES-style [126] physiological logging mechanism; more advanced mechanisms are left as future work. **Logging.** Before any write operation is committed, it must append a log entry to the WAL (write-ahead-log) and wait (log full or default 1ms interval) until the log is flushed. The log entry points to a page on disk and describes an operation (e.g., insert, delete). **Snapshotting/Checkpointing.** Like Aurora [167], Bf-Tree checkpointing is done asynchronously and continuously offline by replaying the WAL entries in parallel. Online snapshotting is also implemented by pausing the write operation and writing back the dirty mini-pages to disk. The mapping table is appended to the last pages of the snapshot file. Inner pages are also written back to disk for faster/simpler recovery. A special mapping table is generated and appended to the snapshot file to map the inner pages’ virtual memory addresses to the physical disk offset. **Recovery.**

Bf-Tree recovery consists of two steps: (1) build the in-memory representation from the snapshot file, (2) replay the WAL to recover to the latest state. We first load the special mapping table mentioned above to rebuild inner pages. Then for every inner page, we use the mapping table to find the physical disk offset and load it into memory. We recursively resolve and load its children’s pages. Replaying the WAL consists of simply finding the corresponding page and re-applying the operation.

6.5 Evaluation

Bf-Tree re-designs the traditional B-Tree using variable-length buffer pools and mini-pages. In this section, we compare the performance of Bf-Tree with state-of-the-art key-value stores. Specifically, we aim to answer the following questions:

- How does Bf-Tree compare against RocksDB [48], conventional B-Tree, its modern variants [22, 38, 102], and Leanstore [98] on various workloads?
- How does Bf-Tree perform on different workloads, contentions, and cache sizes?
- Where does Bf-Tree spend its time on its different components?

6.5.1 Experimental setup

We run a YCSB-like benchmark with 200 million initial records, each of which is 32 bytes (16-byte key and 16-byte value). We used a Zip-f distribution with a skew factor of 0.9 (80% of requests access 33% of records); more distributions will be explored in the skewness section

The default workload consists of 50% of reads and 50% of writes with 2GB memory cache. We warm up the system, repeat the benchmark five times, and report the best throughput.

We implemented Bf-Tree in 13k lines of Rust. We use the latest `io_uring` feature of the Linux kernel for efficient IO. Specifically, we use the kernel polling mode [17, 68], which creates dedicated kernel polling threads to perform direct IO with zero system calls and bypass the OS page cache. The benchmark is performed on a CloudLab machine `sm110p`, with 32 hyper-threads clocked at 2.4GHz, 128 GB of memory, and 1TB of NVMe PCIe 4.0 SSD with over 600k IOP/s. It runs Ubuntu 22.04 with kernel 5.15 and ext4 as the filesystem. Logging/snapshotting/checkpointing are orthogonal to the core design of Bf-Tree and are disabled for all baselines.

We use lightweight formal methods [26] to validate the correctness of Bf-Tree implementation. Specifically, we employ differential fuzzing to check that Bf-Tree acts semantically the same as our reference model (the B-Tree in Rust’s standard library), run Bf-Tree on CloudLab [50] using with libfuzzer [146] with address sanitizer [145] to continuously check for memory issues (e.g., memory leak, use-after-free). We use shuttle [26] to deterministically and systematically explore different thread interleaving to uncover the bugs caused by subtle multithread interactions. We leverage Rust’s reference model [83] to statically check that all accesses to mini-pages are safe.

6.5.2 Baselines

We compare Bf-Tree with (1) RocksDB (commit 54d6286); (2) B δ -Tree (explained below); (3) conventional B-Tree; (4) Leanstore (commit 677126c); and (5) TwoTree. This section discusses how we set up and configure the baselines. Due to the nature of the complex high-dimensional configuration space, we only highlight the key configurations of each of these systems.

RocksDB. RocksDB is the state-of-the-art LSM-tree-based key-value store. It is widely used in production and is highly optimized for both read and write workloads. RocksDB has three caching components: memtable, block cache, and row cache. The memtable absorbs write operations and is flushed to disk when it is full; the block cache caches the blocks of sstables, while the row cache caches individual records of sstables. It is up to the users to configure how to use the memory in these three caches.

We spend half of the memory on writing and half on reading, among which half is on the block cache and half on the row cache. For better performance, we enabled direct IO and disabled fsync and WAL.

Conventional B-Tree. We implemented a conventional B-Tree from scratch in Rust. For fair comparisons, the conventional B-Tree enjoys all optimizations from Bf-Tree except the mini-page design, including hybrid latching, fence keys, prefix compression, look-ahead bytes, and `io_uring` based IO. In other words, the conventional B-Tree is Bf-Tree but with page-level caching.

B δ -Tree. To study the effect of delta records – like Bw-Tree [102] – we implemented a B-Tree with delta records, which we call the *B δ -Tree*. For fair comparisons, this implementation is also in Rust and has all the optimizations in Bf-Tree, except for the mini-page design. The B δ -Tree is implemented to be as close in design as possible to the Bw-Tree, a

state-of-the-art B-Tree used in production. To limit delta chains’ memory consumption, we store all delta records and page cache into the same circular buffer similar to that from Bf-Tree, thereby preventing the system from using excessive memory.

Leanstore. Leanstore [98] is a B-Tree optimized for both in-memory and larger-than-memory workloads by reducing the overhead of the mapping table; Bf-Tree instead focuses on improving caching efficiency and reducing write amplification. We include Leanstore mainly to verify that Bf-Tree’s in-memory performance is not compromised by its novel buffer pool design.

TwoTree. TwoTree [193] is a new index structure that reuses existing components to build record caching systems. We compare with TwoTree to verify that our record caching (achieved by mini-pages) is as performant as TwoTree.

6.5.3 Overall performance

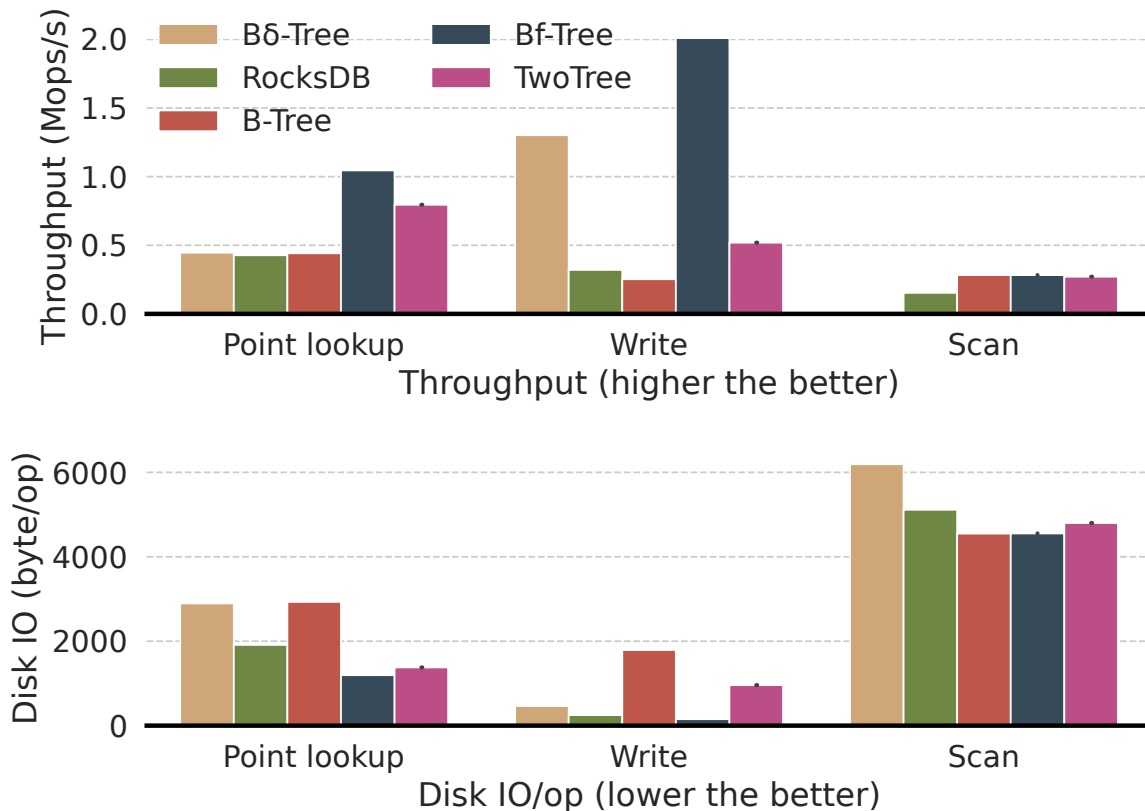


Figure 6.6: Bf-Tree vs. baselines on various workloads – Bf-Tree is the only system performing well on all three representative workloads.

This experiment examines how the five systems perform on the three most important workloads in practice: point lookup, write, and scan. We use the default benchmark setup mentioned in Section 6.5.1 and run each workload separately. Figure 6.6 (top) shows the throughput (the higher the better) of the systems in these workloads with 31 threads, and Figure 6.6 (bottom) shows the corresponding disk IO per operation (the lower the better). Figure 6.1 previewed this result as a three-dimensional summary across point lookup, scan, and update.

For write, conventional B-Tree performs the worst, as a single record update would incur a full page write, as evidenced by the highest disk IO per operation. B δ -Tree, RocksDB, and Bf-Tree all implement write buffers that batch write operations, leading to lower disk IO per operation. Bf-Tree is faster than B δ -Tree because its mini-pages can grow larger and absorb more writes before flushing, whereas B δ -Tree caps its delta chains at about 10 records. While having higher disk IO/operation, B δ -Tree performs similarly to RocksDB due to its more efficient IO handling (e.g., zero-copy IO) and more efficient in-memory data structure.

For point lookup, Bf-Tree and TwoTree stand out, as they employ record-level caching in the mini-page, which is more efficient at identifying individual hot records. B δ -Tree and B-Tree use page-level caching, which leads to similar throughput and disk IO. RocksDB's row cache also helps to improve caching efficiency, as indicated by the fact that its disk IO is lower than B δ -Tree and B-Tree. However, RocksDB's in-memory skip list is less efficient than B-Tree based implementations, leading to similar throughput.

For range scan, all B-Tree based systems perform similarly, as they all cache the entire leaf page in the memory, preserving record locality and allowing efficient scan. RocksDB performs the worst, as it needs to scan all levels of the sstables and merge the results, leading to higher disk IO per operation.

This experiment provides a high-level view of the system's performance on different workloads. It shows that Bf-Tree is the only system that can perform well in all three representative workloads, making it a good choice for general purpose key-value store.

6.5.4 Scalability

This section examines the scalability of Bf-Tree and the baselines for larger-than-memory and in-memory workloads.

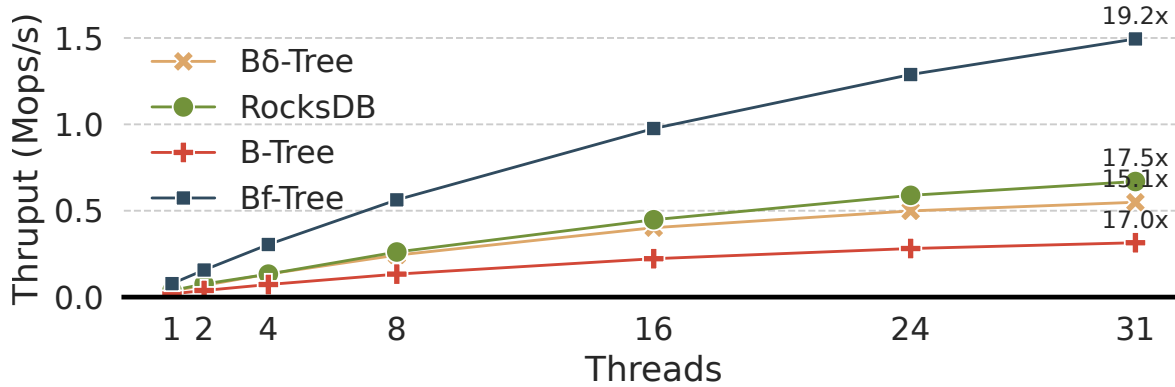


Figure 6.7: The scalability of Bf-Tree vs baselines. – Bf-Tree scales well before saturating the SSD bandwidth. RocksDB does not saturate the bandwidth at thread 31.

Figure 6.7 uses the default workload (50% read and 50% update) and varies the number of threads. All systems scale well before reaching the maximum disk bandwidth. Among them, Bf-Tree performs the best, with the lowest write amplification and the most efficient caching. Its efficient io-uring-based IO handling also allows it to scale well without being bottlenecked by the kernel overhead.

Bf-Tree, Bδ-Tree, and B-Tree saturated the disk bandwidth at 31 threads, with a scale factor of 19.2×, 15.1×, and 17.0×, respectively. Bδ-Tree is worse than RocksDB (17.5×) because it does not have recording-level caching like the row cache in RocksDB. B-Tree is the worst among the four systems, as its page-granularity caching is inefficient and incurs high write amplification.

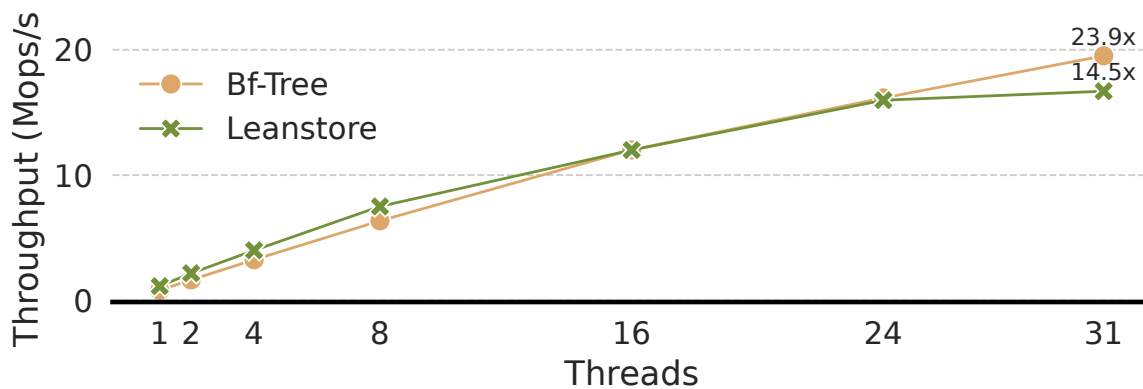


Figure 6.8: In-memory performance compared with Leanstore. – When data fits in memory, Bf-Tree, B-Tree, Bδ-Tree fold in a similar design. This is a sanity check that Bf-Tree’s in-memory performance is as good as the state-of-the-art Leanstore.

Figure 6.8 uses the same workload as the previous experiment but ensures that the data fits in memory. When data fit in memory, Bf-Tree, B-Tree, and B δ -Tree fold in the same design. This experiment compares them with Leanstore, a state-of-the-art B-Tree that is highly optimized for in-memory workloads. Leanstore uses pointer swizzling to reduce the overhead of the mapping table; its techniques are complementary to Bf-Tree’s, we include Leanstore here to sanity check that Bf-Tree’s in-memory performance is not compromised by its mini-page design. At low thread count, Leanstore performs slightly better than Bf-Tree, as it avoids the mapping table overhead in Bf-Tree, specifically, for each operation, Leanstore may have one less cache miss than Bf-Tree. At high thread count, Bf-Tree performs slightly better than Leanstore, reaching a scale factor of 23.9 $\times$ as opposed to 14.5 $\times$ of Leanstore, indicating that Bf-Tree is more efficient at handling contention. Note that Bf-Tree’s techniques can be applied to Leanstore, and vice versa, and we leave it to future work to apply the Leanstore’s techniques to Bf-Tree. Leanstore performs similarly to baseline B-Tree for larger-than-memory workloads, as they both use page-level caching.

6.5.5 Latency

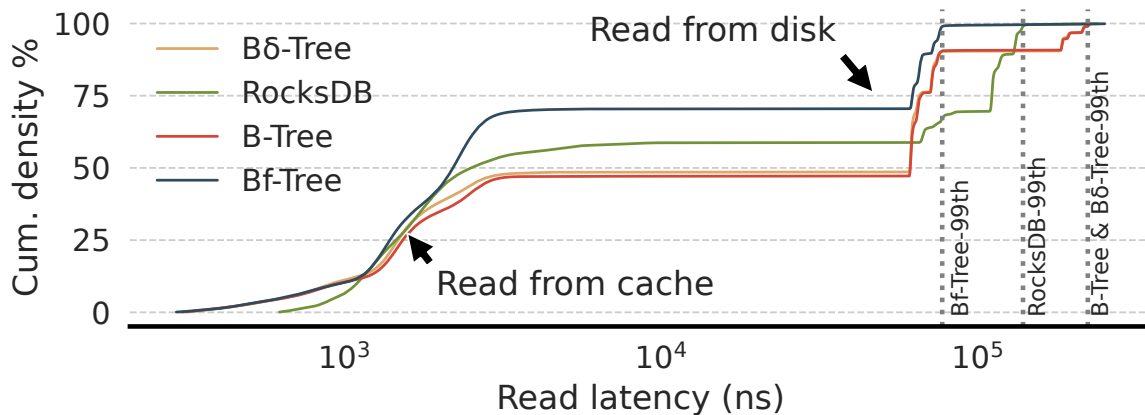


Figure 6.9: Latency distribution of Bf-Tree vs baselines. – Bf-Tree has the best 50th and 99th percentile latency due to its efficient caching and IO handling.

This experiment examines the latency distribution of Bf-Tree and baselines with read-only workload and single thread execution. The x-axis shows the latency in nanoseconds, and the y-axis shows the cumulative latency distribution, starting from 0% to 100%. We measured the end-to-end latency of the systems, i.e., from issuing the request to the system

returning the result. We used dashed vertical lines to show each system's 99th-percentile tail latency.

Figure 6.9 shows that all systems present a two-stage latency distribution: when data is cached in memory, the latency is around 1 μ s, and when accessing data from disk, the latency increases 100 $\times$ to around 100 μ s. Bf-Tree's variable length mini-pages allow it to cache more records in memory, leading to an almost 75% cache ratio. In contrast, all other systems can only cache around 50% of the records, leading to the lowest 50th percentile latency of 1.18 μ s, while B δ -Tree, RocksDB, B-Tree has the 50th percentile latency of 1.61 μ s, 1.86 μ s, and 58.7 μ s, respectively. When data is larger than memory, Bf-Tree has the lowest 99th percentile latency (70 μ s), almost 2 $\times$ lower than RocksDB (131 μ s), thanks to its efficient zero-copy zero-syscall IO handling.

The multiple latency spikes in RocksDB are due to the LSM tree's multi-level structure. The last few latency spikes of B-Tree and B δ -Tree are due to cache replacement, where new pages are being promoted to cache and old pages are being evicted to disk. Bf-Tree also presents a similar stair pattern but a much smaller spike due to its high cache ratio.

6.5.6 Skewness

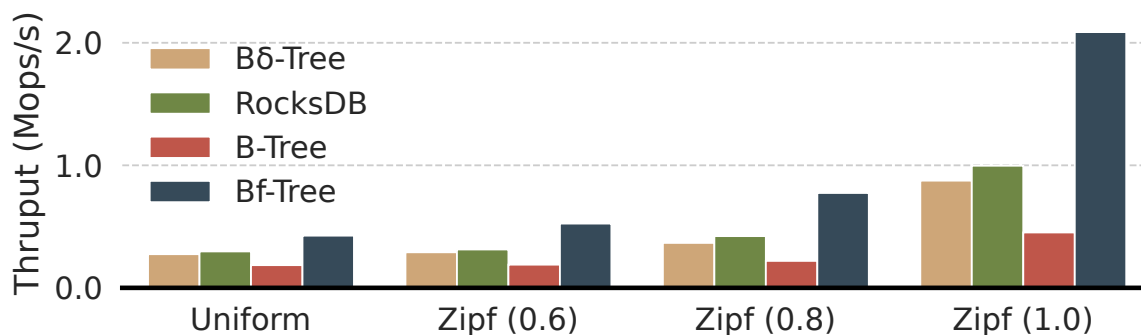


Figure 6.10: Impact of workload skewness. –Bf-Tree performs consistently the best across all skew levels.

Figure 6.10 shows the impact of workload skewness on the systems' performance, from Uniform distribution to Zipf distribution with a skew factor of 1.0. This experiment evaluates how well the systems can handle workload skewness and whether they can handle high contention. The performance of all systems increases as the workload becomes more skewed, as the hot records are more likely to be cached in memory. Bf-Tree consistently performs the best, and its performance is the most pronounced when the workload presents

a high skew. This is because Bf-Tree’s record-level caching is more efficient at identifying hot records than the page-level caching used in B-Tree. RocksDB also has a row cache, but its row cache memory budget is divided among memtable, block cache, and row cache, leading to less efficient caching and Bf-Tree which can adapt the entire memory for record caching. Similarly, B δ -Tree’s delta chains and the page cache compete for the same memory budget, leading to less efficient caching than Bf-Tree.

All systems perform poorly on uniform distribution, as caching is less effective when every record has equal hotness. In this case, Bf-Tree still has the best caching ratio because its mini-page design can better handle the internal fragmentation of the leaf pages. Practical B-Tree leaf pages only have about 70% of the space used due to the nature of page splitting. When caching the entire page in memory, this leads to a 30% waste of memory. Bf-Tree’s mini-pages will dynamically grow and shrink to fit the actual number of records in the page, leading to a better caching ratio.

6.5.7 Read write ratio.

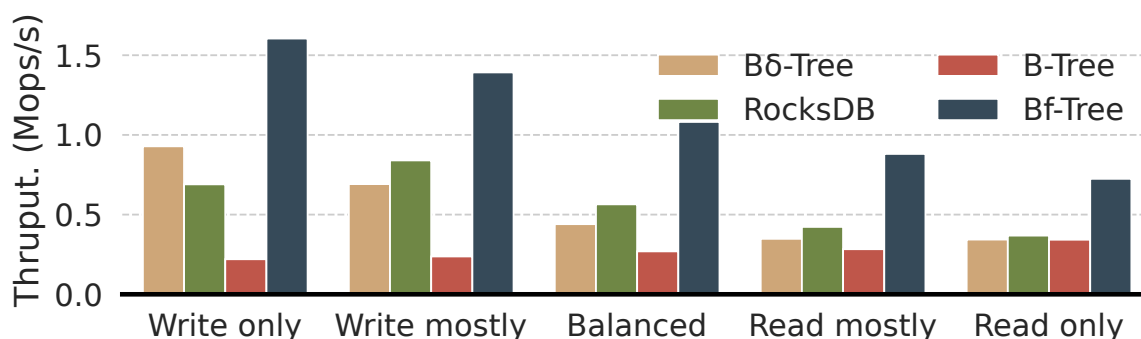


Figure 6.11: Impact of read and write workloads.

Figure 6.11 shows the impact of read and write workloads on Bf-Tree and baseline systems. For systems that implement buffered write, e.g., Bf-Tree, B δ -Tree, and RocksDB, the throughput is higher with more write operations. This is because most write operations are buffered in memory and only flushed to disk when the buffer is full. But for read operations, each cache miss incurs a random disk IO to read the data from the disk. For conventional B-Tree, a higher read ratio leads to better performance, as read operations incur one read IO, while write operations incur both read and write IO. Overall, Bf-Tree performs significantly better than the baselines in all read and write ratios. This is because Bf-Tree’s mini-page can

cache individual hot records and absorb write operations, leading to the best performance across all read and write ratios.

6.5.8 Cache sensitivity

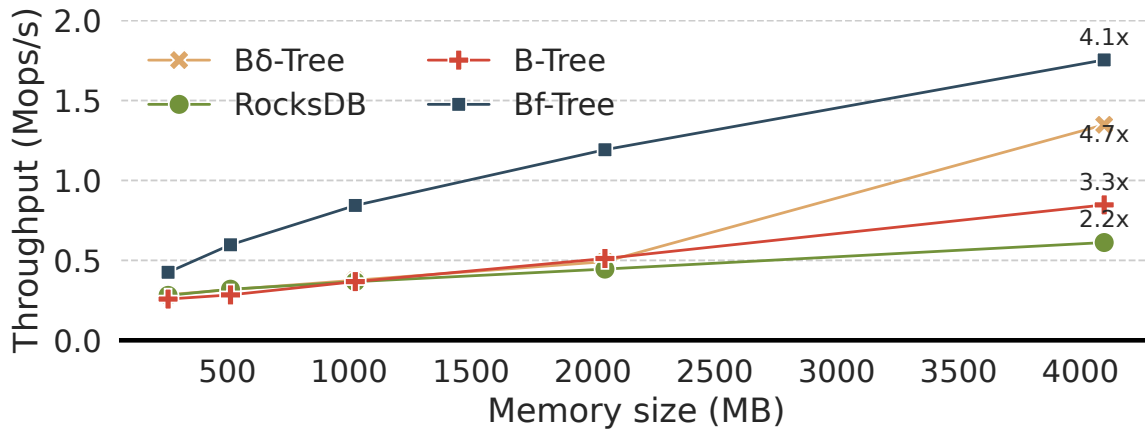


Figure 6.12: Impact of cache size.

Figure 6.12 examines the cache sensitivity of Bf-Tree and baseline systems. We used a read-only workload and varied the memory size from 256MB to 4GB, and reported how their throughput changed in response to the cache size. As memory size increases by 16 $\times$, RocksDB’s performance steadily increases but only to 2.2 $\times$, indicating a low caching efficiency. B δ -Tree and B-Tree perform similarly as they both use page-level caching for the leaf pages, with an increase of 4.7 $\times$ and 3.3 $\times$, respectively. In particular, both B δ -Tree and B-Tree have a slow performance increase when most data are on disk and a fast performance increase when most data are in memory, consistent with previous study [74]. Bf-Tree has a similar performance increase of 4.1 $\times$ and is consistently the best across all memory sizes. The gap between Bf-Tree and the baseline systems is wider when most data are on disk, showcasing Bf-Tree’s efficient caching mechanism. The gap becomes narrower when most data are in memory, as all B-Tree based systems fold to the same design when data fit in memory.

6.5.9 Time-component analysis

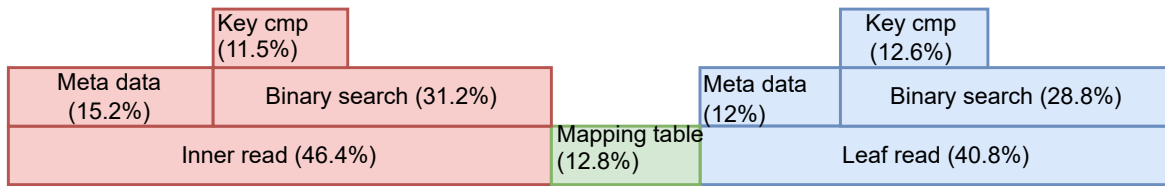


Figure 6.13: Time spent on different components of the system – Recreated from the flamegraph [66] for better clarity.

Figure 6.13 shows the flamegraph [66] of the Bf-Tree for a read-only in-memory workload. Larger-than-memory workloads present similar results, except that >95% of the time is spent on disk IO.

The inner node search takes roughly half the time, involving multiple inner node searches, one mapping table read, and one leaf node search. Searching an inner node first loads the metadata for key-value pairs, then binary searches the keys to find the next level pointer, and finally loads the next node. Each mapping table access first locates the slot from page ID, then loads and acquires a reader lock on the mapping table and uses the physical address to load the leaf page. Leaf page search is similar to inner-node search.

Multiple inner node searches spend similar time to one leaf node search. This is because inner nodes are more frequently accessed, most of which can fit in the CPU cache, while leaf node access is less frequent and almost always needs to load from memory. The mapping table uses only 12.8% of time, requiring only one memory access for address translation and lock acquisition.

Within inner and leaf node searches, they spend similar time on metadata load and binary search, as they have the same page layout and optimizations. Among binary searches, roughly one-third of the time is spent comparing the key bytes, and the rest is spent loading data from memory.

6.5.10 Copy-on-access region size

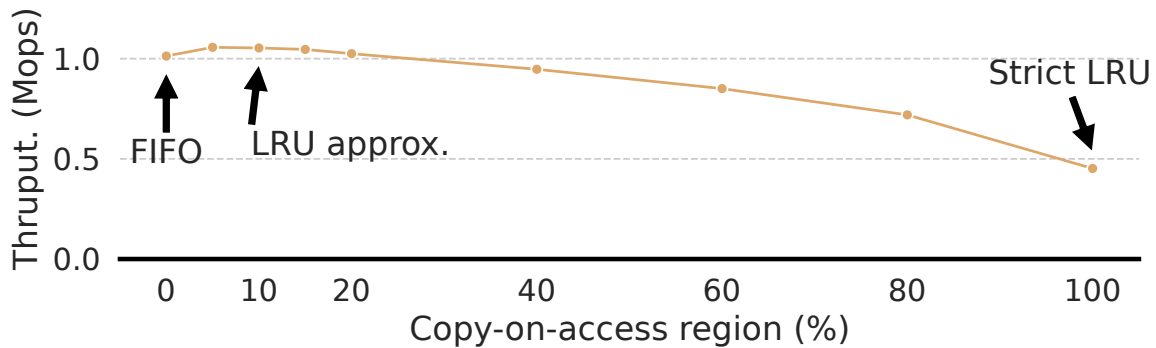


Figure 6.14: Impact of copy-on-access region size (% of total circular buffer size).

Figure 6.14 varies the copy-on-access region size from 0% to 100% of total circular buffer size, the y-axis shows the corresponding throughput. At 0%, the buffer acts like a FIFO queue, demonstrating low runtime overhead but low caching quality. At 100%, the buffer acts like a strict LRU cache, demonstrating high runtime overhead but high caching quality. Bf-Tree by default chooses a 10% copy-on-access region size, which balances the trade-off between runtime overhead and caching quality.

6.5.11 Impact on promotion rate

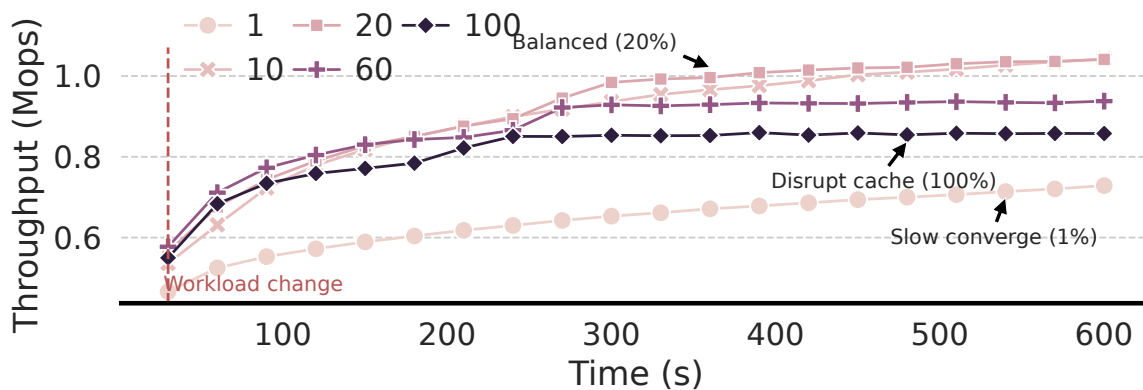


Figure 6.15: Impact of promotion rate: the probability of promoting a record from a disk page to its mini page.

Figure 6.15 varies the promotion rate (the probability of promoting a cold record to cache) from 1% to 100%. The x-axis shows the time since a workload change, and the y-axis shows the throughput. When the promotion rate is low (e.g., 1%), the system favors *caching frequency* and responds slowly to workload change but might converge to a higher throughput. When it is high (e.g., 100%), the system favors *caching recency* and converges quickly but ends up with a lower throughput as it disrupts the cache by promoting cold records. Bf-Tree by default chooses a 20% promotion rate, as it balances the trade-off between recency and frequency.

6.6 Closing the three-axis story

This chapter completes the three-axis White-Box story. LiquidCache (Chapter 4) attacked the *representation* axis by transcoding cached data into a query-friendly encoding chosen adaptively from the workload. SqueezeCache (Chapter 5) attacked the *content* axis by retaining only the fragments of each cached column that the workload actually uses, treating fidelity as a spectrum rather than a binary. Bf-Tree, presented in this chapter, attacks the *granularity* axis by decoupling cache pages from disk pages, so that the cache holds the judiciously chosen subset of each page worth caching – a few hot records, a buffer of recent updates, or the full page when range scans warrant it.

The Bf-Tree result restated. We presented a novel B-Tree variant optimized for point lookup, write, and range scan workloads. A mini-page design achieves low write amplification and efficient caching by separating cache pages from disk pages, and a novel variable-length buffer pool, built on a circular log, manages those mini-pages concurrently. We implemented Bf-Tree in 13k lines of Rust with strong correctness guarantees; on a YCSB-like benchmark Bf-Tree is 2× faster than RocksDB on range scans, 6× faster than a conventional B-Tree on writes, and 2× faster than both on point lookups, and matches in-memory B-Trees when data fits in memory.

Together with the I/O substrate of Chapter 7, the three contributions form a coordinated demonstration that a cache given permission to see inside the data it holds – the encoding, the content, the granularity – recovers work that the rest of the stack would otherwise repeat, query after query and update after update. The remainder of the dissertation discusses the engineering required to build and ship these ideas (Chapter 7), surveys the related work across all three axes (Chapter 8), and concludes (Chapter 9).

Chapter 7

Engineering Lessons from Building White-Box Caches

The three contributions of this dissertation – LiquidCache (Chapter 4), SqueezeCache (Chapter 5), and Bf-Tree (Chapter 6) – are not just research prototypes that live only inside a benchmarking harness. LiquidCache and SqueezeCache form a single open-source codebase of over fifty thousand lines of Rust, deployed in production by a small but real community of users; Bf-Tree is a separate 13,000-line implementation built on the same I/O substrate. This chapter is about what building and shipping those systems taught us – not the mechanisms themselves, which the preceding chapters covered, but the engineering required to make White-Box caching real.

Earlier chapters reported what White-Box caching buys: order-of-magnitude reductions in CPU, memory, and I/O. This chapter reports what it costs to build. A cache that understands the encoding, contents, and access granularity of its data has far more to get right than one that treats data as opaque bytes. A cache that chooses its own encoding must account for its own memory; a cache that manages its own disk must guarantee its own crash consistency; a cache that invents its own data layout must prove that the layout behaves like the one it replaced. The performance wins of the earlier chapters are only half the story; this chapter is the other half – the obligations those wins created, and the design choices that kept them manageable.

The engineering outcomes are concrete. The techniques are organized into five crates that let any analytical engine adopt them one at a time, and SqueezeCache now ships as the default eviction mechanism of the open-source project. The disk-management substrate is formally verified with Verus; the most aggressive data-layout deviation in the dissertation – squeezed strings – is checked continuously by differential fuzzing against Arrow’s reference implementation. Roughly five thousand lines of the underlying techniques have been merged upstream into Apache DataFusion, Arrow, and Parquet, where the StringView

representation and a rewritten Parquet decoder are now defaults that benefit engines with no connection to this dissertation.

Each section that follows isolates one of these obligations and the design choice that kept it manageable:

1. **Adoption.** White-Box techniques are invasive – they override decisions the host engine would otherwise make for itself – so they must be separable enough to adopt one at a time (Section 7.1).
2. **I/O ownership.** A cache that wants NVMe-class throughput and exact memory accounting must take over the I/O path from the OS, yet without forcing an I/O runtime on the engines that embed it (Section 7.2).
3. **Crash consistency.** A cache that manages its own disk inherits the filesystem’s correctness obligations, on exactly the code paths where bugs are unreproducible (Section 7.3).
4. **Representational equivalence.** A cache that invents a leaner data layout must remain behaviorally indistinguishable from the standard layout it replaced (Section 7.4).
5. **Metadata cost.** Fine-grained caching multiplies the number of entries to track, so the index over them must itself be nearly free (Section 7.5).
6. **Externalization.** Much of what makes a White-Box cache fast is not cache-specific at all, but a latent improvement to the shared substrate – best returned upstream (Section 7.6).

The chapter closes with the cross-cutting lessons these experiences add up to (Section 7.7).

7.1 A modular core for an invasive technique

A White-Box cache earns its performance by overriding decisions the host engine would otherwise make for itself – how bytes are encoded, when they are evicted, how predicates run against them. That is precisely what makes such techniques hard to adopt: a system that intrudes on so many of the engine’s responsibilities is easy to reject wholesale. The first engineering requirement, then, is that the intrusion be *separable* – that an engine can hand the cache the one decision it wants reclaimed and keep the rest. The LiquidCache codebase is structured to make that possible, splitting into five crates along the disaggregation boundary.

- **liquid-cache-core** is the cache’s engine-agnostic core.

It holds the Liquid format implementation, the transcoding pipeline, the pushdown evaluation logic, the squeeze mechanism, the variable-length buffer pool, and the disk-management substrate. As a self-contained library, it can be embedded in any analytical system, not just DataFusion.

- **liquid-cache-datafusion** is the DataFusion integration layer.

It provides a Parquet reader that transparently delegates to LiquidCache, an optimizer rule that rewrites query plans to push down filters and projections, and a TableProvider that swaps in for the standard Parquet TableProvider with a one-line code change.

- **liquid-cache-local** embeds the cache in the same process as the DataFusion engine – the deployment used when the cache and compute share a machine.
- **liquid-cache-server** and **liquid-cache-client** together form the disaggregated deployment.

The server runs the embedded query engine and serves filtered batches over Arrow Flight; the client exposes the same TableProvider as `liquid-cache-local` but transparently proxies to the server.

Together the five crates contain roughly fifty thousand lines of Rust, plus another five thousand lines contributed upstream to DataFusion, Arrow, and Parquet (Section 7.6). The split exists for more than tidiness. Because each crate corresponds to a single reclaimed decision – the format and squeeze logic in the core, disaggregation in the server/client pair, in-process embedding in `liquid-cache-local` – an adopter can take only the decisions it wants, each White-Box contribution can be tested in isolation, and external systems can pick up the Liquid format without the disaggregation, or the disaggregation without committing to DataFusion.

7.2 Owning the I/O path without imposing it

As long as a cache hands its bytes to the operating system, the OS page cache will cache them a second time, evict them on its own schedule, and hide their cost from the cache’s own memory budget. For a black-box cache that is a convenience. For a White-Box cache it is corrosive: the entire premise of SqueezeCache is that the cache knows, to the byte, what it holds and why – a claim the page cache silently falsifies by double-caching data behind the cache’s back and making its own replacement decisions. Taking control of representation and content therefore forces a third move: taking control of the I/O path itself. The difficulty

– and the substrate that LiquidCache (Chapter 4) and Bf-Tree (Chapter 6) share – is to do so without forcing that control onto the engines that embed the cache.

The conventional I/O path cannot keep pace with modern NVMe SSDs, whose bandwidth now approaches that of main memory. In one of our setups, memory bandwidth is about 80 GB/s while disk bandwidth is about 14 GB/s per drive; with five drives the aggregate reaches 70 GB/s. If we perform I/O using `pread`, the kernel incurs a double copy on each read – one from disk into the kernel page cache, another from the page cache into user space. The theoretical maximum bandwidth is therefore only half of memory bandwidth, around 40 GB/s, well below what the hardware can deliver.

To avoid the double-copy issue, three main approaches exist: (1) `mmap`, (2) kernel bypass via SPDK or similar, and (3) the modern Linux I/O stack, `io_uring`. `mmap` is an OS mechanism that maps a file into memory and lets the OS bring pages from disk on page fault; from the programmer’s perspective it is the least-friction approach. But `mmap` has serious drawbacks: it is unpredictable whether a memory access will cause a page fault, which is unacceptable for a user-space asynchronous runtime such as Tokio (in which a non-async function must be non-blocking), and `mmap` does not scale to high thread counts due to the overhead of setting up the page table and TLB shootdowns. Kernel bypass via SPDK often provides the highest performance, as it bypasses the kernel entirely and manages everything in user space; but it is hard to set up and maintain, requires the program to own the whole device with no filesystem in the way, and demands root privileges. `io_uring`, the modern Linux solution for efficient I/O, lets user-space programs and the kernel share the same set of buffers. Learning from the over-simplified `pread` interface, `io_uring` offers a comprehensive solution – but that flexibility comes at a cost. There are many ways to perform I/O with `io_uring`, each with trade-offs that are not obvious to the user. The rest of this section describes how we extract NVMe-class throughput from `io_uring` while keeping the cache’s interface indistinguishable from ordinary async code.

7.2.1 Async I/O in the Rust ecosystem

Owning the I/O path means confronting how immature asynchronous file I/O still is. Unlike the synchronous `pread` API, where the OS abstracts away the complexity of I/O management, efficient I/O requires coordination across the entire stack: (1) the OS must expose a non-blocking I/O primitive; (2) the user-space asynchronous runtime must build on that primitive, ideally behind a unified API for both networking and file I/O; and (3) the

application must be structured to be non-blocking – restructuring requests so they can be batched, interleaving compute with I/O, and so on.

Rust provides a low-level `io_uring` library with zero-overhead abstractions that LiquidCache can use directly. What Rust lacks is a standardized asynchronous runtime. This is a double-edged sword: it leaves room for performance optimization, but it also fragments the ecosystem. Notably, the most widely used runtime, Tokio, does not support `io_uring` directly; all of its I/O is performed by a pool of blocking worker threads. Other runtimes, such as `smol`, do support `io_uring`, but adopting one would require rewriting every Tokio-specific I/O call against the new runtime’s API.

Restructuring the application to fit the `io_uring` model is usually the hardest part, often demanding substantial refactoring – a difficulty documented across several prior works. Fortunately, LiquidCache builds on DataFusion, a well-engineered query engine whose execution model already accommodates flexible I/O.

7.2.2 A runtime-agnostic interface

The cache does not get to choose its host’s runtime. A caching library that imposed `io_uring`, or any particular async runtime, on the systems embedding it would be unadoptable – it would trade one form of opacity for a worse form of coupling. LiquidCache therefore faces two constraints specific to its role as a library: (1) it must remain runtime-agnostic, imposing no particular asynchronous or I/O runtime on its users; and (2) it must not burden the user with anything beyond ordinary async functions.

Concretely, LiquidCache’s interface should be no more complex than `async fn get(key: &str) -> Result<Vec<u8>>` and `async fn insert(key: &str, value: &[u8]) -> Result<()>` – indistinguishable from the equivalent calls a user would write against Tokio. All of the `io_uring` machinery lives behind that boundary.

7.2.3 The I/O-thread model

The design that satisfies those constraints concentrates all I/O behind a single thread, so that the aggressive internals never surface in the interface. LiquidCache dedicates one thread to all I/O: it submits requests to the kernel and polls for their completion. To reduce contention inside the kernel, this thread maintains several `io_uring` instances. Worker tasks – those executing queries, for example – hand their I/O requests to the I/O thread over a

message-passing channel. We provide our own read and write primitives that transparently forward a request to the I/O thread and await its completion, so that application code sees only ordinary async functions.

Several alternative designs are possible. The first gives each worker its own `io_uring` instance and lets it submit to the kernel directly. This reduces contention on any single instance, but it multiplies the number of rings to allocate and manage, and it still requires a thread to poll for completions. The second moves completion polling into the asynchronous runtime itself, so that the scheduler polls for I/O completion just as it already does for network readiness via `epoll`. This is the cleaner design – it mirrors how networking is already handled – but Tokio does not support it today, and will not until a future major release. We chose the dedicated-thread model precisely because it delivers control over I/O while asking nothing of the host runtime: it works under any runtime, and it can be retired in favor of the cleaner scheduler-integrated design once the ecosystem catches up.

7.2.4 Reimplementing a slice of the filesystem

LiquidCache’s unit of caching is a single batch – typically 8192 rows, or a few kilobytes on disk. The naive approach stores each batch as its own file and leaves space management and lookup to the filesystem. This sufficed for our initial implementation, but as LiquidCache reached production we heard two recurring complaints from users: the scheme creates an enormous number of small files, which can quickly exhaust the system’s inode budget, and repeatedly opening and closing files is expensive. This is the same pattern as the page cache, one level down: the filesystem offers functionality LiquidCache does not need – permission control, file locking, directory hierarchy – at the price of excessive metadata and system-call overhead. A White-Box cache that already manages its own bytes is better off managing their on-disk home as well.

Newer versions of LiquidCache therefore store all batches in a single file, sharing one inode and one file descriptor, and manage space allocation and lookup themselves. Each batch is rounded up to a 4 KB boundary; to store a batch, LiquidCache allocates a contiguous 4 KB-aligned block and writes the batch into it. An in-memory index maps each batch ID to its disk offset and size. The file also reserves special pages that hold a write-ahead log of mutations – inserting an entry, deleting an entry, and so on. A deletion first appends a deletion record to the log and then removes the entry from the in-memory index, which is reconstructed by replaying the log when the file is opened.

Over time the file accumulates free space, leaving holes between live batches. LiquidCache runs garbage collection to (1) locate the holes, (2) copy live data from the tail into them, and (3) append log records noting the relocations. Collection is triggered once the ratio of deleted to total data exceeds a threshold (e.g., 10%). Because it only appends relocation records, garbage collection never blocks ongoing reads and writes and so introduces no latency spikes; it does, however, consume disk bandwidth, so a natural future optimization is to run it only when the disk is underutilized.

The log itself grows with every mutation, consuming space and lengthening startup. LiquidCache periodically performs a deep compaction – akin to snapshotting – that rewrites the log to retain only the latest state of each entry. Deep compaction stops the world and runs infrequently (e.g., once a day). Even so, startup remains $O(n)$ in the number of batches. A disk-resident index, loaded directly at startup, would remove this cost, but engineering one is non-trivial and tends to be slower than an in-memory index on the hot path.

The decision that closes the loop is to always use direct I/O. The kernel page cache works against everything this section set out to achieve: it complicates memory accounting, it delivers unpredictable performance (we have no control over its replacement decisions), and it double-caches data the cache already holds, wasting memory. With direct I/O, LiquidCache is the only significant cache in the query engine and can account for its own memory precisely against its budget – the byte-exact self-knowledge on which SqueezeCache’s content management depends, and the reason owning the I/O path was worth the trouble.

7.3 Inheriting the OS’s correctness burden

The previous section ended with the cache managing its own single-file store – its own write-ahead log, free-space allocator, and index. In doing so it took over a job the filesystem used to do, and with it the filesystem’s hardest obligation: never lose or corrupt data across a crash. This is the sharpest instance of the chapter’s theme – the more the cache takes over from the operating system, the more it must get right. The filesystem and page cache spent decades making crash consistency boring; a cache that bypasses them to gain control and accounting inherits that problem in full, on exactly the code paths – log replay, space allocation, garbage collection – where bugs surface only under a crash at the wrong microsecond and are nearly impossible to reproduce. The disk-management substrate is

thus both the part of the codebase most likely to silently corrupt data and the part where bugs are hardest to find by testing. We therefore verify it directly with Verus, an SMT-backed verifier for Rust.

Our verification targets three invariants. *Write-ahead logging* is verified to durably persist every committed insert and delete before any in-memory state is acknowledged, so that crash recovery can reconstruct the index from the log alone. *The in-memory index* is verified to track exactly the live entries in the data file, with no dangling pointers and no stranded extents. *Space management* is verified to never allocate the same byte range to two entries and to reclaim freed ranges correctly during garbage collection.

Unlike protocol verification, where the verified artefact is a model that an unverified implementation refines, we verify the Rust implementation directly. The verified module compiles and runs at full speed; the proof obligations are discharged at compile time and erased from the runtime binary. For the parts that can silently corrupt data, we found it cheaper to discharge that obligation once with a proof than to chase it forever with tests.

7.4 Staying equivalent to the format you replaced

A White-Box cache that controls its own representation is free to invent data layouts the rest of the stack has never seen. SqueezeCache’s squeezable string layout (Section 5.3.1) is the most aggressive such invention in this dissertation: it drops bytes, inlines only prefixes and fingerprints, and re-hydrates from disk on demand. But the engines reading from the cache were written against Arrow’s standard string layout and must not be able to tell the difference. Freedom over representation thus comes with an equivalence obligation: under every mix of inserts, queries, evictions, and re-hydrations, the squeezed layout must produce the same answers as the layout it replaced – precisely the property that unit tests, written one case at a time, systematically miss.

We therefore subject the layout to *differential fuzzing* against Arrow’s reference string array, treating the format we replaced as the specification the replacement must meet. A fuzzer generates random schemas, random sequences of operations (insert, filter, project, evict, hydrate), and random predicate workloads; both implementations process the same sequence; the outputs are diffed byte-wise. Discrepancies trigger automatic minimization to a minimal failing case, which is then either fixed or whitelisted as an expected difference (for example, our layout’s deliberate dropping of the variable-length buffer when a query

touches only the prefix). The fuzzer runs continuously against the main branch and has caught several behavioral divergences that the unit-test suite missed – most often around edge cases at the boundary between in-memory and disk-resident representations.

7.5 The metadata cost of fine granularity

Breaking opacity tends to push caching to a finer grain. LiquidCache holds individual batches rather than whole files; SqueezeCache tracks per-fragment squeeze state rather than whole columns; Bf-Tree manages variable-length mini-pages rather than fixed disk pages. Fine granularity is where much of the performance lives, but it carries a cost that is easy to overlook: it multiplies the number of entries the cache must index, and that index sits on the hot path of every lookup. A White-Box cache therefore needs an in-memory index that is both compact and fast, or the metadata overhead quietly eats the gains that finer granularity was meant to deliver.

A radix tree backs the in-memory index of the single-file store (Section 7.2) and the catalog of cached row groups in the LiquidCache server. We implemented a SIMD-accelerated, lock-free variant that supports concurrent insert, lookup, and range scan; the implementation borrows from ART [99] for prefix compression and adds vectorized key comparison for cache-friendly traversal at high fan-out. The radix tree is small enough – about 3,000 lines of Rust – to merit its own implementation rather than a third-party dependency, and tight enough that it does not show up as a hot spot in any profile we have measured.

7.6 Returning the generic core upstream

A surprising fraction of what made these caches fast turned out not to be about caching at all. To break opacity inside the cache, we repeatedly had to fix something in the shared substrate beneath it – the in-memory string representation, the Parquet decoder, the optimizer’s pushdown rules. Those fixes are not cache-specific; they are latent improvements to interfaces that many systems share. We contributed them upstream to the Apache Software Foundation, where they benefit any system built on DataFusion, Arrow, or Parquet – including engines that have no connection to this dissertation – and where a far wider set of workloads validates the technique than any single cache ever could.

Arrow StringView. The squeezable string layout’s split header/buffer design (Section 5.3.1) shares its roots with the StringView representation that we contributed to Apache Arrow [70, 71]. StringView is now the default in-memory string representation in Apache DataFusion and accelerates substring search, prefix comparison, and zero-copy slicing across the wider Arrow ecosystem.

Parquet decoding pipeline. The Parquet reader in Apache Arrow Rust was rewritten as part of LiquidCache’s effort to reduce decoding cost (Chapter 4). The new pipeline shares decompression buffers between the predicate-evaluation pass and the projection-materialization pass, eliminating one decompression step per filter. The change is now default behavior for any DataFusion or DuckDB-via-Arrow-Rust user reading Parquet.

Filter-pushdown infrastructure in DataFusion. The optimizer rules that allow filters and projections to be pushed into a custom TableProvider were extended to support the multi-layer pushdown that LiquidCache requires (filters into the cache server, residual filters above). The extension applies to any DataFusion TableProvider, not just LiquidCache’s.

In total, roughly 5,000 lines of code have been merged upstream across these three projects, with the corresponding pull-request history available in the Apache repositories on GitHub.

7.7 Lessons learned

The sections above each traced one obligation that White-Box caching created and the design choice that contained it. Stepping back from the individual systems, three lessons recur across all of them – lessons that we suspect generalize to any system that trades opacity for control.

More control means more to get right. Every performance win in this dissertation came from the cache taking over a decision that some lower layer used to make: the encoding, the eviction schedule, the on-disk layout, the I/O path. Each transfer of control is also a transfer of responsibility, and the responsibilities do not shrink just because the code that discharges them is now ours. A cache that manages its own disk inherits crash consistency (Section 7.3); a cache that invents its own layout inherits the obligation to behave like the layout it replaced (Section 7.4). These are precisely the obligations that the layers we bypassed had spent decades making boring, and they fail on code paths –

crash recovery, arbitrary operation interleavings – that ordinary testing covers poorly. The lesson is that ambition at the level of performance has to be matched by rigor at the level of correctness: formal verification and differential fuzzing were not optional polish but the price of admission for taking control safely, and we expect that price to rise, not fall, as White-Box techniques reach further down the stack.

Exhaust an interface before breaking it. It is tempting, once opacity is identified as the enemy, to break every abstraction in sight. That instinct is wrong. Most of the wins in this dissertation came not from new interfaces but from using existing ones more fully – pushing filters through DataFusion’s optimizer (Section 7.6), riding Parquet’s existing decode path, expressing all of our I/O as ordinary async functions (Section 7.2). An existing interface that almost does what you need is worth far more than a new one that does it exactly, because the existing interface comes with adopters, tooling, and years of hardening for free. When a new interface genuinely is required, the discipline is to make it *just* large enough to capture the decision being reclaimed and no larger: the runtime-agnostic I/O boundary (Section 7.2) reclaims control of the I/O path while asking nothing of the host runtime, and the five-crate split (Section 7.1) lets an adopter take one reclaimed decision without the others. The cost of a new interface is paid by everyone who has to adopt it, so it should be the smallest interface that still earns its keep.

Breaking an abstraction is easy; building one is hard. Anyone can reach through an abstraction to grab the performance hiding behind it – inline the bytes, skip the copy, special-case the hot path. The result is fast and brittle: it wins once, in one system, and resists every subsequent change. The harder and more valuable act is to build a *new* abstraction under which those optimizations become not a special case but the default – one that other systems can adopt without understanding how the speedup works. `StringView` is such an abstraction; so is the squeeze interface that lets an engine reclaim memory without knowing how the bytes are stored. White-Box caching, despite its name, is an attempt at the latter rather than the former: not a license to violate abstractions for speed, but a discipline for finding the few new abstractions under which speed and opacity stop being at odds. That is the harder path, and the one we believe is worth taking.

Chapter 8

Related Work

The three contributions of this dissertation – LiquidCache, SqueezeCache, and Bf-Tree – each sit at a different point in a rich landscape of caching, data representation, and storage-system research. This chapter surveys that landscape thematically, grouping prior work by the question it addresses rather than by which of our chapters cites it. Each section opens with a body of related work and closes by positioning the relevant contributions of this dissertation.

8.1 Caching architectures and policies

Caching is among the oldest topics in computer systems [14], and the literature on eviction policy alone spans decades. Classical algorithms – LRU, LRU-K [131], ARC [122], 2Q [82], CLOCK – balance recency and frequency to maximize hit rates. Recent simpler proposals match or exceed them: S3-FIFO [177] replaces LRU’s complexity with three FIFO queues; SIEVE [189] achieves competitive performance with a single FIFO and a moving cursor. A parallel line applies machine learning to eviction (GL-Cache [176], Baleen [174]), and recent cost-aware policies [28, 90] account for cloud pricing and consistency constraints.

Mechanism design has been similarly active. CacheLib [23] provides a production-grade, tunable framework deployed at scale; predecessor research caches such as Crystal [52] and federated caches such as Alluxio [4] serve cloud-native workloads. Elastic variants such as CompuCache [188] and Ditto [148] exploit spot capacity, while InfiniCache [171] builds cache state inside ephemeral serverless functions. Tiered-memory caches [73, 116, 129, 130, 141] span DRAM, CXL-attached memory, and NVMe to extend capacity beyond local DRAM. Production cloud warehouses – Snowflake [162, 170], Google’s Napa [3], Amazon Redshift [5, 137], BigQuery [123], Krypton [34], FoundationDB’s storage layer [192] – each pick a different point in this space.

The contributions of this dissertation are orthogonal to eviction policy and complementary to mechanism design. A White-Box cache can run any of the eviction policies above and deploy on any of these substrates; what changes is the *unit* on which a policy operates – an encoded Liquid fragment for LiquidCache, a squeezed batch for SqueezeCache, a variable-length mini-page for Bf-Tree.

8.2 Pushdown and semantic caching

Pushdown moves operators closer to the data they read. The basic form – predicate pushdown into the storage operator [158] – dates to the early relational systems; the cloud-native resurgence has pushed filters into specialized hardware (FPGAs [149], SmartSSDs [47], SmartNICs [88], DPUs [78]), into object-storage services (S3 Select [6], Azure Data Lake [125]), and – closest to this dissertation – into the cache layer. FlexPushdownDB [179, 180] and Fusion [117] explore *cooperative* pushdown that chooses between cache and compute dynamically based on the cache’s CPU budget; both treat cached bytes as opaque. LiquidCache attacks the decoding cost that makes such cooperative scheduling necessary in the first place: by transcoding into a query-friendly format, the cache evaluates predicates cheaply enough that pushdown is the right choice for most workloads, leaving a residual placement decision – when filters are non-selective – that reduces to a simple selectivity test rather than the CPU-budget-aware dynamic scheduling FlexPushdownDB and Fusion require.

Semantic caches cache something other than the raw data. Result caching stores past query outputs and serves only exact repeats; materialized views maintain pre-computed sub-queries chosen by the user or system. Predicate caching [142] records which row ranges have satisfied past predicates and accelerates subsequent scans. Time-series specializations such as STsCache [87, 105] and TSCache [112] exploit narrow regimes, while ReCache [18] and ChronoCache [62] mid-tier-cache analytical results to anticipate user behavior. All are precise about *what* is cached but rigid about *how much*: each entry is present in full or not at all. SqueezeCache generalizes this family by treating fidelity as a spectrum and the lineage that drives the squeeze decision as a per-column property of the workload.

8.3 Data representation and encoding

Analytical data has converged on a small set of public, columnar formats. Apache Parquet [169] and Apache ORC [10] dominate on disk; Apache Arrow [93] is the corresponding in-memory representation. Since Parquet’s 2013 release, a wave of encoding schemes has aimed to compress harder or decode faster: FSST [25] for strings, FastLanes [1] for SIMD-decodable bit-packing, ALP [2] for floating-point, with earlier streaming-time-series variants Chimp [109], Gorilla [134], and ELF [106]. Frame-of-reference and dictionary encoding are nearly universal building blocks; BtrBlocks [91] cascades these into a single per-column scheme.

A second wave proposes outright replacements for Parquet – Vortex [156], Nimble [124], LanceDB [95], and BtrBlocks itself – whose encodings are co-designed for fast decoding. The technical case is compelling, but Parquet has accumulated a decade of ecosystem support that none of these formats yet match [61, 89, 185], and even Parquet’s own evolution is incremental [9]. LiquidCache takes a different path: it adopts the same modern encodings *inside the cache*, leaving the on-disk file format untouched. Compression-aware execution – the principle that operators should evaluate against encoded data when they can – generalizes this stance to the engine itself; MorphStore [40] is its purest expression.

Beyond compression, lightweight per-block summaries accelerate pruning. Column imprints [155], Sieve [166], and instance-optimized layouts [46] capture per-block statistics; string fingerprints [157] target substring search in particular. SqueezeCache uses fingerprints as one squeeze target for strings, but the broader idea – that any lightweight summary structure can serve as a squeeze target once the cache knows when to ask for one – is what the chapter argues.

8.4 Resource disaggregation

Cloud-native architectures disaggregate compute from storage; object stores such as S3 [5, 13, 128] hold petabytes while compute scales elastically. Building on this success, researchers have pursued further disaggregation of memory and other resources. FaRM [49], LegoOS [147], AIFM [140], Pond [104], TPP [120], Redy [187], and Hao et al. [73] investigate disaggregated DRAM tiers built on RDMA or CXL. These approaches achieve

physical disaggregation at the cost of specialized hardware. LiquidCache and SqueezeCache pursue *logical* disaggregation in software: the cache server is an ordinary VM that runs an embedded query engine and requires no hardware beyond what any cloud provider already offers.

Serverless and ephemeral-compute models extend this disaggregation to the compute side. Bauplan [163, 164] and cloud-function-based analytics [24] show that analytical compute can be both ephemeral and cost-effective. InfiniCache [171] puts the cache itself into ephemeral functions, trading state durability for elastic capacity. Bf-Tree’s I/O substrate is designed to coexist with this regime: a runtime-agnostic, async-friendly disk layer that does not assume long-lived processes.

8.5 Larger-than-memory range indexes

The B-Tree [65] has been the dominant range index for larger-than-memory data for half a century, with countless variants. Bw-Tree [102, 173] and B ϵ -Tree [22, 38] chain delta records to batch updates; SplinterDB focuses on low read/write amplification. Leanstore [98] uses pointer swizzling for in-memory efficiency. Two-Tree [74, 193], Anti-Caching [45], Siberia [56], and TreeLine [182] overlay record caches on page caches to improve point lookups. Each enhancement targets one symptom of page-aligned caching, but none changes the underlying coupling between cache page and disk page; Bf-Tree addresses the coupling directly.

The LSM-Tree [132] is the other major larger-than-memory index family, used in RocksDB [48], LevelDB [64], BigTable [33], TiDB [79], and many others. Log-structured writes reduce write amplification at the cost of slower reads and expensive compaction; modern variants reduce compaction overhead (WiscKey [118] separates keys from values; WALTZ [96] leverages ZNS SSDs) or improve filtering (Chucky [43] replaces Bloom filters). Bf-Tree achieves write-amplification savings comparable to an LSM-Tree by batching updates in variable-length mini-page write buffers, without paying the LSM read-and-compact costs.

A third line of work targets the cache-fits regime. ART [99], Blink-hash [30], BP-tree [103], HydraList [121], and Pea hash [114] optimize in-memory indexes for points and ranges. FASTER [31, 32, 107] pushes hybrid log designs into hash-based larger-than-memory KV stores. Bf-Tree’s in-memory performance is informed by these systems; its

variable-length mini-page can be viewed as their generalization to the larger-than-memory regime where some of the index spills to disk.

8.6 Modern hardware and I/O

Modern NVMe SSDs have closed much of the latency and bandwidth gap with DRAM [68], prompting reconsideration of long-standing assumptions. 4 KB random writes can almost saturate an SSD’s bandwidth [182], making log-structured writes less attractive than they once were, and the conventional kernel-mediated I/O path is now a significant fraction of every read’s cost. Kernel-bypass paths via SPDK [181] and `io_uring` [17] eliminate context-switch and data-copy overheads; LiquidCache and Bf-Tree both build on `io_uring` through the shared I/O substrate of Chapter 7.

Persistent memory and other emerging hardware have spawned a family of index designs: BzTree [16], PLIN [190], NBTree [186], APEX [115], Halo [77], and Hamming-tree [85] target byte-addressable persistent memory, with comprehensive evaluations in [75] and [101]. Sherman [172] and Tiered Memory [74] propose systems for disaggregated and tiered memory. PIM-Tree [84] targets processing-in-memory. Transparent-compression SSDs offer yet another dimension: storage hardware that compresses internally can change the relevant trade-offs at the file system and index layers [135].

These directions are complementary to this dissertation. White-Box caching is a software architecture orthogonal to which substrate the cache runs on, and the mini-page abstraction in Bf-Tree could be deployed on top of any of these emerging substrates – a possibility we leave as future work.

Chapter 9

Conclusion

This dissertation began with the observation that caching, despite half a century of refinement, has remained a deliberately opaque abstraction. A cache accepts blobs, indexes them by key, scores them by recency or frequency, and evicts them as a unit; the cache does not know what is inside a blob, how it is encoded, which fields a query needs, or which bytes are actually hot. For decades that opacity was a virtue. This dissertation has argued – and the three system chapters have demonstrated empirically – that for modern data systems it has become the bottleneck, and that a different stance is both possible and practical.

9.1 Summary

The reframing this dissertation proposes is *White-Box caching*: a cache, in a modern data system, should be allowed to see inside the data it holds, understand the operations being performed against it, and choose its representation, content, and unit accordingly. The freedoms this opens up fall along three axes – representation, content, granularity – each corresponding to a kind of information a black-box cache is forced to discard. Chapter 3 developed the shared design vocabulary along which a White-Box cache exposes these freedoms: a richer ingest/query/reflect interface, three information-flow channels (schema metadata, pushdown protocol, access traces), and a four-state fidelity spectrum that replaces the binary keep-or-evict decision of conventional caches.

The three system chapters each instantiated one axis.

LiquidCache (Chapter 4) took up the *representation* axis. By transparently transcoding Parquet into an in-cache cascade of dictionary, FastLanes bit-packing, FSST, ALP, and frame-of-reference encodings – each layer independently decodable – the cache evaluates predicates against whichever encoded layer is most efficient for the predicate at hand. On ClickBench and TPC-H, *LiquidCache* reduced cache CPU by up to 10× compared to Parquet pushdown and network traffic by two orders of magnitude compared to file-serving caches,

while keeping memory comparable to compressed Parquet. The Liquid format lives only inside the cache, so the open lakehouse ecosystem continues to see Parquet, and several of the underlying techniques – StringView and a rebuilt Parquet decoding pipeline – have been upstreamed and are now defaults in Apache DataFusion, Arrow, and Parquet.

SqueezeCache (Chapter 5) took up the *content* axis. Even with LiquidCache’s adaptive encoding, profiling showed that up to 98% of cached bytes are never read by the queries that warmed them – even under Belady-optimal replacement. Squeezing replaces each cached entry with a compact, lossy subset that still answers the predicates the workload is running. Per-datatype squeezable layouts (prefix + fingerprint for strings, quantization/clamping for integers, semantic extraction for datetimes, shredding for JSON variants) supply the targets; *lineage pushdown* supplies the optimizer-to-cache communication that decides which target to retain per column; a multi-state policy navigates the resulting fidelity spectrum. On ClickBench and JSONBench, SqueezeCache improved cache hit ratios by up to 4×, reduced query latency by up to 22×, and cut I/O by up to 12× compared to an uncompressed cache (2×, 2×, and 8× respectively over a compression-only baseline). SqueezeCache is the default eviction mechanism of the open-source LiquidCache project.

Bf-Tree (Chapter 6) took up the *granularity* axis in the transactional regime. Where prior larger-than-memory range indexes either align cache pages with disk pages (B-Tree variants) or accept slower reads to amortize writes (LSM-Trees), Bf-Tree introduces a variable-length *mini-page* that holds a judiciously chosen subset of each disk page: a few hot records when point lookups dominate, a write buffer when updates dominate, or the full page when range scans warrant. A novel circular-log buffer pool with a copy-on-access second-chance region manages the mini-pages concurrently and at NVMe speeds. On a YCSB-like benchmark, Bf-Tree is 2× faster than RocksDB on range scans, 6× faster than a conventional B-Tree on writes, and 2× faster than both on point lookups – the only system among our baselines to perform well on all three operations simultaneously. Bf-Tree demonstrates that the White-Box philosophy is not confined to analytical caches: the same move, looking inside the cached unit, pays off in transactional KV indexes too.

Together, LiquidCache and SqueezeCache form a single open-source artefact of over fifty thousand lines of Rust [72], deployed in production by a small but real community of users (Bf-Tree is implemented separately), with substantial upstream contributions that have already changed defaults in widely deployed open-source analytical libraries.

9.2 Cross-cutting findings

Three patterns recur across the three systems and deserve to be named, since they are the parts of White-Box caching that generalize beyond any single contribution.

Lineage as a cache API. Each system needed a way for the engine to tell the cache what the workload would do with each column: filter expressions in LiquidCache, semantic extraction targets in SqueezeCache, access traces in Bf-Tree. A consistent pattern emerges: lightweight, declarative information attached to schemas or pushed alongside queries is enough. The cache does not need to be tightly coupled to the engine, and the engine does not need to be rewritten for the cache; a small, additive metadata channel suffices. This makes the design portable and means each new engine that adopts the cache pays a one-time integration cost measured in tens of lines.

Transparent transcoding as an evolution strategy. The three systems all change the format of data at rest in memory while leaving the format on disk – Parquet, the lakehouse stack, or the B-Tree’s page layout – untouched. The cache becomes the layer where new encodings, new fidelity choices, and new granularity decisions can be deployed at the pace of a software release rather than the pace of an ecosystem consensus. The open lakehouse stack has been locked into the 2013 Parquet encodings [61, 89, 185] for a decade because every alternative requires every reader to upgrade together; a White-Box cache breaks that coupling.

Fidelity as a spectrum, not a binary. Conventional caches expose two states per entry, full and absent, and treat eviction as a one-way valve. A White-Box cache exposes a continuum – full, compressed, lossy, evicted – and chooses among them per entry under explicit budget constraints. The continuum is the abstraction that allows the cache to behave intelligently when memory is the binding resource, which it increasingly is in modern data systems where DRAM cost rivals CPU cost [104].

9.3 Limitations

The honest limits of the work also deserve to be stated.

The non-pushdown regime. LiquidCache and SqueezeCache transmit filtered but uncompressed data over the network, which can exceed the compressed unfiltered data size when filters are non-selective. Adaptive pushdown that decides at runtime whether to push, using

cardinality estimates from the optimizer, would close this gap; FlexPushdownDB [180] and Fusion [117] explore related ideas in different regimes. We leave the systematic integration of cost-based pushdown with White-Box encodings as future work.

Already-well-compressed data. SqueezeCache’s partial-eviction step is only useful when the data is not already near-perfectly compressed. On benchmarks with simple categorical strings and narrow integer ranges (TPC-H, TPC-DS), the squeeze is unnecessary; the cache falls back to compressed entries with no overhead but also no incremental win. This is fundamental: a cache cannot squeeze information that is already absent from a tightly compressed representation.

Engine coupling at the optimizer. Although the cache APIs are deliberately additive, lineage pushdown requires an optimizer rule and a schema-metadata channel that the host engine must expose. For DataFusion this was a 10-line integration; for engines with less flexible optimizer surfaces it may be larger. A portable plan format such as Substrait [160] would reduce the cost further, and is the most promising path to engine-agnostic White-Box caching.

9.4 Future work

The three axes of White-Box caching are not exhaustive. At least four directions seem worth pursuing.

A fourth axis: consistency. This dissertation treats data as effectively immutable on cache-relevant timescales – a fair assumption for the analytical lakehouse, where files are immutable and appendable. Workloads with frequent updates, materialized views over changing data, or transactional consistency requirements would benefit from a fourth White-Box axis: the cache choosing *when* each cached representation is allowed to lag behind the underlying data. Bf-Tree already does this implicitly through its write-buffer mini-pages, but the analytical chapters do not. What a content-axis cache should do when the underlying column is partially overwritten – re-squeeze, hydrate, or invalidate – is an open question.

Multi-tenant scheduling of transcoding and squeezing. Background transcoding and squeezing consume CPU that could be used for queries. In multi-tenant deployments, deciding which tenant’s data to transcode, which queries to delay, and which cache budgets

to flex is a scheduling problem we have only touched. Recent work on cost-aware caching [28, 90] is a starting point.

Hardware substrates. White-Box caching is a software architecture and is orthogonal to the substrate the cache runs on. CXL-attached memory, persistent memory, processing-in-memory [84], and transparent-compression SSDs each offer dimensions a White-Box cache could exploit – for example, by squeezing *toward* a representation that fits naturally in the slower tier of a tiered memory deployment. The mini-page abstraction in particular seems well-suited to CXL-attached memory, where access cost is intermediate between DRAM and SSD.

Engine-agnostic deployment. Substrait, as noted, is the cleanest path to deploying the same White-Box cache under multiple engines (Spark, DuckDB, DataFusion). The lineage analysis is already implementable as a logical-plan rewrite; the protocol with the cache is already a small metadata channel. What remains is the engineering effort of building it once against a portable plan format rather than a single engine’s internal IR.

9.5 Closing remarks

The opening of this dissertation argued that disaggregation of compute, storage, and memory has stretched a fundamental tension: the further apart these tiers move, the more every byte costs to fetch, and the more redundantly the rest of the stack pays for that fetch. Caching has always been the discipline that bridges such gaps. For most of its history, it has done so by remaining deliberately ignorant of what it holds – a design choice that kept caches simple, general, and composable, and that worked for as long as the units being cached were close enough in shape to the units being consumed.

That alignment no longer holds. A modern analytical query consumes a fraction of a column extracted from a structured value buried inside a compressed page; a modern transactional index updates a record buried inside a 4 KB page on a device whose latency rivals memory. The cache that does not see inside its contents is the cache that pays, on every query and every update, for the gap between what is held and what is used.

The contribution of this dissertation is a stance more than a single technique: caching for modern data systems should be *White-Box* – the cache should see, choose, and adapt – and three concrete systems show what that stance buys. The numbers are large enough to be interesting: 10× on CPU, 4× on hit ratios, 22× on latency, 6× on writes. The deeper claim

is structural: that the next foundation for caching is not a new policy, a new replacement algorithm, or a new hardware tier, but a new abstraction in which the cache is given permission to look inside the data it holds. The chapters of this dissertation have shown how three concrete systems make that abstraction practical, deployable, and – through their upstream contributions and open-source artifacts – already in use.

Bibliography

- [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, and others. 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. Retrieved March 21, 2024 from <https://www.alluxio.io/>
- [5] Amazon Web Services. 2024. Amazon Redshift - Cloud Data Warehouse. Retrieved March 21, 2024 from <https://aws.amazon.com/redshift/>
- [6] Amazon Web Services. 2024. Querying data in place with Amazon S3 Select. Retrieved March 19, 2024 from <https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html>
- [7] Amazon Web Services. 2024. Amazon ElastiCache for Valkey and for Redis OSS. Retrieved from <https://aws.amazon.com/elasticache/redis/>
- [8] Amazon Web Services. 2024. Use S3 Select with Spark to improve query performance.
- [9] Apache Parquet. 2024. Page Index - Apache Parquet. Retrieved from <https://parquet.apache.org/docs/file-format/pageindex/>
- [10] Apache Software Foundation. 2013. Apache ORC: Optimized Row Columnar file format for big data.
- [11] Apache Software Foundation. 2025. Apache DataFusion Comet. Retrieved from <https://github.com/apache/datafusion-comet>
- [12] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, and others. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

- [13] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, 2021. 28.
- [14] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces* (1.00 ed.). Arpaci-Dusseau Books.
- [15] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2023. *Operating Systems: Three Easy Pieces* (1.10 ed.). Arpaci-Dusseau Books.
- [16] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.
- [17] Jens Axboe. 2019. Efficient IO with io_uring.
- [18] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. 2017. Recache: Reactive caching for fast analytics over heterogeneous data. (2017).
- [19] Adrian Garcia Badaracco and Andrew Lamb. 2025. Dynamic Filters: Passing Information Between Operators During Execution for 25x Faster Queries. Retrieved from <https://datafusion.apache.org/blog/2025/09/10/dynamic-filters/>
- [20] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, and others. 2022. Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 International Conference on Management of Data*, 2022. 2326–2339.
- [21] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.
- [22] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. 2015. An introduction to B-trees and write-optimization. *login; magazine* 40, 5 (2015).
- [23] 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)*, 2020. 753–768.
- [24] 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. <https://doi.org/10.1145/3589306>
- [25] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.

- [26] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, and others. 2021. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021. 836–850.
- [27] Boudewijn Braams. 2018. Predicate pushdown in parquet and Apache spark. *Ph. D. dissertation* (2018).
- [28] Zhichao Cao and others. 2022. Making Cache Monotonic and Consistent. *Proceedings of the VLDB Endowment* 16, 4 (2022), 891–904.
- [29] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020. 209–223.
- [30] Hokeun Cha, Xiangpeng Hao, Tianzheng Wang, Huanchen Zhang, Aditya Akella, and Xiangyao Yu. 2023. Blink-hash: An Adaptive Hybrid Index for In-Memory Time-Series Databases. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1235–1248.
- [31] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, 2018. 275–290.
- [32] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: an embedded concurrent key-value store for state management. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1930–1933.
- [33] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
- [34] 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. <https://doi.org/10.14778/3611540.3611545>
- [35] ClickHouse. 2022. ClickBench: A Benchmark for Analytical Databases.
- [36] ClickHouse. 2022. ClickBench: A Benchmark For Analytical Database Management Systems.

- [37] ClickHouse contributors. 2025. JSONBench: a Benchmark For Data Analytics On JSON.
- [38] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. {SplinterDB}: Closing the bandwidth gap for {NVMe}{Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020. 49–63.
- [39] 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*, 2022.
- [40] 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.
- [41] Databricks. 2024. Optimize performance with caching on Databricks. Retrieved March 21, 2024 from <https://docs.databricks.com/aws/en/optimizations/disk-cache>
- [42] Databricks. 2024. Amazon S3 Select.
- [43] Niv Dayan and Moshe Twitto. 2021. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*, 2021. 365–378.
- [44] Biplob Debnath, Sudipta Sengupta, Jin Li, David J Lilja, and David HC Du. 2011. BloomFlash: Bloom filter on flash-based storage. In *2011 31st International Conference on Distributed Computing Systems*, 2011. 635–644.
- [45] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1942–1953.
- [46] 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*, 2021. 418–431. <https://doi.org/10.1145/3448016.3457270>
- [47] 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*, 2013. 1221–1230.

- [48] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32.
- [49] 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)*, 2014. 401–414.
- [50] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, July 2019. 1–14. Retrieved from <https://www.flux.utah.edu/paper/duplyakin-atc19>
- [51] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, and others. 2019. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, 2019. 1–14.
- [52] 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.
- [53] 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*, 2021. 445–458.
- [54] 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.
- [55] Kira Duwe, Angelos Anadiotis, Andrew Lamb, Lucas Lersch, Boaz Leskes, Daniel Ritter, and Pınar Tözün. The Five-Minute Rule for the Cloud: Caching in Analytics Systems.
- [56] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. 2014. Trekking through siberia: Managing cold data in a memory-optimized database. *Proceedings of the VLDB Endowment* 7, 11 (2014), 931–942.
- [57] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the BSDcan conference, ottawa, canada*, 2006.
- [58] Facebook. 2021. RocksDB Bloom Filter.
- [59] Facebook. 2023. Block Cache.

- [60] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. 2022. Sqlite: past, present, and future. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3535–3547.
- [61] Mateusz Gienieczko, Maximilian Kuschewski, Thomas Neumann, Viktor Leis, and Jana Giceva. 2025. AnyBlox: A Framework for Self-Decoding Datasets. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4017–4031.
- [62] 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*, 2020. 2391–2406. <https://doi.org/10.1145/3318464.3380593>
- [63] Google and contributors. 2014. FlatBuffers: Memory Efficient Serialization Library.
- [64] Google. 2021. LevelDB - A fast key-value storage library written at Google.
- [65] Goetz Graefe and others. 2011. Modern B-tree techniques. *Foundations and Trends® in Databases* 3, 4 (2011), 203–402.
- [66] Brendan Gregg. 2016. The flame graph. *Communications of the ACM* 59, 6 (2016), 48–57.
- [67] GreptimeTeam. 2025. GreptimeDB. Retrieved from <https://github.com/GreptimeTeam/greptimedb>
- [68] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2090–2102.
- [69] Xiangpeng Hao and contributors. 2025. Improve ``variant_get" performance on a perfect shredding.
- [70] Xiangpeng Hao and Andrew Lamb. 2024. Using StringView / German Style Strings to Make Queries Faster: Part 1- Reading Parquet. Retrieved September 13, 2024 from <https://datafusion.apache.org/blog/2024/09/13/string-view-german-style-strings-part-1/>
- [71] Xiangpeng Hao and Andrew Lamb. 2024. Using StringView / German Style Strings to Make Queries Faster: Part 2 - String Operations. Retrieved September 13, 2024 from <https://datafusion.apache.org/blog/2024/09/13/string-view-german-style-strings-part-2/>
- [72] 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. <https://doi.org/10.14778/3773731.3773741>

- [73] 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.
- [74] Xiangpeng Hao, Xinjing Zhou, Xiangyao Yu, and Michael Stonebraker. 2024. Towards Buffer Management with Tiered Main Memory. *Proc. ACM Manag. Data* 2, 1 (February 2024), Article 31. <https://doi.org/10.1145/3639286>
- [75] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. 2022. Evaluating Persistent Memory Range Indexes: Part Two [Extended Version]. *arXiv preprint arXiv:2201.13047* (2022).
- [76] Gavin Henry. 2019. Howard chu on lightning memory-mapped database. *Ieee Software* 36, 06 (2019), 83–87.
- [77] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. 2022. Halo: A hybrid PMem-DRAM persistent hash index with fast recovery. In *Proceedings of the 2022 International Conference on Management of Data*, 2022. 1049–1063.
- [78] Jiasheng Hu, Philip A Bernstein, Jialin Li, and Qizhen Zhang. 2024. DPDPUs: Data Processing with DPUs. *arXiv preprint arXiv:2407.13658* (2024).
- [79] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, and others. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
- [80] InfluxData. 2025. InfluxDB. Retrieved from <https://github.com/influxdata/influxdb>
- [81] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009. 24–35.
- [82] Theodore Johnson and Dennis Shasha. 1994. X3: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th VLDB Conference*, 1994. 439–450.
- [83] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. Rust-Belt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.
- [84] Hongbo Kang, Yiwei Zhao, Guy E Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B Gibbons. 2023. PIM-tree: A Skew-resistant Index for Processing-in-Memory. In *Proceedings of the 2023 ACM Workshop on Highlights of Parallel Computing*, 2023. 13–14.
- [85] Saeed Kargar and Faisal Nawab. 2023. Hamming Tree: The Case for Energy-Aware Indexing for NVMs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

- [86] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, 2016. 1675–1687.
- [87] 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. <https://doi.org/10.14778/3746405.3746421>
- [88] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojić, and Gustavo Alonso. 2021. Farview: Disaggregated memory with operator off-loading for database engines. *arXiv preprint arXiv:2106.07102* (2021).
- [89] Laurens Kuiper. 2025. Query Engines: Gatekeepers of the Parquet File Format. Retrieved from <https://duckdb.org/2025/01/22/parquet-encodings.html>
- [90] Adarsh Kumar and others. 2025. Linear Elastic Caching via Ski Rental. In *CIDR*, 2025.
- [91] 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.
- [92] LakeSail. 2025. Sail. Retrieved from <https://github.com/lakehq/sail>
- [93] 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*, 2024. 5–17.
- [94] Andrew Lamb. 2024. Apache DataFusion is now the fastest single node engine for querying Apache Parquet files.
- [95] LanceDB. 2025. LanceDB. Retrieved from <https://github.com/lancedb/lancedb>
- [96] Jongsung Lee, Donguk Kim, and Jae W Lee. 2023. WALTZ: Leveraging Zone Append to Tighten the Tail Latency of LSM Tree on ZNS SSD. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2884–2896.
- [97] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free list sharding in action. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*, 2019. 244–265.
- [98] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018. 185–196.

- [99] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013. 38–49.
- [100] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, 2016. 1–8.
- [101] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment* 13, 4 (2019), 574–587.
- [102] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013. 302–313.
- [103] Amanda Li. 2023. BP-Tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees. Doctoral dissertation.
- [104] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, and others. 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*, 2023. 574–587.
- [105] J Li and others. 2025. STsCache: An Efficient Semantic Caching Scheme for Time-series Data Workloads Based on Hybrid Storage. *Proceedings of the VLDB Endowment* 18, (2025).
- [106] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-Based Lossless Floating-Point Compression. *Proc. VLDB Endow.* 16, 7 (March 2023), 1763–1776. <https://doi.org/10.14778/3587136.3587149>
- [107] Tianyu Li, Badrish Chandramouli, and Samuel Madden. 2022. Performant almost-latch-free data structures using epoch protection. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*, 2022. 1–10.
- [108] 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.
- [109] Panagiotis Liakos, Katia Papakonstantinou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3058–3070.

- [110] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. 2021. Decomposed bounded floats for fast compression and queries. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2586–2598.
- [111] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A deep dive into common open formats for analytical dbmss. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3044–3056.
- [112] 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. <https://doi.org/10.14778/3484224.3484225>
- [113] Zhicheng Liu and Jeffrey Heer. 2014. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2122–2131.
- [114] Zhuoxuan Liu and Shimin Chen. 2023. Pea Hash: A Performant Extendible Adaptive Hashing Index. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.
- [115] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A high-performance learned index on persistent memory. *arXiv preprint arXiv:2105.00683* (2021).
- [116] J Lu and others. 2025. DEX: Scalable Range Indexing on Disaggregated Memory. *Proceedings of the VLDB Endowment* 17, (2025), 2603–2616.
- [117] Jianan Lu, Ashwini Raina, Asaf Cidon, and Michael J Freedman. 2025. Fusion: An Analytics Object Store Optimized for Query Pushdown. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025. 540–556.
- [118] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.
- [119] Chen Luo and Michael J Carey. 2020. Breaking down memory walls: Adaptive memory management in LSM-based storage systems. *Proceedings of the VLDB Endowment* 14, 3 (2020), 241–254.
- [120] 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*, 2023. 742–755.

- [121] Ajit Mathew and Changwoo Min. 2020. HydraList: A scalable in-memory index using asynchronous updates and partial replication. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1332–1345.
- [122] 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)*, 2003.
- [123] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, and others. 2020. Dremel: A decade of interactive SQL analysis at web scale. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3461–3472.
- [124] Meta. 2024. Nimble: A New File Format for Storage of Large Columnar Datasets.
- [125] Microsoft. 2024. Azure Data Lake Storage query acceleration. Retrieved March 19, 2024 from <https://learn.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-query-acceleration>
- [126] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.
- [127] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Addison-Wesley New York.
- [128] 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.
- [129] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. 2024. HPCache: memory-efficient OLAP through proportional caching revisited. *The VLDB Journal* 33, 6 (2024), 1775–1791.
- [130] 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*, 2023.
- [131] 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.
- [132] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, (1996), 351–385.

- [133] 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.
- [134] 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.
- [135] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. 2022. Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, February 2022. USENIX Association, Santa Clara, CA, 69–82. Retrieved from <https://www.usenix.org/conference/fast22/presentation/qiao>
- [136] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*, 2019. 1981–1984.
- [137] 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.
- [138] Adrian Riedl, Philipp Fent, Maximilian Bandle, and Thomas Neumann. 2023. Exploiting Code Generation for Efficient LIKE Pattern Matching. In *VLDB Workshops*, 2023.
- [139] RocksDB Team. 2021. RocksDB Secondary Cache. Retrieved February 1, 2024 from <https://rocksdb.org/blog/2021/05/27/rocksdb-secondary-cache.html>
- [140] 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)*, 2020. 315–332.
- [141] U Sabarwal and others. 2023. HetCache: Synergising NVMe Storage and GPU Acceleration for Memory-Efficient Analytics. In *CIDR*, 2023.
- [142] 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*, 2024. 347–359.
- [143] 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.

- [144] Samsung Semiconductor. 2024. PM9A3 (MZQL2960HCJR-00A07) - Datacenter SSD. Retrieved from <https://semiconductor.samsung.com/us/ssd/datacenter-ssd/pm9a3/mzql2960hcjr-00a07/>
- [145] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012. 309–318.
- [146] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, 2016. 157.
- [147] 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)*, 2018. 69–87.
- [148] 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)*, 2023. 675–691. <https://doi.org/10.1145/3600006.3613144>
- [149] 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*, 2011. 385–386.
- [150] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. 2005. A Logic of File Systems. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, 2005.
- [151] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2004. Life or Death at Block-Level. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004. 379–394.
- [152] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. Database-Aware Semantically-Smart Storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, 2005.
- [153] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2004. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*, 2004. 15–30.
- [154] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2003. Semanti-

- cally-Smart Disk Systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003. 73–88.
- [155] Noah Slavitch. 2020. Workload-Aware Column Imprints. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020. 2865–2867. <https://doi.org/10.1145/3318464.3384411>
 - [156] SpiralDB. 2024. Vortex: An extensible, state-of-the-art columnar file format.
 - [157] Mihail Stoian, Johannes Thürauf, Andreas Zimmerer, Alexander van Renen, and Andreas Kipf. 2025. Instance-Optimized String Fingerprints. *arXiv preprint arXiv:2507.10391* (2025).
 - [158] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O’Neil, and others. 2018. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518.
 - [159] Zhaoyuan Su, Ammar Ahmed, Zirui Wang, Ali Anwar, and Yue Cheng. 2024. Everything You Always Wanted to Know About Storage Compressibility of Pre-Trained ML Models but Were Afraid to Ask. *arXiv preprint arXiv:2402.13429* (2024).
 - [160] subtrait-io. 2021. Subtrait: Cross-Language Serialization for Relational Algebra. Retrieved from <https://github.com/subtrait-io/subtrait>
 - [161] Yutian Sun, Tim Meehan, Rebecca Schlussek, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, and others. 2023. Presto: A decade of SQL analytics at Meta. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.
 - [162] Jan Vincent Szlang, Sebastian Bress, Sebastian Cattes, Jonathan Dees, Florian Funke, Max Heimel, Michel Oleyunik, 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.
 - [163] 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*, 2024. 31–36.
 - [164] Jacopo Tagliabue, Ryan Curtin, and Ciro Greco. 2024. FaaS and Furious: abstractions and differential caching for efficient data pre-processing. Retrieved from <https://arxiv.org/abs/2411.08203>
 - [165] Tonbo.io. 2025. Tonbo. Retrieved from <https://github.com/tonbo-io/tonbo>
 - [166] 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. <https://doi.org/10.14778/3611479.3611520>
- [167] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017. 1041–1052.
 - [168] 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*, 2018. 1–6.
 - [169] Deepak Vohra and Deepak Vohra. 2016. Apache parquet. *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools* (2016), 325–335.
 - [170] 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)*, 2020. 449–462.
 - [171] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupperecht, 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)*, 2020. 267–281.
 - [172] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A write-optimized distributed b+ tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*, 2022. 1033–1048.
 - [173] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, 2018. 473–488.
 - [174] 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)*, 2024. 347–371.
 - [175] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. *ACM Transactions on Storage (TOS)* 13, 3 (2017), 1–26.

- [176] 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)*, 2023. 115–130.
- [177] 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*, 2023. 130–149.
- [178] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. 2019. Reducing garbage collection overhead in {SSD} based on workload prediction. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.
- [179] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulmaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid push-down and caching in a cloud DBMS. *Proceedings of the VLDB Endowment* 14, 11 (2021).
- [180] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulmaga, and Michael Stonebraker. 2024. FlexpushdownDB: rethinking computation pushdown for cloud OLAP DBMSs. *The VLDB Journal* 33, 5 (2024), 1643–1670.
- [181] Ziyue Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyun Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (Cloud-Com)*, 2017. 154–161.
- [182] Geoffrey X Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. Treeline: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment* 16, 1 (2022), 99–112.
- [183] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, 2012. 15–28.
- [184] 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.
- [185] Xinyu Zeng, Ruijun Meng, Martin Prammer, Wes McKinney, Jignesh M Patel, Andrew Pavlo, and Huanchen Zhang. 2025. F3: The Open-Source Data File Format for the Future. *Proceedings of the ACM on Management of Data* 3, 4 (2025), 1–27.

- [186] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: a Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1187–1200.
- [187] Qizhen Zhang, Philip A Bernstein, Daniel S Berger, and Badrish Chandramouli. 2021. Redy: Remote dynamic memory cache. *arXiv preprint arXiv:2112.12946* (2021).
- [188] 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*, 2022.
- [189] 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)*, 2024. 1229–1246.
- [190] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, and others. 2022. Plin: a persistent learned index for non-volatile memory with high performance and instant recovery. *Proceedings of the VLDB Endowment* 16, 2 (2022), 243–255.
- [191] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. 2014. Pipelined compaction for the LSM-tree. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014. 777–786.
- [192] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, and others. 2023. FoundationDB: A Distributed Key-Value Store. *Communications of the ACM* 66, 6 (2023), 97–105.
- [193] 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.