HPCache: Memory-Efficient OLAP Through Proportional Caching

Hamish Nicholson
hamish.nicholson@epfl.ch
EPFL
Lausanne, Switzerland

Periklis Chrysogelos
periklis.chrysogelos@epfl.ch
EPFL
Lausanne, Switzerland

Anastasia Ailamaki
anastasia.ailamaki@epfl.ch
EPFL, RAW Labs SA
Lausanne, Switzerland

ABSTRACT

Analytical engines rely on in-memory caching to avoid disk accesses and provide timely responses by keeping the most frequently accessed data in memory. Purely frequency- & time-based caching decisions, however, are a proxy of the expected query execution speedup only when disk accesses are significantly slower than in-memory query processing. On the other hand, fast storage offers loading times that approach or even outperform fully in-memory query execution response times, rendering purely frequency-based statistics incapable of capturing impact of a caching decision on query execution. For example, caching the input of a frequent query that spends most of its time processing joins is less beneficial than caching a page for a slightly less frequent but scan-heavy query. As a result, existing caching policies waste valuable memory space to cache input data that offer little-to-no acceleration for analytics.

This paper proposes HPCache, a buffer management policy that enables fast analytics on high-bandwidth storage by efficiently using the available in-memory space. HPCache caches data based on their speedup potential instead of relying on frequency-based statistics. We show that, with fast storage, the benefit of in-memory caching varies significantly across queries; therefore, we quantify the efficiency of caching decisions and formulate an optimization problem. We implement HPCache in Proteus and show that i) estimating speedup potential improves memory space utilization, and ii) simple runtime statistics suffice to infer speedup expectations. We show that HPCache achieves up to 12% faster query execution over state-of-the-art caching policies, or 75% less in-memory cache footprint without deteriorating query performance. Overall, HPCache enables efficient use of the in-memory space for input caching in the presence of fast storage, without any requirement for workload predictions.

CCS CONCEPTS

• **Information systems → Online analytical processing engines**; *Data scans.*

KEYWORDS

caching, NVMe, storage, buffer management, analytics

DaMoN'22, June 13, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9378-2/22/06…$15.00
https://doi.org/10.1145/3533737.3535100

ACM Reference Format:
Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. HPCache: Memory-Efficient OLAP Through Proportional Caching. In *Data Management on New Hardware (DaMoN'22), June 13, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3533737.3535100

1 INTRODUCTION

Improvements in CPU and DRAM efficiency allow in-memory query processing in analytical engines, placing frequently accessed datasets in-memory [37] and avoiding slow disk accesses. However, CPU and DRAM improvement rates have slowed in recent years, while advances in flash storage have enabled increased persistent-storage bandwidth [16, 23, 24, 32]. As a result, storing the working set in memory is no longer always advantageous. For example, when a query is CPU or memory latency bound it has lower throughput than storage bandwidth, so storing the query's entire input in memory is wasteful because it is possible to achieve the same execution time with the input located on disk. Instead, it would be more beneficial to use the same memory for another query.

For decades, database designs were based on the assumption that disk IO is the bottleneck in execution times and relied on in-memory caching, in the buffer pool, to bypass it. There are two lines of work that improve buffer pool performance for analytics: i) improving the efficiency of accessing buffer pool pages [22, 26], and ii) improving the probability that frequently used pages remain in memory [9, 10, 20, 27, 30, 31, 34]. In the first line of work, fast access to the buffer pool accelerates storage [2, 22, 26] and reduces the buffer pool overhead but relies on the effectiveness of the buffer pool policy to accelerate query execution. In the second line of work, frequency-based eviction policies improve the cache hit rate. High-bandwidth storage, like multiple NVMes per machine, however, allows for data loading times that are competitive to in-memory query processing (Figure 2). Thus, improving the hit rate no longer implies faster analytics, and as a result, the available in-memory space is underutilized, slowing down query execution.

In this paper, we propose HPCache, an eviction policy and tuning agent that optimizes the caching efficiency of the buffer pool for analytical workloads on high-bandwidth storage. We show that i) caching pages that have the same access frequency can yield significantly variable query acceleration results, and ii) efficient cache use should aim for partial column caching to avoid diminishing returns. HPCache is a buffer management policy that considers both the query frequency and the acceleration impact of any caching decision to derive a memory-efficient caching decision. Towards that end, HPCache i) examines the query execution to understand caching benefits, and ii) automatically tunes the caching priority and in-memory per-column space budget based on past execution

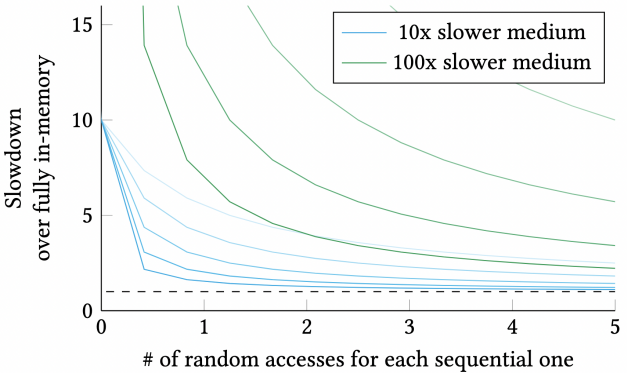

Figure 1: Conceptual slowdown for offloading sequential accesses to a 10x (blue) or a 100x (green) slower medium, over not offloading. The color shadow shows the penalty of a in-memory random access over a sequential access: 1x for the lighter, 16x for the darker one.

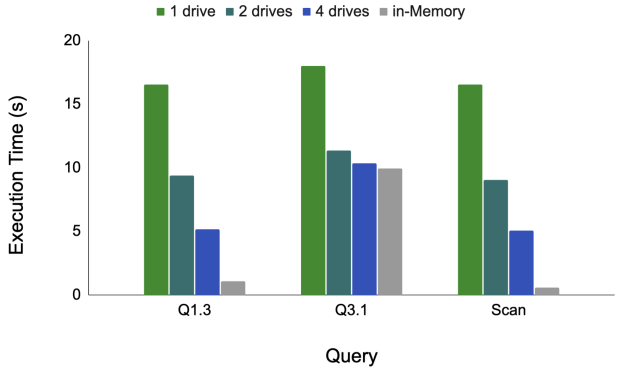

Figure 2: Run time for a varying number of NVMe drives.

behavior. Overall, HPCache improves the efficiency of in-memory data caching for analytics, allowing faster query execution for a given memory budget.

In summary, HPCache makes the following **contributions**:

- We show that in the presence of high bandwidth storage, frequency-based in-memory caching policies cache inputs that provide little query acceleration (Section 2).
- We propose HPCache, a policy that enables efficient memory utilization by considering the expected query acceleration of different caching decisions (Section 3). To avoid unreliable predictions, HPCache continuously tunes the caching policy based on run-time statistics (Section 4).
- When compared to state-of-the-art eviction policies, HPCache's improved caching efficiency allows for up to 12% speedup with the same memory budget, and up to 75% memory footprint reduction (Section 5).

Overall, HPCache enables analytical query processing to efficiently use the available in-memory space when accessing larger-than-memory datasets. Thus, it allows faster analytical response times for the same memory budget, or decreased memory footprint without execution time degradation.

2 IN-MEMORY CACHING AND HIGH-BANDWIDTH DISKS

Directly attached NVMe arrays have enough bandwidth to invalidate the general rule-of-thumb that scanning persistent data is always slower than in-memory execution. The rest of this section shows how high read bandwidth affects the execution speedups achieved by in-memory caching and how frequency-based caching policies can result in ineffective eviction decisions. Finally, we quantify the relative value of caching different data.

Fast storage. While disks were once considered slow, recent advances in flash technology have resulted in servers having comparable disk read bandwidth and in-memory bandwidth. Recent NVMes can sustain GBps of read bandwidth, e.g., an Intel D7-P5600 achieves 7 GBps. Furthermore, CPUs' support for 100s of PCIe 4.0 lanes per socket [13] allows multiple NVMes on the same server to achieve 10s-100s GBps of aggregated read bandwidth – just a single order of magnitude lower than the CPU memory bandwidth.

Quantifying "fast". Still, an order of magnitude more bandwidth is significant. Thus, analytical engines rely on CPU memory for a wide range of operations: except for higher read bandwidth, CPU memory also provides better random IO performance and lower latency than NVMe arrays. As a result, CPU memory has been a crucial element in enabling efficient in-memory joins [3, 4, 6] and analytics in general [7]. Thus, for many queries, reading the input data is only a small portion of the memory operations and only dominant for scan-heavy and few-small-join queries [33].

Offloading random or latency-sensitive accesses to disk incurs a significant overhead. However, offloading sequential data accesses can have a minimal impact on response times: random and latency-sensitive accesses cost more than sequential ones; thus, they quickly dominate the query execution time. As such, when query complexity increases, e.g., due to additional random accesses, the high overhead of random & latency-sensitive operations boosts their impact on query execution time, reducing the impact of sequential accesses. The shrinking of the relative contribution of sequential accesses to the total execution time as query complexity increases, combined with the high disk bandwidth, allows for pushing sequential scans to the disk for a minor penalty – a reverse application of Amdahl's law. Instead, the analytical engine can use the saved space for simpler or more selective, sequentially-scan-heavy queries.

However, the sensitivity of deciding which sequential scans to offload to disk is a function of the disk read bandwidth: for high read bandwidth, the slowdown varies significantly with relative costs. As an example, Figure 1 shows the expected overall slowdown when pushing a sequential read to two different storage media. For a given ratio of random/sequential accesses (x-axis), the slowdown for the fast (blue) medium varies significantly: from near 1 (no slowdown) to half an order of magnitude for the majority of the plotted range. In contrast, the slowdown observed for the slower (green) medium is already significantly higher for the same range.

Thus, given an acceptable-slowdown threshold, the faster medium allows simpler queries to run with offloaded data compared to the slower medium. Overall, the observed slowdowns for on-disk execution become increasingly sensitive to query complexity.

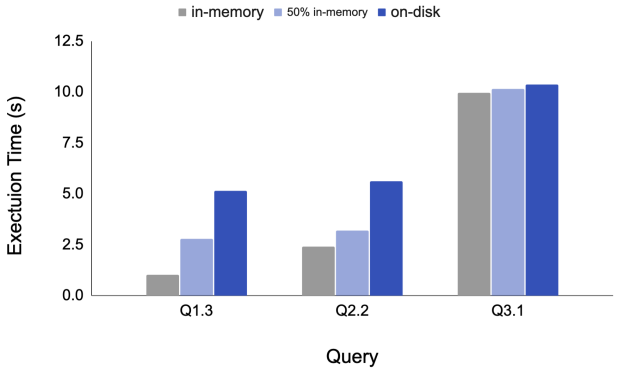

Figure 3: Execution time for three different data placements.

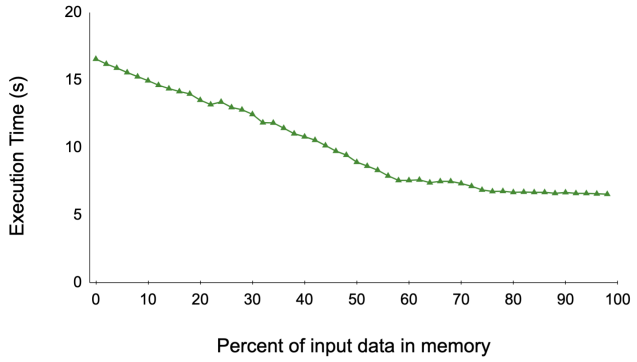

Figure 4: Execution time for Q2.1 with increasing proportions of input data cached in memory (one NVMe)

In-memory analytics vs. data loading: a race. Contrary to common belief, avoiding disk accesses no longer significantly reduces analytical query response time. In some cases, in-memory query processing can even be slower than loading input data from storage. Similar to the aforementioned sequential vs. non-sequential case, the slowdown due to accessing on-disk data depends on the query. Figure 2 shows the in-memory execution time for two Star Schema Benchmark (SSB) (Q1.3 and Q3.1) queries, the execution time for the same queries when the data are loaded from storage, and the time to load 96GB of binary data in memory from a variable number of NVMes. Both queries access 96 GB of input data, but Q1.3 has one cache-resident join while Q3.1 has four higher cardinality joins. With the input partitioned across four NVMes, Q1.3 and Q3.1 are 1% and 2x slower than scanning the data. Furthermore, Q1.3 sees a 4.8x slowdown when the input is not in memory. Caching Q3.1's input data has minimal speedup potential: the query, either way, spends most of its time processing the joins. Overall, whether in-memory caching of the input data will reduce query response time or not is query-dependant, even for simple queries.

Memory efficiency of caching. While avoiding disk accesses does not harm single-query execution, it leads to inefficient memory use when considering multiple queries. With data loading times comparable to execution times for some queries, caching input data for a query that spends most of its time on non-input operations can result in wasting memory that could be used to accelerate another query. Further, it contradicts the prior wisdom of caching the most frequently accessed data. The most common caching heuristics for analytics involve prioritizing data that has not been used for a long time (LRU [36]), that was recently consumed (MRU [10]), or a combination thereof and second-chance approaches [20]. However, even if Q3.1 of the previous example were executed significantly more frequently than Q1.3, caching its input would provide little benefit: caching reduces disk IO, but as shown above, Q3.1's execution time will improve very little. In contrast, caching the inputs of Q1.3 will accelerate each execution of Q1.3. Thus, treating all IO savings equally results in suboptimal query response times.

3 HPCACHE: HYBRID PROPORTIONAL CACHING

Based on the above observations, we propose Hybrid Proportional Caching (HPCache), a new data placement strategy that makes caching decisions based on the expected query response time reductions. Rather than aiming to reduce disk accesses, HPCache builds on three key principles to efficiently use the in-memory space: i) not all inputs provide the same query acceleration, ii) pages should be prioritized based on the expected impact on query execution time, iii) optimal caching decisions should aim for the sweet spot of matching query execution times with loading the remaining data. The rest of this section outlines the above key principles and their effect on in-memory caching decisions.

Not All Bytes Are the Same. Some queries are more sensitive to the location of their input data than others. Multiple factors affect the query response time, including the query access patterns, the data placement of both intermediate and input data, and the ability of the execution strategy to effectively use the available hardware resources, e.g., through prefetching, vectorization, and cache-awareness. Considering the query execution time, we can roughly classify queries into two extreme categories: input-access-sensitive queries whose execution time is highly determined by the bandwidth available to access input data, and processing-sensitive queries that spend most of their time accessing intermediate structures like join hash tables. As a result, Caching different inputs in memory provides different response time gains. *Thus, we use the expected benefit (Section 4.1) when deciding between alternative caching decisions to use the in-memory space efficiently.*

Impact- & Frequency-based caching. Each query's caching efficiency depends on its input-access-sensitivity: a highly input-access sensitive query will observe a higher (relative) speedup than a processing-sensitive query when its inputs are cached in memory.

Figure 3 shows the execution time for three analytical queries when using three data placement methods: fully in-memory, fully on-disk, and a hybrid configuration with half the input data in-memory and half on-disk. Both Q1.3 and Q2.2 are input-access-sensitive queries, and thus, their execution time is halved when moving half of their inputs in memory. However, when moving to fully in-memory, the two queries have different execution times,

despite their equal input size due to their differences in query processing. Q3.1 is a processing-sensitive query, so it executes in nearly the same amount of time regardless of the location of its inputs. Equally allocating memory to cache the inputs of these three queries would result in a slower speed up than preferentially caching the inputs of the input-access-sensitive queries.

However, per-query caching efficiency is not the only metric that matters when selecting which query inputs to cache: query frequency can also boost or reduce the overall efficiency. In a given workload, if the same or similar queries appear many times, then there is a multiplicative benefit to accelerating the repetitive query by caching its inputs.

To optimize the caching efficiency, HPCache combines the traditional frequency-based approaches with impact-based query acceleration expectations to achieve the best of both worlds. Specifically, it weights queries and input columns based on their occurrence frequency but calculates the overall expected execution time instead of purely optimizing the number of disk accesses. *We model query execution as a flow (Section 4.2) to approximate how different data placements would affect the total execution time for a sequence of queries and select the best placement given a memory budget.*

Partial input caching. Caching entire columns is wasteful: even in the simplified case of a single query, caching more than a specific fraction of a column provides minimal additional query speedup. Figure 4 shows how the execution time of SSB SF1000 Q2.1 improves as we increase the percentage of in-memory input data while the rest reside on an NVMe. Initially, the execution is dominated by the data loading time and thus improves linearly as more data are found in the in-memory cache. However, despite using more memory, the query execution time sees almost no reduction after a specific point. From that point up to having fully in-memory inputs, query execution is dominated by the actual processing. Thus, additional caching has no benefit: it would be preferable to use it for another query. HPCache limits the number of pages that are cached for each column to avoid diminishing returns from caching unnecessarily high portions of a specific column. *The limit is determined in a per-column base, and it's proportional to the expected impact of caching the corresponding column (Section 4.3).*

4 TUNING & MONITORING

To materialize HPCache, we provide a model that captures per-input caching impact (Section 4.1) and use it to provide a memory-efficient caching configuration (Section 4.2). Finally, we show how HPCache continuously adapts its caching configuration based on updated estimates and caching configurations (Section 4.3).

4.1 Impact modeling

We model the impact following a two step procedure: i) model the impact of caching the inputs of a specific query, and ii) combine impacts from multiple queries to determine the overall impact for caching a column.

The benefit of caching a query input. To model the impact of caching a specific query input, we model the execution time for the pipeline [25] consuming the corresponding input: by definition, caching the input will only change the performance of that pipeline. Any other pipeline that joins with the current one will use a newly materialized data structure produced by the modeled one. Further, the pipeline operates like a pipe as it consumes its input at a specific rate. This rate is limited either by input access bandwidth, or the pipeline's maximum throughput. Thus, if we have the size of the pipelines inputs (B), the storage bandwidth (S_{bw}), the memory bandwidth (M_{bw}), the proportion of inputs in memory (x), as well as the pipelines maximum throughput (P_{bw}), then we can approximate the execution time for partially cached inputs as

$$T_{pip,x\%} = \max\left(\frac{(1-x)*B}{S_{bw}}, \frac{x*B}{M_{bw}}, \frac{B}{P_{bw}}\right) \quad (1)$$

This results in a line similar to Figure 4 as the proportion of inputs in memory (x[1]), varies from 0% to 100%. The query's execution time is initially bottle necked by storage IO (the left side of the max) until the query execution time reaches in-memory processing speeds (where all terms of the max are equal). At this point, caching any additional inputs in memory does not improve the execution time (the right side of the max). We calculate P_{bw} using knowledge from previous query executions and show in Section 4.3 how we relax this requirement. To calculate the reduction of execution time from one input caching rate ($x\%$) versus another ($y\%$), it is sufficient to subtract the $T_{pip,x\%}$ from $T_{pip,y\%}$.

The benefit of a byte. Some columns may be used by multiple queries. To compute the caching impact of a column, we need to aggregate the execution speed-up it will allow across multiple queries. However, as the column participates in multiple pipelines, even if it is fully in-memory it may have to wait for other columns of the same pipeline to be loaded and vice-versa. Thus, to allow estimating the impact, we rely on a finer-grained granularity that builds the end-to-end execution time model based on per-column-per-pipeline time estimates. To build the time estimates we need to subdivide the time estimate of the previous paragraph to column granularity. We approximate this division by splitting $T_{pip,x\%}$ across the columns based on their relative sizes.

4.2 A Balanced Model

To provide a memory efficient caching configuration we model the expected execution time and the expected memory budget and formulate two optimizations problems – one to optimize for the execution time given a memory footprint, and one to optimize the memory footprint given a slowdown budget. Due to space constraints, our description focuses on the former, but the same principles apply to the latter.

Modeling as a flow. To decouple modeling from hyperparameter tuning, such as retrieving $T_{pip,mem}$ and whether future or past queries are available, we generalize the problem formulations and model execution of L queries, which can be either future or past ones – and we optimize execution across these L queries.

To calculate the total execution time (T) over these L queries, we split the queries into pipelines and for each pipeline, we sum the time for the various participating columns. The summation across pipelines is supported by the fact that pipelines execute one after the other, while the summation across columns is valid because the accredited per-column times already split the execution time based on column size. Similarly, based on the caching ratios, we can

[1]We use x both as a percentage, in the form 70%, in subscript, and as a proportion, in the form 0.7, within equations

compute the in-memory space $C(x\%)$ of having the corresponding proportions cached in-memory.

For example, for two queries Q_A and Q_B, that have a single pipeline each and they touch columns $\{i, j\}$ and $\{j, k\}$, respectively, we approximate the total execution time as:

$$T_{x_{i,j,k}\%} = max\left(T^{i}_{pip_A, x_i\%}, T^{j}_{pip_A, x_j\%}\right) + max\left(T^{j}_{pip_B, x_j\%}, T^{k}_{pip_B, x_k\%}\right) \quad (2)$$

by making the approximation that:

$$T_{pip_A, x_{i,j}\%} = max\left(T^{i}_{pip_A, x_i\%}, T^{j}_{pip_A, x_j\%}\right) \quad (3)$$

because a pipeline's throughput will be bottle necked by the column with the lowest available access bandwidth. For example, if the query is IO bound and column i is on disk while j is in memory, the query will process only as fast as column j can be read from disk. Note that shared columns share cache proportions (e.g., $x_b\%$). Also, while the methodology could model cold caches, this formulation is optimized for query repetitions and thus ignores cold cache cases such as column b being fully on disk before Q_A, but brought into the cache before Q_B starts.

The above formulation of $T_{x\%}$ allows for capturing i) the impact of different configurations, ii) the shared caching proportions across queries, and iii) the frequency-based importance of each query, as queries that repeat multiple times in the L window will appear multiple times in the summation.

Static tuning & optimization. To provide an efficient configuration, HPCache sets up and solves a minimization problem that finds the $x\%$ that minimizes $T_{x\%}$, subject to $C(x\%) < B$, where B is the memory budget. We solve the optimization problem using a convex optimizer. The optimal configuration $x\%$ is then provided to the buffer pool. The buffer pool maintains a list of data pages for each column. Whenever the allocation of a column is full and it requires space for a new data page, it evicts a page from the same column. The execution engine requests data pages from the buffer pool as it would do for any other caching policy.

4.3 Continuous Tuning

HPCache tracks query execution as it evolves, continuously adapting the caching configuration by taking into consideration the recent query and performance history. Furthermore, HPCache continuously monitors the query execution to tune its estimates of the inferred P_{bw} for the currently executing pipeline.

Looking back. The above model optimizes the execution across L queries. This allows HPCache to optimize for future as well as past queries. In the default case, though, HPCache uses $L-1$ past queries combined with the current query to decide a new caching configuration. For the past queries, it estimates P_{bw} based on the pipeline execution times. However, HPCache does not fix the caching ratios of the previous queries. Instead, it allows re-optimizing and reducing or increasing them based on incoming queries – essentially it optimizes them under the assumption that a similar pattern as the last $L-1$ queries will repeat next.

Monitoring & inspection. In general, it is hard to predict P_{bw} reliably for each pipeline. Instead, HPCache estimates P_{bw} during query execution. In each pipeline invocation, the pipelines of interest receive a set of input blocks corresponding with one block per accessed attribute. The execution engine reports to HPCache the execution time for each pipeline invocation *after* the input pages are present in memory. We can then calculate an estimate of P_{bw} from the pages per second that the pipeline is processing and the known input size of the pipeline. Since many instances of the same pipeline run in parallel, we assume that pipeline processing time is equally distributed across threads.

HPCache uses the estimate of P_{bw} along with the execution times of the previous queries as inputs for the optimization problem formulated in Section 4.2. HPCache re-solves the problem intermittently in a background thread using the latest statistics, both refining its estimated P_{bw} for the current pipeline and the overall optimal column proportions. The new optimal configuration is continuously supplied to the buffer pool as a maximum number of pages to cache for each column. If some columns exceed their maximum allocations in the new configuration, then the buffer manager moves pages from the over-represented column into a global free list. Free list pages are evicted first to make room for new pages. While pages are in the free list, they are still available to threads requesting them until they are evicted.

5 EVALUATION

We implement HPCache in Proteus [11, 21, 29], a parallel in-memory analytical engine that uses code generation. For all IO, we use the `read` system call with files opened using the `O_DIRECT` flag to bypass the operating system buffer cache. We use 2 MiB pages, which allows us to use 2 MiB hugepages. We use the CVXPY [1, 14] library to solve the optimization problem outlined in Section 4.3.

Methodology. The experiments run on a dual socket server with 376GB DRAM. Each socket is a 12-core Intel Xeon Gold 5118 CPU at 2.3GHz. The CPUs are connected by 2 UPI links. The server is equipped with four NVMe drives, 3 Samsung MZPLL1T6HAJQ-00005 and 1 Dell Express Flash PM1725a, measuring 4970MiB/s and 5780MiB/s read bandwidth, respecively. Single-drive experiments use the single Dell drive. We use the Star Schema Benchmark [28], with scale factor 1000, resulting in 24 GB per binary lineorder column. All queries use 48 threads.

5.1 Micro Benchmarks

Proportional Caching. Figure 5 evaluates the relationship between storage bandwidth and the amount of input data that needs to be cached in memory to achieve peak query performance. We run SSB Q2.1 and vary the percentage of the input columns that are memory resident before the query begins from 0% to 100%. When data is fully loaded in memory, this query has a throughput of 14.7 GB/s, which is between the bandwidth of using 2 and 4 drives on our server. The 1 and 2 drive experiments execute in 16.5 and 9.8 seconds, respectively, as the available storage bandwidth is the initial bottleneck. There is a linear decrease in execution time as the proportion of data in memory increases in the 1 and 2 drive case. In the 4 drive case, there is never a benefit to caching data in memory as the query is never bottlenecked by storage bandwidth. As the storage bandwidth increases, the proportion of data that needs to be in memory before achieving the minimum execution time decreases. Further, once storage bandwidth exceeds query throughput, there is no benefit to caching any data in memory.

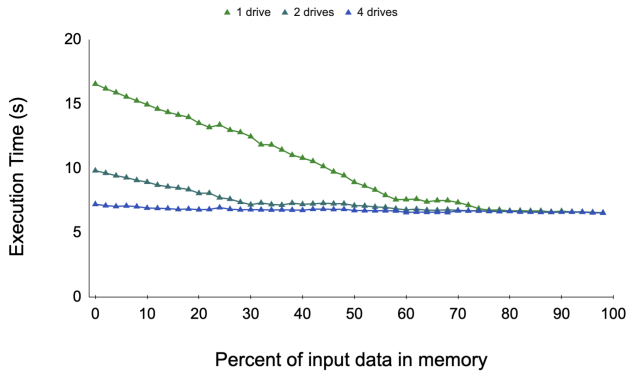

Figure 5: Execution time of SSB Q2.1 as the percentage of the queries inputs in memory range from 0% to 100%. 96 GB working set.

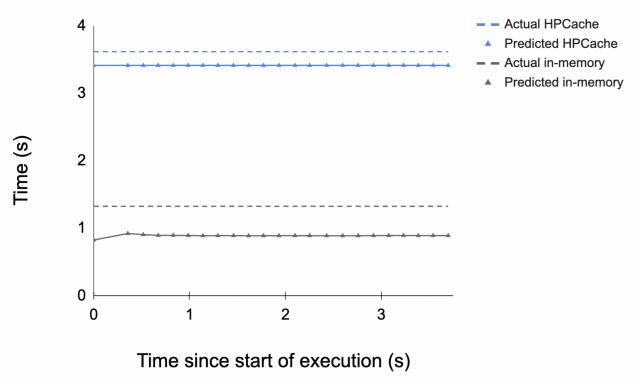

Figure 6: Runtime query execution time predictions vs actual execution times for SSB Q1.1.

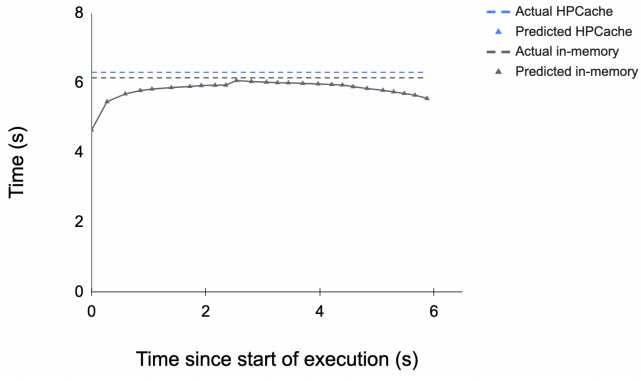

Figure 7: Runtime query execution time predictions vs actual execution times for SSB Q2.2.

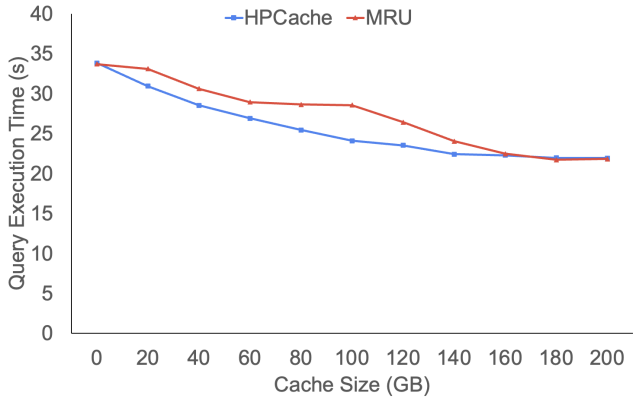

Figure 8: Two queries with disjoint sets of input columns, run one after the other. 96GB working set per query.

Predicted Execution Times. Figure 6 and Figure 7 evaluate the accuracy of the run-time estimates of the pipeline execution times for SSB Q1.1 and SSB Q2.2 respectively. SSB Q1.1 is an input-access-sensitive query, while SSB Q2.2 is a processing-sensitive query. We choose these two queries because we want to evaluate our model on both input-access-sensitive and processing-sensitive queries. Each query is run with a warm cache and an input memory budget of 25 GB – approximately $^1/_4$ of its working set. SSB Q1.1 has two main pipelines. The first, shorter pipeline selects from the dimension table to build a hashtable for a hash join, and the second selects from the fact table and probes the hashtable. The build pipeline only consumes 4 pages of data and completes too quickly for HPCache to estimate execution times, so HPCache does not attempt to optimize that pipeline's inputs. SSB Q2.2 is similar, but has three small build pipelines. Hence, the predictions are solely for the probe pipeline of each query. We plot the pipeline execution time predictions made while the pipeline is executing as per Equation 1 as well as the in-memory run time predictions $\left(\frac{B}{P_{bw}}\right)$ as per Section 4.3. Further, we also plot, with dashed lines, the actual run time of the pipelines with i) 25 GB memory budget, and ii) a fully in-memory execution.

While HPCache slightly underestimates the processing throughput for in-memory data, its consistent nature means it has minimal effect on the actual cache configuration. Since the Q1.1 probe pipeline is a input-access-sensitive pipeline, the predicted HPCache time is based on the storage and memory-bandwidth. In contrast to Q1.1, Q2.2 is processing-sensitive, so the current run-time prediction and the in-memory prediction are identical and are both based on the pipeline's predicted maximum throughput. The prediction model consistently underestimates the real execution time, both for the currently executing query and for the in-memory case. It does not account for the time to open or close the pipelines. Additionally, the in-memory predictions assume that all 48 logical cores are only processing the pipeline. It does not account for anything else utilizing the CPU, such as the background prediction thread, and the overhead of looking up pages in the buffer pool. Further, the actual HPCache execution time of Q2.2 slightly exceeds the in-memory time due to the overhead for the storage IO management.

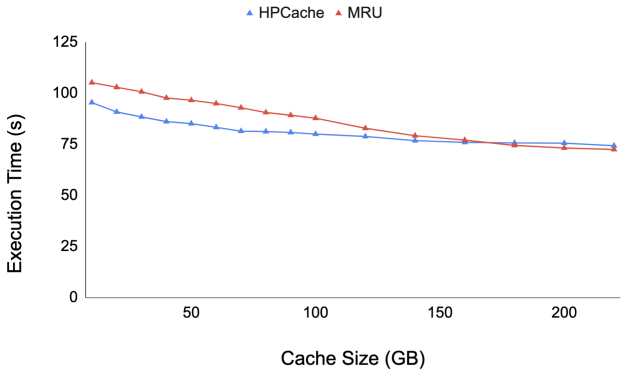

Figure 9: MRU versus HPCache on the full SSB.

Alternating Queries with Disjoint Input Columns. Figure 8 demonstrates the run-time adaptivity of proportional caching. We alternatively run two queries, each three times. Each query has throughput greater than storage bandwidth; the first has a throughput of 32 GB/s and the second 40 GB/s. We repeat the experiment 5 times and report the average. We compare HPCache with a most recently used eviction policy. There is a distinct plateau for MRU between 80 GB and 100GB; this is because at this point MRU has already cached enough data for the first query such that it is no longer input bandwidth bound, but has not cached any data for the second query which is still input bandwidth bound. The MRU execution time then starts improving again after 100 GB as it starts caching input data for the second query. This contrasts with HPCache which adapts to the workload and caches input data that most speeds up both queries, and does so more consistently as the memory budget increases. MRU converges to HPcache when the full working set fits in memory, but HPcache outperforms MRU in this experiment for all the cases as it uses the budget more efficiently. With a memory budget of 100 GB, HPCache achieves a speed up of 15.6% compared to MRU.

5.2 SSB: a stress test

Figure 9 shows the performance of HPCache and a MRU caching policy on the full SSB for varying memory budgets. We measure total execution by running each query once in order starting from a cold cache. For each configuration, we run the experiment 5 times, clearing the cache before each iteration, and report the average. The order of queries in the SSB is beneficial to MRU, because the three most input-access-sensitive queries, Q1.1, Q1.2 and Q1.3 (which share the same input columns) are the first three queries in the benchmark. For memory budgets greater than 160 GB, that are comparable in size to the working set of all of the queries, MRU slightly outperforms HPCache. Both HPCache and MRU reduce the execution time as the memory budget increases: inputs are cached for several of the SSB queries that are input-access-sensitive. The performance improvements of both flatten out at around 150 GB as at this point, enough input data are cached such that the input-access-sensitive queries are no longer bottlenecked by loading their inputs. With a memory budget of 20 GB HPCache achieves same performance as MRU with a budget of 80 GB.

6 BACKGROUND & RELATED WORK

OS-assisted buffer pools. Due to the difficulty of implementing an efficient buffer pool, systems have often abandoned buffer pools altogether and instead relied on the operating system kernel's paging capabilities, often through mmap; for example, MonetDB relies on memory-mapped IO to support larger than memory datasets [5, 18]. Memory-mapped IO offloads a significant part of the buffer pool implementation to the OS and offers hardware-assisted page-fault handling. However, memory-mapped IO also reduces the control that the database engines have over buffer management and can even result in sufficient contention and reduced prefetching that prevents the database engine from saturating NVMe arrays [12].

Optimizing buffer pool accesses. Providing persistency and support for out-of-memory data traditionally introduces two overheads with respect to the buffer pool. First, having a centralized buffer pool creates a point of contention [19]. Second, persistency requires a level of indirection when translating in-memory references to out-of-memory object references. Graefe et al. [17] use pointer swizzling to eliminate buffer pool overheads when all data fits in memory. Pointer swizzling dereferences page references and replaces swizzled pointers with in-memory pointers. By avoiding a hashtable, they avoid a costly central point of contention. LeanStore [22] extends on pointer swizzling by speculatively unswizzling pages, which keeps hot pages in memory without explicitly tracking page accesses in shared data structure. Umbra [26] extends LeanStore with support for variable length buffer frames, improving handling of large objects. Overall, optimizing buffer pool accesses reduces the overhead imposed on mostly-in-memory analytics while adding support for out-of-memory data. In contrast, HPCache improves the cache efficiency with respect to the performance gains achieved by caching data in memory.

Buffer pool eviction policies. Apart from handling larger-than-memory datasets, buffer pools promise efficient in-memory data caching. Multiple eviction policies have been proposed to increase the cache hit frequency, using the databases' access patterns, such as partitioning the buffer pool by relation [34], or into priority or access patterns zones and using an access pattern-optimized eviction policy inside each partition [27, 30]. Partitions are sized based on the expected performance benefit. Other approaches provide each query with a buffer large enough for the queries modelled hotset of pages [31]. LRU and MRU are used to increase the hit chance inside each partition, with second-chance eviction policies like 2Q reducing the cache pollution [20]. However, directly-attached NVMe arrays provide significant bandwidth to make data scans competitive to query execution times. Instead of relying on frequency-based cache eviction, HPCache takes into consideration the overall benefit of data caching and prioritizes caching of high-benefit data over frequently-loaded but low-benefit inputs.

Heterogeneous storage. The multitude of available storage devices provides a rich spectrum of performance and budget trade-offs. Do et.al [15] reduce the computational cost of log-structured storage by offloading the computation required for garbage collection and recovery onto computational SSDs. Mosaic [35] is a storage engine specialized for scan-heavy workloads. It calculates performance-budget Pareto-optimal data placements for data residing across multiple types of storage devices. Mosaic uses workload

traces to model column-granular data placement as an optimization problem and solves it offline using linear optimization. Borovica et al. [8] propose Skipper, an execution framework optimized for cold storage devices (CSD): as CSD can result in high delays when accessing data from powered off disks, Skipper uses out-of-order execution to codesign the execution order and disk requests to hide unnecessary latencies. Both Skipper and Mosaic optimize the performance and cost of analytics on scan-heavy workloads, assuming the storage medium is the bottleneck. In contrast, in this work, we focus on improving analytical response times over high-bandwidth, directly attached NVMes.

7 CONCLUSION

In this paper we show that i) caching pages that are accessed with the same frequency can yield significantly different query acceleration results, and ii) optimizing the memory footprint requires partial column caching to avoid diminishing returns. We proposed HPCache, an eviction policy and tuning agent that optimizes the caching decisions of the buffer pool for analytical workloads on high-bandwidth storage. HPCache both inspects query execution to predict caching benefits and automatically tunes page caching priority. HPCache improves the efficiency of in-memory data caching for analytics, allowing faster query execution time and improved NVMe bandwidth utilization for a given memory budget. HPCache achieves up to a 12% speed up or 4x reduction in memory utilization compared to a MRU eviction policy.

ACKNOWLEDGMENTS

This work was partially funded by the Swiss National Science Foundation (SNSF) project "Efficient Real-time Analytics on General-Purpose GPUs" subside no. 200021_178894/1.

REFERENCES

[1] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. 2018. A rewriting system for convex optimization problems. *Journal of Control and Decision* 5, 1 (2018), 42–60.

[2] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. 2019. Optimal column layout for hybrid workloads. *Proceedings of the VLDB Endowment* 12, 13 (Sept. 2019), 2393–2407. https://doi.org/10.14778/3358701.3358707

[3] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 168–180. https://doi.org/10.1145/3448016.3452831

[4] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). ACM, 37–48. https://doi.org/10.1145/1989323.1989328

[5] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (Dec. 2008), 77–85. https://doi.org/10.1145/1409360.1409380

[6] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database architecture optimized for the new bottleneck: Memory access. In *VLDB'99, proceedings of 25th international conference on very large data bases, september 7-10, 1999, edinburgh, scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 54–65. http://www.vldb.org/conf/1999/P5.pdf tex.bibsource: dblp computer science bibliography, https://dblp.org tex.biburl: https://dblp.org/rec/conf/vldb/BonczMK99.bib tex.timestamp: Wed, 11 May 2022 08:53:25 +0200.

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. CIDR, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[8] Renata Borovica-Gajic, Raja Appuswamy, and Anastasia Ailamaki. 2016. Cheap data analytics using cold storage devices. *Proc. VLDB Endow.* 9, 12 (2016), 1029–1040. https://doi.org/10.14778/2994509.2994521

[9] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2010. SSD Bufferpool Extensions for Database Systems. *Proc. VLDB Endow.* 3, 2 (2010), 1435–1446. https://doi.org/10.14778/1920841.1921017

[10] Hong-Tai Chou and David J. DeWitt. 1986. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Algorithmica* 1, 3 (1986), 311–336. https://doi.org/10.1007/BF01840450

[11] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proceedings of the VLDB Endowment* 12, 5 (Jan. 2019), 544–556. https://doi.org/10.14778/3303753.3303760

[12] 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*. 7.

[13] Advanced Micro Devices. 2021. *AMD EPYC™ 7003 SERIES Data Sheet.* Advanced Micro Devices. https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf

[14] Steven Diamond and Stephen Boyd. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research* 17, 83 (2016), 1–5.

[15] Jaeyoung Do, Ivan Luiz Picoli, David B. Lomet, and Philippe Bonnet. 2021. Better database cost/performance via batched I/O on programmable SSD. *VLDB J.* 30, 3 (2021), 403–424. https://doi.org/10.1007/s00778-020-00648-z

[16] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2012. Dark Silicon and the End of Multicore Scaling. *IEEE Micro* 32, 3 (2012), 122–134. https://doi.org/10.1109/MM.2012.17

[17] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. 2014. In-memory performance for big data. *Proceedings of the VLDB Endowment* 8, 1 (Sept. 2014), 37–48. https://doi.org/10.14778/2735461.2735465

[18] Stratos Idreos, F. Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35 (Jan. 2012), 40–45. http://sites.computer.org/debull/A12mar/monetdb.pdf

[19] 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 - EDBT '09*. ACM Press, Saint Petersburg, Russia, 24. https://doi.org/10.1145/1516360.1516365

[20] Theodore Johnson and Dennis E. Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann, 439–450. http://www.vldb.org/conf/1994/P439.PDF

[21] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast queries over heterogeneous data through engine customization. *Proceedings of the VLDB Endowment* 9, 12 (Aug. 2016), 972–983. https://doi.org/10.14778/2994509.2994516

[22] 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)*. IEEE, Paris, 185–196. https://doi.org/10.1109/ICDE.2018.00026

[23] John D. McCalpin. 1991-2007. *STREAM: Sustainable Memory Bandwidth in High Performance Computers.* Technical Report. University of Virginia, Charlottesville, Virginia. http://www.cs.virginia.edu/stream/ A continually updated technical report. http://www.cs.virginia.edu/stream/.

[24] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[25] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[26] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. CIDR. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[27] Chris Nyberg. 1984. *Disk scheduling and cache replacement for a database machine.* Master's thesis. UC Berkeley.

[28] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking (Lecture Notes in Computer Science)*, Raghunath Nambiar and Meikel Poess (Eds.). Springer, Berlin, Heidelberg, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

[29] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through Elastic Resource Scheduling.

arXiv:2004.05437 [cs, eess] (April 2020). http://arxiv.org/abs/2004.05437 arXiv: 2004.05437.

[30] Allen Reiter. 1976. *A Study of Buffer Management Policies for Data Management Systems.* Technical Report. WISCONSIN UNIV MADISON MATHEMATICS RESEARCH CENTER.

[31] Giovanni Maria Sacco and Mario Schkolnick. 1982. A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model. In *Eigth International Conference on Very Large Data Bases, September 8-10, 1982, Mexico City, Mexico, Proceedings*. Morgan Kaufmann, 257–262. http://www.vldb.org/conf/1982/P257.PDF

[32] Allen Samuels. 2016. The Consequences of Infinite Storage Bandwidth. (2016). https://events.static.linuxfound.org/sites/events/files/slides/Keynote_Allen%20Samuels_Final.pdf Vault Linux Storage & Filesystems Conference.

[33] Utku Sirin and Anastasia Ailamaki. 2020. Micro-architectural Analysis of OLAP: Limitations and Opportunities. *Proc. VLDB Endow.* 13, 6 (2020), 840–853. https://doi.org/10.14778/3380750.3380755

[34] Michael Stonebraker, John Woodfill, Jeff Ranstrom, Marguerite C. Murphy, Marc Meyer, and Eric Allman. 1983. Performance Enhancements to a Relational Database System. *ACM Trans. Database Syst.* 8, 2 (1983), 167–185. https://doi.org/10.1145/319983.319984

[35] Lukas Vogel, Viktor Leis, Alexander van Renen, Thomas Neumann, Satoshi Imamura, and Alfons Kemper. 2020. Mosaic: a budget-conscious storage engine for relational database systems. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 2662–2675. https://doi.org/10.14778/3407790.3407852

[36] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. 2017. LRC: Dependency-aware cache management for data analytics clusters. In *2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*. IEEE, 1–9. https://doi.org/10.1109/INFOCOM.2017.8057007

[37] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (July 2015), 1920–1948. https://doi.org/10.1109/TKDE.2015.2427795