Short Paper

Adaptive Compression For Databases

Leon Windheuser
Technical University of Munich
leon.windheuser@tum.de

Christoph Anneser
Technical University of Munich
anneser@in.tum.de

Huanchen Zhang
Tsinghua University
huanchen@tsinghua.edu.cn

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

ABSTRACT

Efficient utilization of dynamic random access memory (DRAM) is crucial for achieving high-performance query processing in database systems, especially as data volumes continue to grow. Unfortunately, the cost of DRAM is unlikely to decrease in the coming years, and it is already the dominating cost factor in modern data centers. Consequently, lightweight in-memory compression techniques can reduce the memory footprint and maximize the data stored in memory. However, compressing all data, regardless of the compression algorithm's efficiency, causes additional CPU overhead during query execution. To address this challenge, we introduce AdaCom, a novel framework that selectively applies lightweight succinct encodings only to infrequently accessed data. By doing so, we mitigate the performance overhead associated with compression. In our experimental evaluation, we demonstrate that AdaCom reduces the memory footprint by up to 40% while retaining most of the performance (≈ 95%).

1 INTRODUCTION

In recent years, database systems that store large fractions of their datasets in dynamic random access memory (DRAM) became increasingly popular because of their unprecedented query performance [16, 27, 32]. They enable real-time analytics over terabytes of data, which is a desired feature for financial services and emerging new business types. On the one hand, the overall data volume is increasing exponentially, and it is expected to reach 175 ZB by 2025 [39]. On the other hand, because of the end of Moore's law [37], memory (DRAM) prices stabilized recently [26, 29], making pure in-memory database systems impractical for large datasets. For example, memory has already become the dominating cost factor in data centers, with 40% of Meta's rack costs [31] and 50% of Azure's servers [3] coming from memory.

Decreasing the memory footprint of database systems can reduce operating costs and improve performance by fitting more data into memory, resulting in fewer cache misses and memory stalls. Several ideas have already been proposed to reduce the DBMS memory footprint, many focusing on more space-efficient index structures [6, 42, 43] or lightweight data compression algorithms [23, 25].

However, applying lightweight in-memory compression to entire relations or columns is not optimal because no matter how lightweight the compression algorithm is, it will still incur extra computation during query execution, negatively impacting query performance.

In this work, we propose **Ada**ptive **Com**pression (AdaCom) for databases, which is a new framework that borrows the principle ideas of workload-adaptive hybrid indexes [6]: (1) We use lightweight sampling to track the data segment's accesses and (2) periodically identify frequently accessed ('hot') and rarely accessed ('cold') data segments. (3) Then, we adaptively compress cold data segments using simple bit packing and uncompress the hot data segments.

We integrate AdaCom into DuckDB and provide an extensive experimental evaluation in Section 4. Our experiments show that applying lightweight bit packing to DuckDB's column segments can improve performance by up to 7× when DuckDB's standard encoding no longer fits into memory. For datasets fitting completely into memory, AdaCom can reduce the memory footprint by up to 38% while achieving 95% of the performance-optimized encodings.

We make the following contributions:

- We introduce AdaCom, a generic approach to adaptively apply lightweight compression to rarely accessed data in database systems.
- We provide an open-source prototype implementation of AdaCom and a full system-integration for DuckDB[1].
- Based on an extensive set of micro- and end-to-end system benchmarks, we demonstrate AdaCom's applicability and experimentally show that it can significantly reduce DuckDB's memory by up to 40% footprint while retaining most of its performance (≈ 95%).

2 RELATED WORK

Compressing Data for Disk. The performance of disk-based database management systems is mainly dominated by the high latency and low bandwidth of disk I/O. Despite the recent advances of modern non-volatile memory express (NVMe) cards, a large performance gap remains between DRAM and disk, which makes even more compute-intensive compression algorithms attractive as long as the reduced I/O outweighs the de-/compression's CPU overhead. Therefore, much work has been done on evaluating the various compression techniques on database storage [9, 20, 21, 38].

Lightweight In-Memory Compression. With the advent of in-memory database systems enabled by ever-growing memory capacities, fetching data from DRAM has become the new bottleneck [33]. However, the difference in random access latencies between processor caches (≈ 1ns) and DRAM (≈ 10 − 50ns) is much smaller compared to fetching data from SSDs (≈ $10^5 ns$) [11, 41], and in contrast to disk compression, in-memory compression must be *far less compute-intensive* to be outweighed by the compression benefits. Therefore, the database community has started to explore applying *lightweight* in-memory compression

[1] https://github.com/leonwind/duckdb-adaptive-compression

to database systems. Krüger et al. [28] evaluated several lightweight compression techniques, such as run-length, bit-vector, and dictionary encoding, as well as null suppression for analytical and transaction processing in column-oriented in-memory database systems. Increasing memory capacities also result in an increase of access latencies to DRAM [33]. For that reason, SanssouciDB uses lightweight in-memory compression to utilize memory more efficiently by fitting more data into cache lines, improving the overall query performance and reducing its memory footprint [33].

'*Succinct*' is a data store that uses a compressed representation for its input data and reduces the memory consumption by up to an order of magnitude [4]. Furthermore, its new query algorithm allows for random access, count, search, wildcard, and range queries directly on the succinct data representation. In 2021, Heinzl et al. [23] stress the importance of integer columns in query processing based on various synthetic and real-world benchmarks. Therefore, they investigated seven open-source integer compression libraries and algorithms, including bit packing, null suppression, frame-of-reference (FOR), dictionary, and delta encoding, and evaluated their performance and compression ratio in the in-memory HTAP database system Hyrise [22]. They conclude that most benchmarks such as TPC-H, TPC-DS [1], and JOB [30] require efficient random access to integer columns and, therefore, perform better with lightweight *bit packing* compression.

Succinct Data Structures. Succinct data structures (SDS) reduce the space consumption close to the information-theoretical lower bound [24]. In contrast to general-purpose compression algorithms like LZ4 [10] or Snappy [13], succinct encodings allow for efficient random data access in constant time ($O(1)$) without requiring the entire structure to be decompressed first. Therefore, succinct data structures could be attractive for in-memory databases that must store rapidly growing volumes of data to help fit more data into memory and reduce the amount of expensive I/O disk accesses. A variety of data structures can be succinctly encoded, including trees [14, 24], filters [12, 15, 44, 45], and graphs [18]. Nonetheless, succinct data structures smaller memory footprint comes at the cost of an increased number of instructions for accessing single data elements (arithmetic shifts for data alignment), which is one of the reasons for their limited adoption in database systems. Therefore, adaptively applying succinct encodings only to rarely accessed data can help to improve the overall index performance significantly [5, 6].

Succinct Integer Vectors (Bit Packing). Integer vectors can be encoded succinctly by using only the required number of bits per element, which is also known as *bit packing*. Usually, the bits-per-element, also referred to as *bit width*, is determined by the maximum element in the vector v. For example, if m is the maximum element in v, then every element e can be represented by using $\lceil \log_2(m) \rceil$ bits.

Succinct integer vectors can reduce the memory footprint significantly, as the following example illustrates: A vector storing 10B 64-bit integers, each in the range $[0, 10^{10})$, results in a total size of 80 GB. With bit packing, we can encode every element with $\lceil \log_2(10^{10} - 1) \rceil = 34$ bits only, which reduces the total size to 34 bits $\cdot\ 10^{10} = 42.5$ GB, which is a relative space reduction of 46.875%.

Since every element is represented by the same number of bits w, we can randomly access the i-th element in $O(1)$, as the element is stored in the interval $[i \cdot w, i \cdot w + w]$. As this range is not necessarily a multiple of a byte and does not need to be memory-aligned, retrieving its values requires additional shift instructions. However, a succinct integer vector should have a better cache locality due to the reduced size.

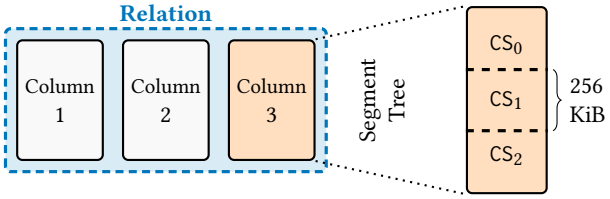


Figure 1: Overview of DuckDB's column layout. Each column is split into column segments (CS) which are indexed by a segment tree.

Succinct Data Structure Library (SDSL). The Succinct Data Structure Library [19] provides different, highly optimized succinct data structures in `C++`. For this work, we use SDSLs succinct integer vector, which has a similar interface as `std::vector` in `C++` with additional capabilities, e.g. bit-compression. We experimentally investigate the impact on performance and cache locality for different access patterns in Section 3.1.

DuckDB. DuckDB [36] is an open-source relational database system optimized for analytical queries (OLAP) and running in-process DBMS embedded in a host application, making it attractive for data scientists and analysts. It uses a column-oriented storage architecture and splits its columns further into smaller column segments, each having a size of up to 256 KiB. The column segments are then internally organized using a segment tree as visualized in Figure 1.

DuckDB implements several compression algorithms for persistent storage, all of them being optimized for a high compression ratio to reduce I/O rather than providing the lightweight compression offered by succinct representations that allow accessing single elements in constant time [35].

3 APPROACH

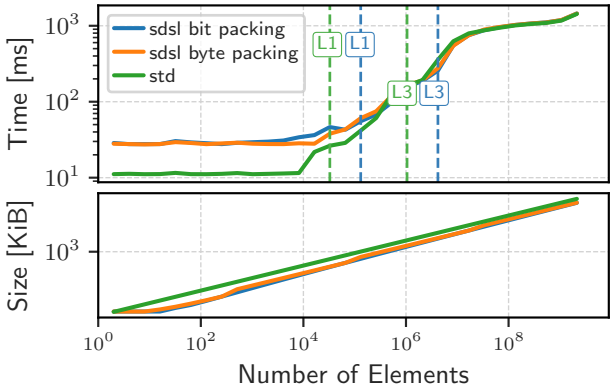
In this work, we propose a new framework for database systems that adaptively applies lightweight compression to cold data only and, therefore, limits the compression overhead.

3.1 Lightweight In-Memory Compression

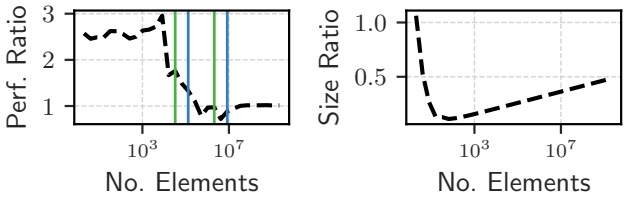
First, we perform micro-benchmarks to understand better the implications of lightweight in-memory compression on space consumption and performance.

We focus on integer columns, as they are the most frequently accessed ones during query processing [23]. We consider vectors containing between two and 2^{31} 64-bit integers and shuffle their elements using Sattolo's algorithm [40] to generate a one-cyclic permutation and use it to perform 10M uniformly distributed point lookups. We execute the benchmark multiple times and compare the `std::vector` (■) to the `sdsl::vector` using bit packing (■) and byte packing (■). Furthermore, we denote the ratio of either latency or size (depending on the plot) of `sdsl::vector` and `std::vector` (■) and visualize the results in Figure 2.

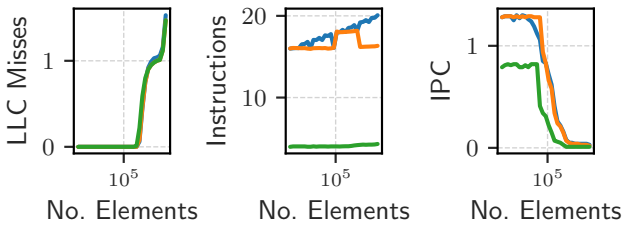
For up to approximately 10^4 elements, the `sdsl::vector` increases the execution time by 3× compared to the `std::vector`, because both vectors fit into the L1 cache and the extra CPU costs of accessing the elements of the succinct vector dominate.



(a) Execution time and the size of the different vector implementations. The vertical green (`std::vector`) and blue (`sdsl::vector`) lines represent the number of elements fitting into the CPU caches. All axes are on a log scale.



(b) Overhead ratio of `sdsl::vector` and `std::vector` for run time and size. Lower values favor using the `sdsl::vector`.



(c) In-depth performance evaluation with `perf`.

Figure 2: Performance comparison of `sdsl::vector` and `std::vector`. All x-axes are on a log scale.

Contrary, with an increasing number of elements, the succinct vector performs slightly better since the default vector exceeds the caches with fewer elements (cf. the performance ratio below one in Figure 2b). As the vectors grow larger than the last level caches (LLC), the performance ratio converges against one as the cache misses begin to outweigh the extra instructions needed to access elements of the succinct vector.

The `sdsl::vector` requires significantly less memory compared to `std::vector`. For 2^{31} elements, the succinct encoding reduces the memory consumption by over 50% since all elements can be encoded with 31 bits compared to the 64 bits required by the `std::vector`.

To verify our observations and to gain deeper insights, we used the `perf` to collect hardware performance counters and show the results in Figure 2c. Surprisingly, for 1M elements, the LLC misses only slightly increase from 0.06 for `sdsl::vector` to 0.22 for `std::vector`, and the difference converges towards 0 with an increasing number of elements, even though `std::vector`'s memory footprint is 50% larger.

As expected, the `sdsl::vector` comes at the cost of more instructions for a single query due to the unaligned memory accesses and shift operations for extracting single elements. While the difference in the number of instructions seems large (4-5 vs. 16-20 instructions per access), the instructions per cycle (IPC) metric shows its impact is limited as the CPU can handle it well during stalls.

Padding To Byte Alignment. As mentioned above, the succinct encoding based on bit packing incurs additional CPU overhead due to unaligned memory accesses. However, we can address this issue by adjusting the size of each element to align with whole bytes, a technique known as *byte packing*. For example, instead of 7, 15, or 23 bits per element, we would use 8, 16, or 24 bits, respectively. While padding increases the overall space consumption, it would reduce the number of instructions and thus increase the performance.

However, Figure 2 and the experimental evaluation in Section 4, show that the implication on the overall performance is limited as both are in the same order of magnitude for LLC misses, instructions, and IPC. While bit packing causes slightly fewer LLC misses and, therefore, utilizes the caches better, it also requires slightly more instructions per access. Consequently, the overhead incurred by unaligned memory accesses in the context of bit packing is relatively modest.

3.2 Adaptive Compression

The experiments in Section 3.1 and in [23] have shown that lightweight in-memory compression for integer columns can significantly reduce the database's memory consumption by up to 50% while having a negative impact on raw query performance of up to 10% because of the additional instructions for accessing and decompressing the data. At the same time, most workloads have skewed access patterns, where only a small part of the data is accessed frequently, while most data is rarely accessed [6, 8]. Therefore, we propose AdaCom, which adaptively compresses rarely accessed data to reduce the overall memory footprint while keeping frequently accessed data in a performance-optimized, uncompressed representation.

We visualize our approach in Figure 3. As workloads and queries (1) are unknown beforehand, we track the column segment (CS) accesses (2) at run time to differentiate frequently from rarely accessed segments. For example, 60% of the queries access CS_3, but only 5% access CS_0. Tracking all column accesses would incur significant performance overheads. Instead, AdaCom samples only a subset of the accesses and keeps track of the access statistics in a hashtable (3). Based on the access statistics, we periodically adapt the segment encodings.

AdaCom has two parameters *compaction threshold* and *adaptation period* that have implications on the effectiveness and the performance of our approach:

Compaction Threshold. Based on the sampled column access statistics that are stored in the hashtable (3), we define a compaction threshold $\alpha \in [0, 1]$ that determines how aggressively column segments are compressed. For example, $\alpha = 0.75$ means that 75% of the columns with the fewest accesses will be migrated to the compact encoding. Please note that the compaction threshold could also be made adaptive depending on the available buffer and memory size.

Adaptation Period. AdaCom has an *adaptive compaction manager* (ACM) (4) that runs in the background and initiates the migration of column segments in equidistant time periods. Once the adaptation starts, the ACM (1) sorts the column segments by their access statistics. Next, it (2) compresses rarely accessed,

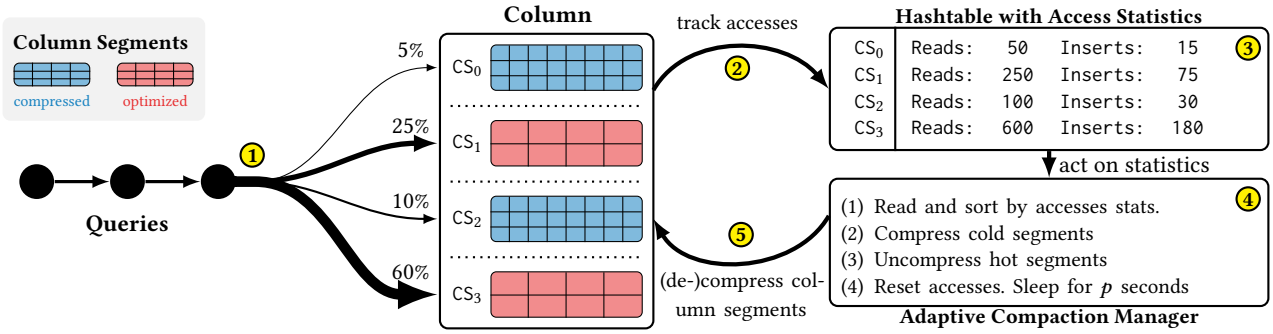


Figure 3: Overview of AdaCom's approach to adaptively compress column segments based on the access statistics.

cold column segments and (3) uncompresses the frequently accessed, *hot* segments if they are not yet stored in the targeted encoding format ⑤. Last, the ACM (4) resets the access statistics and sleeps for a predefined adaptation period p.

If the workloads and access patterns change, compact segments can decompress again and vice versa. The ACM runs in the background without interrupting the query processing since it only needs to read the existing segment to create the compacted/uncompacted version in parallel. While this approach creates a temporary memory overhead during the compression and decompression of column segments, the size overhead is limited since the column segments are usually small, e.g., 256 KiB in DuckDB.

Updates and Inserts. As the bit width in bit packing depends on the maximum element in the vector, updates and inserts may require a re-encoding of the entire vector, e.g., when the new or modified value is larger than the current maximum and cannot be represented by the current bit width.

However, in real-world workloads, inserts are often skewed, and a few key ranges are updated and inserted the most [7], but they can still affect a compacted segment. Because decoding a 256 KiB segment is computationally expensive, we would like to avoid an *oscillation effect* where a segment, which is neither clearly hot nor cold, gets de-/compressed every other adaptation phase. Therefore, we track a history of inserts per segment and disallow compaction of recently decompressed segments.

A perfect use-case for our approach are primary key columns, as they should never get updated and the keys are monotonously increasing. Enumerations (enums) are another good example since the number of enum items is usually fixed and relatively small, which offers a good compression ratio.

Integration into DuckDB. We integrated AdaCom into DuckDB, where it tracks the accesses per column segment in a hashtable. We run the adaptive compression manager in a new thread that periodically initiates the adaptation process ④ every p seconds.

We use the `sdsl::vector` for the compact and the `std::vector` for the default encoding of column segments. To further reduce the space consumption of the compacted column segments, we use frame-of-reference (FOR) encoding. Therefore, we extract the minimum element of each column segment `CS` and store only the difference $e - min$ for every e in `CS`, which reduces the number of bits (bit width) that is used by the `sdsl::vector`.

Since DuckDB splits a column into smaller segments and tracks basic statistics for each segment (such as the minimum and maximum element), our approach can utilize this information when compacting a segment without requiring additional work. DuckDB's column segments have a maximum size of 256 KiB and can, thus, only store up to 65,536 4-byte integers. For example, if the column segment's elements are monotonically increasing, as is the case for primary keys, FOR encoding reduces the space per element to two bytes only, which provides a memory improvement of $\geq 50\%$.

4 EVALUATION

Experimental Setup. We conducted all experiments on a server equipped with a single socket Intel® Core™ i9-7900X CPU at 3.30 GHz, 10 physical cores, 125 GB of DRAM, and a Samsung SSD 850 EVO with 2 TB. We run Ubuntu 22.10 and compile our `C++` code with `g++ 12.2.0` and the `-O3` flag.

We set DuckDB's buffer size to the available physical memory and run end-to-end SQL queries in DuckDB.

DuckDB Memory Consumption. Besides the memory allocated to represent the column data, DuckDB allocates further memory, e.g., to store validity masks. In our evaluation, this additional memory overhead is included in our results. If we would, instead, consider the 'raw data' (e.g., the column data) only, we would achieve even better compression ratios, e.g., saving up to 50% of the data for 32-bit and 75% for 64-bit integers by using `sdsl::vector` and frame-of-reference encoding.

Competitors. We compare three different column segment encodings: the default encoding uses DuckDB's internal vector interface (■), and the succinct encoding is based on the `sdsl::vector` either using bit (■) or byte packing (■).

4.1 Sequential Scan

We scan a primary key column containing 5B primary key 64-bit integers. The results of the three encodings introduced above to store the column segments are shown in Figure 4.

Byte packing/bit packing increases the execution time by 9% / 8.5% of the default column representations performance while reducing the memory footprint by 61.5% / 60%. Furthermore, the difference between bit and byte packing on the performance or the memory consumption is minimal (1.5% performance improvement and 3% memory increase).

Most of `sdsl::vector`'s overhead comes from *copying* data. The default encoding does not require data copies and only passes the data pointer to the following operator. As the query processing layer currently cannot handle the `sdsl::vector`, we must copy the data into DuckDB's internal vector before continuing query processing.

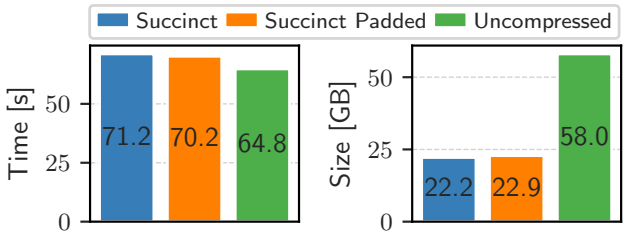


Figure 4: Performance and size of three CS encodings. *Succinct Padded* references the byte-packed encoding.

4.2 Out-of-Memory Performance

As our previous experiments have shown, succinct encodings can significantly reduce the space consumption of integer columns and allow us to fit more data into memory, reducing the number of expensive disk accesses. For this experiment, we scan a column with 5B 64-bit integers and use DuckDB's default vector (■) and a `sdsl::vector` with bit packing (■) to encode the column segments. Furthermore, we limit the buffer size of DuckDB to 25 GB. Figure 5 shows the results.

Compared to the default encoding, the `sdsl::vector` reduces the execution time and the space consumption by up to 7× and 61.5%, respectively, since the succinct encoding enables DuckDB to fit all data into memory. On the contrary, the default encoding leads to buffer overflows and eventually causes many disk accesses that dominate the execution time.

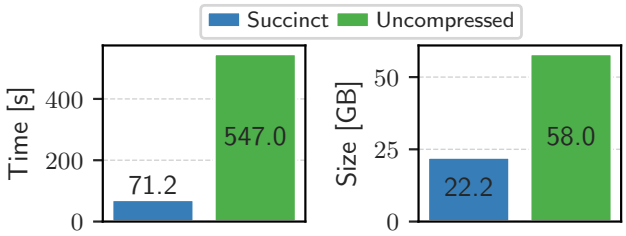


Figure 5: Sequential scan with a buffer size of 25 GB.

4.3 Adaptive Compression

In the following, we evaluate using AdaCom (■) to track accesses at run time and compress rarely accessed segments using the succinct encoding. In contrast, frequently accessed segments are stored in the default encoding. The adaptive compression manager (ACM) checks every $p = 10$ seconds for changing access patterns and adapts the segment encodings accordingly. We set the compaction threshold to $\alpha = 0.9$, meaning the ACM will compress 90% of the least frequently accessed segments. We evaluate our approach's performance by considering query throughput (QPS) and memory footprint.

4.3.1 Implications of Skewness. First, we investigate the performance implications of skewness on the ACM by running point lookups sampled from Zipf distributions having different skew factors k for one minute, ranging from $k = 0$ (no skew) to $k = 2.5$ (highly skewed), on a column with 10M 32-bit integers. Figure 6 shows the different competitors' average QPS and memory sizes.

While the memory usage is independent of the workload's skew, the QPS increases with a higher skew for all competitors due to better cache utilization. While AdaCom is slower than the uncompressed competitor for uniform workloads, it achieves similar performance for skewed workloads where $k \geq 1$ while reducing memory usage by 37% and outperforming the strict succinct encoded competitor.

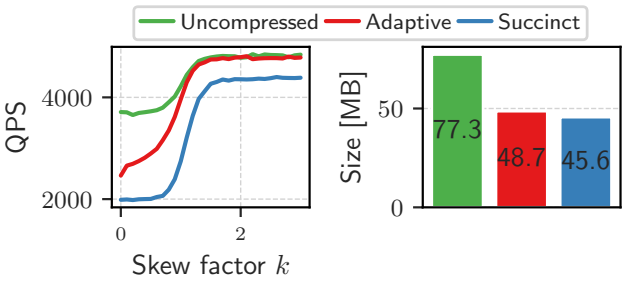


Figure 6: Zipf point lookup queries with different skew factors k from 0 (no skew) to 2.5 (highly skewed).

4.3.2 Meta's RocksDB Workload. We use RocksDB's tool dbbench to generate a workload based on anonymized 64-bit user ids that exhibit access patterns commonly observed at Meta [2, 7]. We evenly split the workload into three phases (where each phase has different access patterns) that we run sequentially, one after the other, on a database containing initially 53M user ids. Each phase consists of point lookups and new inserts of the anonymized user ids. Figure 7 shows the results.

Applying the default encoding to all column segments yields the highest average throughput of ≈ 3300 QPS. However, AdaCom is on average only 3% slower (≈ 3200 QPS), while keeping all segments in the succinct encoding decreases the QPS by 20% (≈ 2600 QPS). When the ACM changes the segments' encodings, the throughput drops to ≈ 2900 QPS for a short time. Since DuckDB is optimized for OLAP workloads and does not use indexes, point lookups require table scans, with small materialized aggregates for pruning, that result in a QPS of only 3300, which is low compared to other systems like SQLite [17, 34].

The memory consumption of AdaCom is up with the default uncompressed encoding until the ACM, after $p = 10$ seconds, starts compressing the rarely accessed segments, and the memory consumption gets reduced by ≈ 30%. Keeping the top 10% segments uncompressed increases memory consumption by 11% compared to the full-succinct option.

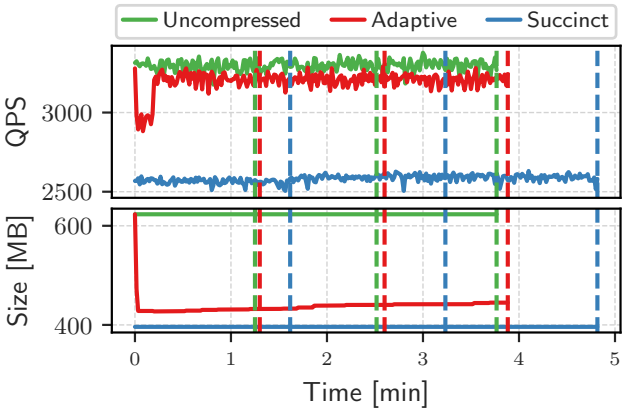


Figure 7: Generated workload with similar access patterns observed at Meta [7] consisting of three phases. Dashed lines indicate the end of each phase for each competitor.

5 DISCUSSION

As shown in the evaluation in Section 4, AdaCom effectively reduces the memory footprint while achieving a similar query performance. However, our approach has some limitations and challenges, further discussed in the following.

Generalizability. While we successfully integrated AdaCom into DuckDB, our prototypical implementation comes with increased code complexity due to the extensive modifications across various parts of DuckDB. This complexity raises questions about the generalizability of AdaCom. Specifically, AdaCom is tightly coupled with DuckDB's column segment architecture, which serves as the foundation for distinguishing 'hot' and 'cold' data segments within individual columns. This architectural dependency suggests that databases with different storage architectures may face challenges in directly applying AdaCom. Alternative approaches may be necessary to identify data segment candidates for compression, depending on the specific access patterns of the workload.

Furthermore, the current prototype has a limitation in the query processing engine. Instead of passing a pointer to the data, queried tuples from compressed column segments are *copied* into DuckDB's default vector. This limitation exists because the query engine does not support succinct vectors natively. Overcoming this constraint could even further minimize the performance gap between standard DuckDB and AdaCom.

AdaCom's Parameter. AdaCom's performance is influenced by two parameters: the *compaction threshold* α and the *adaptation period*. The optimal settings for these parameters depend on multiple factors, including available system resources and the specific workloads the database system processes.

To achieve the best performance – assuming that sufficient memory is available – the compaction threshold α should be set to 0, effectively turning off AdaCom. However, one of AdaCom's advantages lies in its ability to minimize memory consumption without sacrificing much performance. This capability enables databases to deliver comparable performance even on more budget-friendly hardware or cloud instances with limited memory resources (c.f. Section 4.2). Furthermore, the compaction threshold is flexible and can be leveraged for specific use cases. For example, it can be increased at night to conserve memory, which is acceptable when mostly long-running analytical jobs are processed. Conversely, the threshold can be lowered during the day to achieve higher query processing performance and faster response times, which are required for real-time analyses.

In contrast to the compaction threshold, heuristics for optimizing the adaptation period are more difficult to identify due to its high sensitivity to workload variability at run time. Generally, a shorter adaptation period allows AdaCom to react and adapt quickly to changes in workload distribution. However, the analysis and compaction introduce computational overhead, thereby diverting CPU resources from the query engine and negatively impacting the database's performance. The optimal length of the adaptation period may also depend on the 'stability' of the workload. For instance, if the workload remains stable for an extended time period, a longer adaptation period could be beneficial, as frequent column segment compression and decompression are less likely to occur. Conversely, shorter adaptation periods are advantageous if the workload changes rapidly, as they allow for more frequent adjustments in column segment compression and decompression. Our future work will explore various strategies for dynamically adjusting the adaptation period during run time.

6 CONCLUSIONS

In this work, we have shown that lightweight in-memory compression can significantly reduce the memory footprint of databases while achieving similar query performance. Our adaptive compression manager identifies frequently accessed column segments and keeps them in performance-optimized structures, while it compresses rarely accessed segments by keeping their negative impact on raw performance limited.

We plan to extend AdaCom to automatically tune the *compaction threshold* and the *adaptation period* at run time. Furthermore, we will integrate AdaCom into other database systems to assess the generalizability of our approach.

ACKNOWLEDGMENTS

We thank the anonoymous reviewers for their valuable feedback. This work has been supported by the German Research Foundation (DFG) within the SPP 2037 under grant no. Ke 401/22-2.

REFERENCES

[1] 1993. *TPC Website.* https://www.tpc.org/ Last accessed: October 27, 2023.
[2] 2012. *RocksDB.* https://rocksdb.org Last accessed: October 27, 2023.
[3] 2020. *CXL and GEN-Z iron out a coherent interconnect strategy.* https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/ Last accessed: October 27, 2023.
[4] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling Queries on Compressed Data. In *NSDI*. USENIX Association, 337–350.
[5] Christoph Anneser, Andreas Kipf, Harald Lang, Thomas Neumann, and Alfons Kemper. 2020. The Case for Hybrid Succinct Data Structures. In *EDBT*. OpenProceedings.org, 391–394.
[6] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive Hybrid Indexes. In *SIGMOD Conference*. ACM, 1626–1639.
[7] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao
[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *FAST*. USENIX Association, 209–223.
[9] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. 2001. Query Optimization In Compressed Database Systems. In *SIGMOD Conference*. ACM, 271–282.
[10] Yann Collet. 2011. *LZ4 - Extremely fast compression.* https://lz4.github.io/lz4/ Last accessed: October 27, 2023.
[11] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. 1999. A performance comparison of contemporary DRAM architectures. In *Proceedings of the 26th annual international symposium on Computer architecture*. 222–233.
[12] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *SIGMOD Conference*. ACM, 365–378.
[13] Jeff Dean, Sanjay Ghemawat, and Steinar H. Gunderson. 2011. *A fast compressor / decompressor.* https://github.com/google/snappy Last accessed: October 27, 2023.
[14] O'Neil Delpratt, Naila Rahman, and Rajeev Raman. 2006. Engineering the LOUDS Succinct Tree Representation. In *WEA (Lecture Notes in Computer Science)*, Vol. 4007. Springer, 134–145.
[15] Martin Dietzfelbinger and Rasmus Pagh. 2008. Succinct Data Structures for Retrieval and Approximate Membership (Extended Abstract). In *ICALP (1) (Lecture Notes in Computer Science)*, Vol. 5125. Springer, 385–396.
[16] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.
[17] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *Proc. VLDB Endow.* 15, 12 (2022), 3535–3547.
[18] Hana Galperin and Avi Wigderson. 1983. Succinct representations of graphs. *Information and Control* 56, 3 (1983), 183–198.
[19] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*. 326–337.
[20] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *ICDE*. IEEE Computer Society, 370–379.
[21] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *ICDE*. IEEE Computer Society, 370–379.
[22] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. 2010. HYRISE - A Main Memory Hybrid Storage Engine. *Proc. VLDB Endow.* 4, 2 (2010), 105–116.

[23] Linus Heinzl, Ben Hurdelhey, Martin Boissier, Michael Perscheid, and Hasso Plattner. 2021. Evaluating Lightweight Integer Compression Algorithms in Column-Oriented In-Memory DBMS.. In *ADMS@ VLDB*. 26–36.

[24] Guy Joseph Jacobson. 1988. *Succinct static data structures*. Carnegie Mellon University.

[25] Nusrat Jahan Lisa, Tuan Duy Anh Nguyen, Dirk Habich, Akash Kumar, and Wolfgang Lehner. 2019. High-Throughput BitPacking Compression. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*. 643–646. https://doi.org/10.1109/DSD.2019.00101

[26] Uksong Kang, Hak-Soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, S Jang, and Joo Sun Choi. 2014. Co-architecting controllers and DRAM to enhance DRAM process scaling. In *The memory forum*, Vol. 14.

[27] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206.

[28] Jens Krueger, Johannes Wust, Martin Linkhorst, and Hasso Plattner. 2012. Leveraging compression in in-memory databases. *Proceedings of DBKDA* (2012), 147–153.

[29] Seok-Hee Lee. 2016. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*. 1.1.1–1.1.8. https://doi.org/10.1109/IEDM.2016.7838026

[30] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.

[31] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *ASPLOS (3)*. ACM, 742–755.

[32] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[33] Hasso Plattner. 2011. SanssouciDB: An In-Memory Database for Processing Enterprise Workloads. In *BTW (LNI)*, Vol. P-180. GI, 2–21.

[34] Mark Raasveldt. 2021. *duckdb 10 times slower than sqlite processing time series.* https://github.com/duckdb/duckdb/discussions/2368#discussioncomment-1420415 Last accessed: October 27, 2023.

[35] Mark Raasveldt. 2022. *Lightweight Compression in DuckDB.* https://duckdb.org/2022/10/28/lightweight-compression.html Last accessed: October 27, 2023.

[36] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[37] Parthasarathy Ranganathan. 2017. More Moore: Thinking outside the (server) box. (2017). Keynote at the International Symposium on Computer Architecture.

[38] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. *SIGMOD Rec.* 22, 3 (1993), 31–39.

[39] David Reinsel-John Gantz-John Rydning, J Reinsel, and J Gantz. 2018. The digitization of the world from edge to core. *Framingham: International Data Corporation* 16 (2018).

[40] S. Sattolo. 1986. An Algorithm to Generate a Random Cyclic Permutation. *Inf. Process. Lett.* 22, 6 (1986), 315–317.

[41] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*. 1–11.

[42] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD*. ACM, 1567–1581.

[43] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *TKDE* 27, 7 (2015), 1920–1948.

[44] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. 323–336.

[45] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Kimberly Keeton, and Andrew Pavlo. 2019. Succinct Range Filters. *SIGMOD Rec.* 48, 1 (2019), 78–85.