Steering Memory Deduplication in Multi-Tenant Database Systems

Jannis Kowalick[1,*], Alexander Krause[1], Dirk Habich[1] and Wolfgang Lehner[1]

[1]Technische Universität Dresden

Abstract

Improving the memory density of in-memory columnar databases is a necessary step towards more efficient sharing of hardware in multi-tenant systems. Many general-purpose tools attempt to achieve this by scanning systems for duplicate memory regions, and deduplicating these redundancies. In this paper, we show that existing tools sacrifice efficiency for general-applicability, and outline the possibilities unlocked by tapping into the architectural guarantees, and existing metadata for in-memory databases. Based on our previous work on deduplication, where we used a hardware-accelerated delta mechanism to increase memory savings, we detail a process for semantics-based *steering* of this mechanism. This significantly reduces the search space for duplicates, by only comparing database columns which exhibit similar features. Furthermore, we gracefully handle cases where steering causes the comparison of mismatched columns. We benchmark our solution against the widely available Linux Kernel Samepage Merging (KSM) tool, evaluating both based on their achieved memory savings and execution speed for deduplicating the Internet Movie Data Base (IMDB) and Star Schema Benchmark (SSB) datasets. Our solution manages to perform an order of magnitude faster than KSM, while detecting close to the same amount of duplicate data.

Keywords

multi-tenancy, in-memory databases, OLAP, Data Streaming Accelerator, memory-deduplication

1. Introduction

Many years ago, hosting and administrating databases was a necessary requirement for organizations building their digital infrastructure. Today, this work can be delegated via Database as a Service (DBaaS) offerings, or indirectly through Software as a Service (SaaS). Backing these systems, *multi-tenant* databases are often used, consolidating many customers (tenants) into the same database instance, while upkeeping isolation guarantees. This improves the overall resource utilization, since each tenant does not need to provision separate hardware for their own maximum load [1]. In the case of *in-memory* analytical database systems, this efficient resource utilization is not necessarily a given, since memory capacity cannot be multiplexed across tenants like processing power can [2]. Hence, there is a hard limit to how many tenants can be co-located on a single machine.

Data De-Duplication (DeDup) is a promising strategy to save memory in such systems. If the same object of data is redundantly stored by multiple tenants, or in multiple locations by the same tenant, it can be collated into a single one, freeing up space. Even though each tenant may have their own completely unique schema, there are practical situations in which such redundancies appear:

- Multiple tenants can own copies of the same common datasets, such as currency/country information and date dimensions.
- Some tenants set up development and testing copies of their live database.
- Engineers create snapshots of tables for retrospective analysis.
- Intermediate stages of ETL processes include copies of the same columns with each processing step.

36th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), September 29 - October 01, 2025, Regensburg, Germany

jannis.kowalick@tu-dresden.de (J. Kowalick); alexander.krause@tu-dresden.de (A. Krause); dirk.habich@tu-dresden.de (D. Habich); wolfgang.lehner@tu-dresden.de (W. Lehner)

0009-0008-1308-5556 (J. Kowalick); 0000-0002-2616-8739 (A. Krause); 0000-0002-8671-5466 (D. Habich); 0000-0001-8107-2775 (W. Lehner)

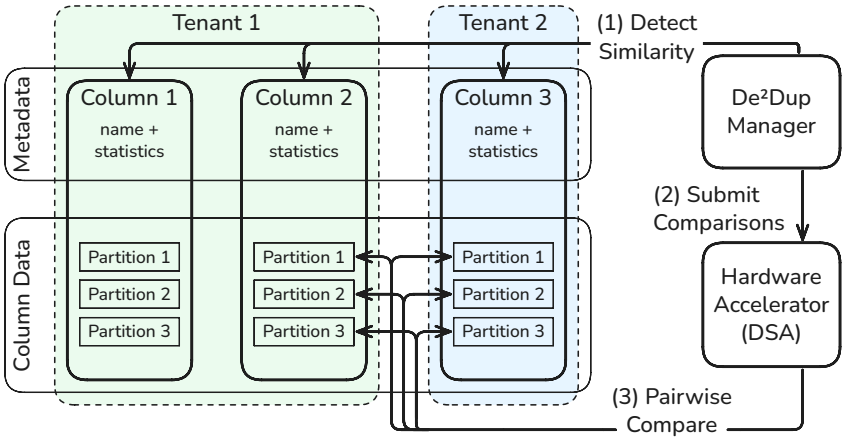


Figure 1: De^{2}Dup steered deduplication process

In some of these situations, exact copies of tables or columns are redundantly stored, while in others, data may only be similar or have been slightly updated over time. To effectively deal with both of these scenarios, our prior work introduced Delta De-Duplication (De^{2}Dup) [3]. By storing the columns of one dataset as a delta in reference to a base dataset, we showed that even tables which diverged from their initial identical state could still be efficiently deduplicated. At the core of this strategy is the De^{2}Dup manager, a new database component tasked with orchestrating the delta deduplication and instructing a specialized hardware accelerator to perform the bulk work.

Until now, we have not specified *how* the De^{2}Dup manager chooses which data to attempt deduplication on. In this paper, we therefore propose a concrete strategy for *steering* delta deduplication based on data similarity in in-memory analytical databases. As Figure 1 illustrates, the strategy exploits the database's common characteristics such as (1) per-column metadata to detect similarity, and (2) logical data ordering to decide how to steer the accelerator towards (3) a simple pairwise comparison. As a result, we are able to significantly limit the search-space for duplicates, and speed up the scanning process by an order of magnitude compared to a general-purpose approach. Our contributions are:

- We analyze related work in the area of memory deduplication, by evaluating it under the lens of in-memory database systems.
- We propose a metadata-based strategy for column similarity-detection.
- We benchmark the performance-benefits of our steered approach by directly comparing it to an existing general-purpose deduplication solution.

The paper is structured as follows: First, we summarize our own previous work in the area of deduplication, which this paper seeks to expand. Then in Section 3, we characterize the state of literature in the area of in-memory data deduplication, and assess its usefulness for in-memory databases. Then, our proposed structure of a steered duplicate search in columnar databases is detailed, pointing out the unique properties of this setting, which allow for significantly faster deduplication. Subsequently, in Section 5 we describe the setup that is used to measure the concrete deduplication performance that our steered approach enables, and interpret the benchmark results. Lastly, we conclude with a short summary and propose possible future research.

2. De^{2}Dup

This section recaps our previous work in the space of data deduplication [3], and contextualize our current contribution. Our past work is based on the same premise as Section 1, in that multi-tenant databases may contain a significant amount of similar data, e.g., from copies of the same base-dataset being edited by different tenants. To reduce these redundancies, we proposed an extension to current database architectures, adding the De^{2}Dup manager component. Its task is to choose columns with potentially similar data, and to start the background-execution of their deduplication. Columns were

chosen as the granularity for orchestration, since they contain continuous ranges of memory, and have significant metadata associated with them.

As the deduplication executor, we used the Data Streaming Accelerator (DSA), an on-chip hardware-accelerator included in Intel's recent Xeon server chips [4]. It supports a specialized `create_delta` operation, generating a delta between two areas of memory. Using this operation, we showed that significant memory savings are possible when delta deduplicating similar but not identical data, compared to deduplication methods based on binary-equality. Furthermore, by offloading the operation to a hardware-accelerator, CPU resources are freed up for query execution.

Subsequently, we used the delta-deduplicated data to measure its impact on query processing performance. We compared two delta-reconstruction methods: *Page Reconstruction* (PR) re-materializes the delta deduplicated column on the granularity of memory pages, by applying the deltas onto a copy of the base page, and then uses this recreated page in normal query execution. Through benchmarks, we saw that this model suffers from scalability issues with an increasing number of page reconstructions, even if there are few changes per page. Our second method is *Delta Weaving* (DW), in which we redesign our physical query operators to allow processing the base column and the corresponding deltas at the same time, by alternating between them. For reasonable numbers of deltas, this approach performed very well, with leeway to improve in the future.

We want to note that, since our last contribution, the delta functionalities of the DSA have been deprecated by Intel, and even disabled through a microcode update due to potential system instabilities in its current implementation. Nonetheless, we see a huge importance in continuing this line of research, since the most recent chips from Intel incorporate even more accelerators along the DSA (e.g., In-Memory Analytics Accelerator (IAA), Quick Assist Technology (QAT), Dynamic Load Balancer (DLB)), that hold potential for accelerating such workloads [4]. Furthermore, by presenting our results, we hope to explore which features the next generation of hardware accelerators could support.

Based on our past results, we see De^{2}Dup as a promising mechanism for memory-efficient databases, with many more avenues to explore in the future. One such area left for future research is defining the concrete workings of the De^{2}Dup manager, and how it can find column pairs to deduplicate. This will be the focus of our current contribution.

3. Related Work

In this section, we highlight existing approaches to in-memory deduplication, first specifically in the context of databases, and then for general-purpose solutions. With this, we will show how current solutions do not match the requirements for in-memory database deduplication for analytical workloads.

For many years, columnar databases have employed **delta-encoding** for sorted columns, encoding each value as the numeric difference to the previous value [5]. Combined with bit-packing techniques, this can be used to effectively compress columnar data in a lightweight way [6]. Note that these "deltas" are distinct from those referred to in the rest of this paper, since they are the logical result of numeric subtraction, instead of a byte-by-byte difference. Though this technique is able to reduce redundancies, it is limited to a local scope, and unable to deduplicate across column or schema boundaries. In **dbDedup** [7], hash fingerprints are used to find similar data in a NoSQL database, and deduplicate it by calculating a byte-by-byte delta. In this case, the stored values are mostly very large strings (e.g., the content of a Wikipedia article), and workloads focused on point-queries, so applying the deduplication on the sub-value-granularity proved effective. For our use-case of analytical databases, where there tend to be many smaller values and queries require large table scans, this granularity would prove to be problematic. We believe that a delta on the granularity of the value being stored (e.g. 64-bit integers) is a much better fit for such use-cases.

Beyond the area of databases, memory deduplication is a prominent research topic in the virtualization space, with many focusing on consolidating more Virtual Machines (VMs) onto the same hardware [8, 9, 10]. The basic premise is that there are arbitrary "guest" processes running in a virtualized sandbox on a host machine. These processes are anonymous, and the host has no semantic knowledge about

the content of their memory allocations. Therefore, techniques must be developed to efficiently find redundancies despite this lack of semantic knowledge. We see this research as being highly relevant to our use-case, since it acts as the baseline for what potential gains are achievable through deduplication in a general-purpose setting.

In [9], Veni and Bhanu broadly classify VM deduplication techniques into three categories: Conventional OS techniques, Para-virtualization-based techniques, and Memory scanning-based techniques. *Conventional OS techniques* build upon concepts from the Operating System space, such as *semantic sharing* and *forks*. Based on a common lineage of data, these enable multiple processes to share read-only references to the same physical memory [11]. These concepts have been applied to VMs, leveraging knowledge about the used base-image to "fork" off multiple versions of a VM [12]. In the space of databases, these concepts are used by Dageville et al. [13] and Aguilar Saborit et al. [14] to efficiently clone existing database tables and schemas by only copying the metadata referencing the actual stored data. However, since these concepts only work if data has a common lineage, they are not a good fit for our scenario of isolated tenants. Furthermore, any later changes to said data will never be deduplicated, even if redundancies exist, making it less effective for long-lived database processes.

Para-virtualization-based techniques include any systems which require modification of the guest OS to function. As a consequence, the guest OS is able to communicate semantic information about its memory to the host, leading to more deduplication possibilities [10, 15]. However, having to modify a (possibly closed-source) guest OS or application limits its applicability. For our work, the issue of communicating semantics between tenant's schemas and the De^{2}Dup manager does not exist, since it has full access to table metadata. This enables possibilities for steering the De^{2}Dup process, which are impossible in the generic VM space.

Memory scanning-based techniques rely neither on common lineage nor communicated semantics. By periodically scanning all VMs memory regions, they are able to not only find redundancies with no further semantic knowledge about their origin, but also adapt to data changes over time. The **Kernel Samepage Merging (KSM)** feature of the Linux kernel [16] is an example of such a scanning-based technique. As a background process, it searches a subset of the system's anonymous memory pages for duplicates, and replaces any pair of matches with a single Copy on Write (CoW) page. To keep the number of page comparisons low, KSM sorts all scanned pages into a tree data structure based on their content. For any subsequent page, a simple look-up can confirm if duplicate content has already been inserted. Though being completely application-agnostic makes KSM a very versatile tool, it also adds a lot of complexity in its implementation. KSM must keep track of all mergeable pages in central data structures, make sure pages aren't frequently changing, and perform frequent rescans to detect new redundancies. Nevertheless, we choose KSM as the baseline deduplication implementation, since it is the most widely accessible deduplication tool. **MDedup++** [9] extends KSM by steering it away from pages which frequently change, leading to fewer CoW pages being copied. Furthermore, when a matching page-pair is deduplicated, it prioritizes the comparison of the two logical next pages in continuous memory, leading to much faster scan times depending on the types of allocations of the workload. Especially for columnar databases, this exploitation of the spatial ordering of pages is highly relevant, since large buffers of continuous values are common. **Difference Engine** [2] similarly keeps track of frequently changing pages through heuristics, and applies different deduplication techniques accordingly. Pages which were not recently *written to* are considered for sharing, while those not recently *read* can be either delta-deduplicated or compressed. For delta-deduplication to be effective, a similarity-based search process based on hash fingerprinting is introduced. Through this, the system is able to efficiently deduplicate near-identical pages, further lowering memory usage. Lastly, **XLH** [8] focuses on improving KSM's reaction time to new deduplication opportunities. By hooking into the virtual file system layer of the VM, any memory pages which recently read from a file can be identified and prioritized by KSM. As a result, short-lived sharing opportunities are detected more quickly.

From all of these approaches, we can conclude that: (1) general purpose scanning approaches need to deal with frequently changing data, (2) deduplication can be effectively steered based on data fingerprinting and semantic metadata, and (3) depending on the workload, scanning continuous memory space can lower deduplication time. Based on these conclusions about VM deduplication, we argue

that in-memory columnar databases can benefit greatly from the existing research. Since columnar databases often exhibit immutability guarantees in regard to their stored data, contain a plethora of metadata about all tables, and extensively use continuous memory regions, many of the mentioned heuristics are not necessary. Instead, a deeply integrated module can be designed, which exploits these architectural guarantees and metadata to perform a fast, low-overhead scan. To the best of our knowledge, this makes our deduplication approach the first to exploit this semantic knowledge for deduplication in databases.

4. Steered Duplicate Search

In this section, we want to introduce a concrete design for the De^{2}Dup manager component, tasked with steering our delta deduplication towards similar data.

We assume the following setting: Consider an in-memory columnar database hosting a large number of tables owned by different tenants, with a subset of those tables containing partially duplicate data. When data is ingested, it is organized into a columnar storage format, which is potentially further subdivided into a set of partitions, as is common in modern columnar databases [17]. The expressions used for partitioning are explicitly set during table creation. Additionally, lightweight statistics about the values contained in each column and partition are collected, including min/max values and value/null-counts. The De^{2}Dup manager has access to all information regarding each tenant's schema, table ordering/partitioning, statistics, and modification timestamps via a central metadata store (visualized in Figure 1).

Based on this setting, our overarching goal is to efficiently find pairs of columns from different tables which are as similar as possible — no matter if they originate from different tenant's schemas, or from the same schema. Generating such pairings based on database schemas and the data they include is already a broad research topic called *Schema Matching* (SM) [18]. Some of the main application areas of SM are the mapping of columns from a data source to those of an existing data warehouse structure (data integration), and autofilling forms in web-browsers [18]. For these use cases, many approaches have been proposed, utilizing similarity metrics based on table properties, the semantics of column names (e.g., via word embeddings), min/max values, and statistical distributions [19, 18, 20]. However, we believe our use-case to be distinct from existing solutions for several reasons. Firstly, in SM approaches, it is often desired to find a single definitive mapping between two schemas, since this is what is needed in most of its use-cases [21]. In contrast, we are attempting to find as many similar columns as possible across a large range of schemas. Moreover, our setting exhibits a large tolerance to false-positives, since a mismatch will merely cause a quickly aborted table scan, instead of incorrect data mappings. Lastly, SM approaches deal with the difficulties of mapping schemas from completely different systems, while ours is more homogenous. For example, usually naming schemes might be completely different across systems, leading to the need to match columns like `postal_code` and `ZIP` together based on semantics. In our case, many examples of duplicate data stem from a common dataset (e.g., database table/dump, parquet file), which contain consistent table/column names. These names can of course be changed, e.g., during data integration, or to align with internal naming standards, though this will often be purely syntactical. For instance, the table `currencies` may be ingested as `dim_currency`, snapshotted as `currencies_20250715`, or further processed into `currencies_cleaned`. For these reasons, we opt to employ an adapted three-step process to find similar column pairs.

Column Search First, for each column in our database, we generate a ranked list of N similar columns. To achieve this, each column is represented by its features in the following structure: *(fully_qualified_column_name, count_values, count_nulls, min, max)*. A distance between two of these structures is calculated using a weighted sum of all the individual distances between their elements. For string types, we employ the Levenshtein distance to capture syntactic similarity, and for numeric types simply the absolute difference. To find the N closest matching columns, we enumerate the distances to all other columns of the same data-type and save the N best results.

Partition Alignment Next, for each pair of similar columns, we further match up all partitions from both in pairs, based on their partition keys. Splitting the process up by partition ensures that insertions in a subset of partitions (e.g., appending new data for a certain month) do not misalign the further comparison process. However, a column could also consist of a single partition, which allows skipping this step, since the alignment is inherently given.

Delta Scan Lastly, for each pair of partitions found, we submit a set of delta operations to the DSA to perform the comparison on a granularity of 4 KiB sized memory pages. These operations will encounter one of three scenarios during the page scan: In the first case, the found pages are exactly equal, in which case no deltas are generated, one of the pages becomes dereferenced and is freed. On the other end of the spectrum, the found pages could be a complete mismatch. This can be due to the column search yielding a false-positive, or because the page has been significantly updated from its original form. In this case, the submitted delta operation returns early as soon as it reaches a configured threshold of deltas, and the page pair is not deduplicated. Using this early return as a heuristic, we can additionally abort any such false-positive matching after the first few pages of the partition, saving additional processing power. Lastly, if the pages are mostly equivalent, with only a few differences, the resulting deltas are stored, and the underlying page is freed. At this point, when a delta deduplicated page is touched by a query, the content needs to be reconstructed by either PR or DW, which happens transparently for the user [3]. To decide which page should be used as the baseline, and which should be encoded as deltas, we choose whichever partition has remained unedited for longer based on modification timestamps. We argue that more ”stable” pages are a better suited candidate to serve as a baseline for delta generation, since any change to the base page would force all deltas referencing it to be updated.

Data Changes In the case of KSM, pages are marked as CoW, i.e. a single update nullifies the saving effect of a complete page. In our case, we can approach this issue threefold. One possibility is to employ PR one any update, insert the update and consequently perform a new run of De²Dup. However, this approach is rather expensive, as it requires significant computation time from the DSA. The second approach is analogous to DW: We probe the original page at the updated position and if the updated value differs from the original value, we create a new delta or update a delta, if one already existed for this position. Thirdly, we could reconstruct the page on the first update and only apply De^{2}Dup after a certain time, e.g., after a transaction ends or after a certain amount of time has passed.

Parameter Selection Our approach is configurable in three ways. The number of most similar columns N to find for each candidate column can be set based on the amount of computing power we are willing to spend on delta deduplication. With $N = 1$, we attempt delta deduplication with only the best matching column, while with higher N, more and more opportunities can be explored. Next, the matching itself works based on a weighted sum of the distances between all features of two columns. By setting these weights ω, we are able to grant higher importance to matching statistics, or to matching naming schemes. Concrete weights will depend on the properties of the given schema, and could best be determined based on tests with a real-world dataset. Such a dataset is currently not available to us, so the weights used in the following evaluation can be chosen freely, since matching column pairs will be exactly equal on all features. Lastly, the maximum number of deltas which should be generated before the DSA aborts the operation can be configured. As we have shown before, above a threshold of 80% of two pages being different, the generated deltas consume more space than the original page [3], due to the overhead of indexes stored in the delta representation. Furthermore, processing large deltas does hurt query performance [3], so a threshold of 25% deltas is likely a good starting point. Again, this storage savings/performance tradeoff should be made based on real-world data the system will run on.

5. Evaluation

This section evaluates the applicability of our steered approach and how our tailored column selection performs against KSM. Since KSM is a kernel background task, we cannot measure distinct components like the identification of matching pages or the isolated deduplication task. Hence, we evaluate both KSM and our approach based on their effective savings and runtime in an end-to-end experiment.

Our steering approach is supplied with pre-existing matching features for all tenant's columns in our test datasets. This includes all semantic information and statistics listed in Section 4. Based on this data, each column in the dataset is matched up with another column deemed most similar. Subsequently, work-descriptors are dispatched to the DSA, generating deltas for the two columns [3]. The total memory savings of this method are measured as the number of pages processed by it, minus the additional space used to store delta records. Our time measurements begin with the start of the matching process, and end when the last page has been scanned by the DSA.

To evaluate KSM, we configure our environment in such a way, that the only buffers available for KSM to scan are our datasets, and no other system processes exposes any memory marked as "mergeable". Additionally, we set KSM's configuration variables to utilize 100% of a single CPU core (advisor_max_cpu = 100), not sleep in-between scans (sleep_millisecs = 0), and process all pages of the dataset during a single scan (e.g., pages_to_scan = 2700000 for a 10 GiB dataset)[1]. Note that the convergence speed of KSM is heavily influenced by these settings, and the reported numbers should thus be considered as a possible lower bound of its runtime. In real world scenarios, it would likely be higher due to its nature as an unobtrusive background process. We can measure the time KSM took to process the entire dataset between enabling the tool, and it's reporting of no pages left to scan. With these steps, we are able to treat KSM as if it were an on-line foreground process running on a single core. Its achieved memory savings are reported through a Linux sysfs interface, which we monitor.

Next, we specify the setting both approaches will execute in. In our testing-setup, one of two datasets is loaded into memory: The Internet Movie Data Base[2] (IMDB) dataset, used in the Join Order Benchmark (JOB) [22], consists of 21 tables comprised of a total of 109 columns. Secondly, the Star Schema Benchmark (SSB) dataset, with a scale-factor of 10 [23], which contains 5 tables and 58 columns. Both dataset's columns are separately encoded in either the Apache Arrow [24], or Apache Parquet format [25]. This choice was made to represent data encoded in a typical format for in-memory databases (Arrow), as well as a format representing compressed in-memory data (Parquet). Consequently, we have four possible setups: IMDB Arrow (6.0 GiB), IMDB Parquet (857.8 MiB), SSB Arrow (9.5 GiB), and SSB Parquet (1.3 GiB). To emulate another tenant with duplicate data being present, we additionally create an exact copy of the given dataset, and register a second set of column features for our matching algorithm. To show the effect of slowly diverging datasets on both methods, we additionally have the option of injecting the copied dataset with an increasing amount of (completely) modified pages.

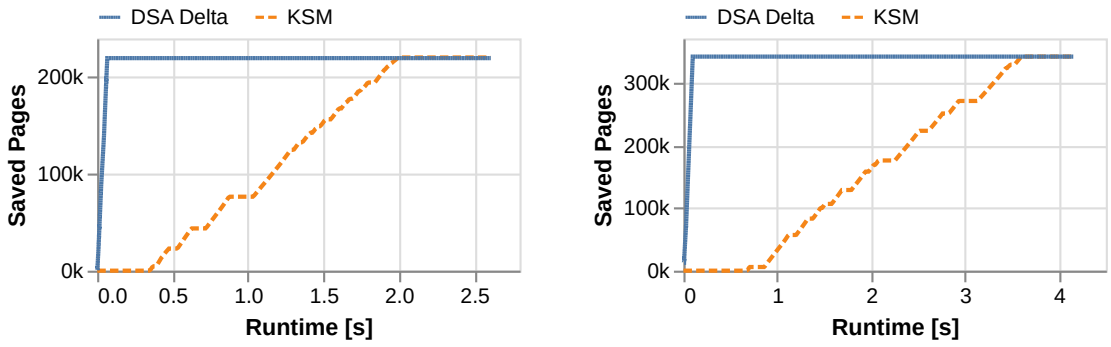
Hardware and Software

All benchmarks are executed on an Intel Xeon w5-3425 CPU, running Ubuntu 24.04.2 LTS with kernel version 6.8.0-60-generic and microcode version `0x2b000639`.

Schema Matching

First, we present the runtime of our column matching algorithm in an isolated experiment. Figure 3 shows the runtime for matching performed on our duplicated tenant schemas. As outlined in Section 4, we compare fully qualified column names (FQCN) and column statistics to determine possible candidates, which ultimately leads to an $\mathcal{O}(n^2)$ complexity, as every column is probed against every other. However, given the small number of columns that must be compared when either dataset is present in isolation, the runtime is sufficiently small to be considered negligible. The scaling becomes an obvious issue if

[1] https://docs.kernel.org/admin-guide/mm/ksm.html
[2] https://imdb.com



(a) IMDB (b) SSB

Figure 2: Cumulative pages saved over time, datasets are Parquet-encoded

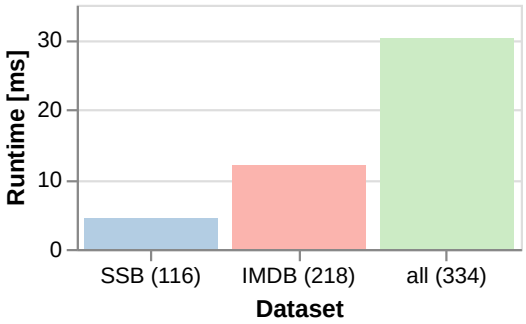


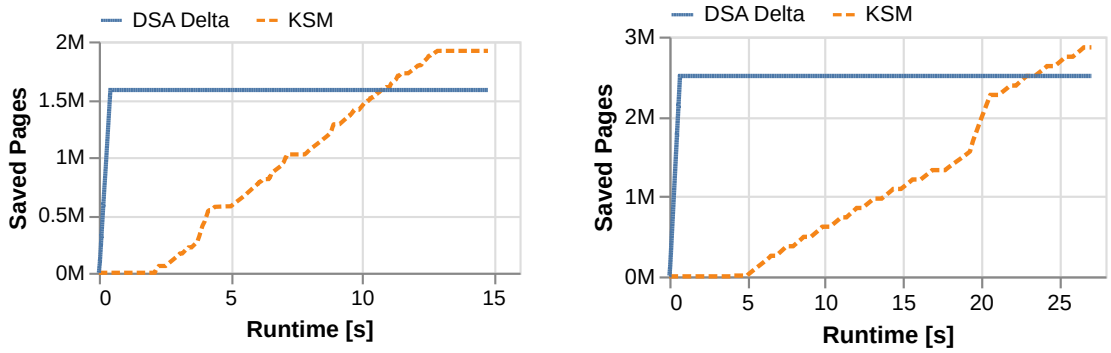
Figure 3: Runtime of column matching for both tenant's datasets and their union (number of columns in parentheses)

both datasets are ingested and all resulting FQCNs are checked across dataset boundaries. This issue stands despite the actual runtime being still quite low with approx, 30 ms. Potential optimizations are, e.g., using data-structures optimized for efficient kNN queries in many dimensions, or more aggressive pruning of matches based on very large statistical differences.

Steered Delta Deduplication

Next, we investigate the runtime behavior of both KSM and our steered approach. Figure 2a visualizes the number of duplicate pages found over the scan's runtime in the two copies of the `IMDB Parquet` dataset. The steered scan approach exhibits a very short and consistent execution, quickly detecting all 857 MiB of duplicate pages across both tenants. KSM behaves much differently during its scan-process, clearly showing when it performs its initial scan (flat section), and when it subsequently starts deduplicating in the second pass. The same behavior is also visible in the larger `SSB Parquet` dataset in Figure 2b, with the steered scanning approach finishing 36x faster than KSM.

The same scans were also executed on the `IMDB Arrow` and `SSB Arrow` dataset, visualized in Figure 4a and Figure 4b. Though KSM still takes significantly longer to fully scan the memory region, it is able to find 21.4% (IMDB) and 14.2% (SSB) more redundant pages than our steered scan. This indicates that the read-optimized Apache Arrow format contains a lot of intra-table redundancies, which KSM can detect by considering all combinations of pages as potential sharing partners. This experiment underlines the different nature of both approaches. We only consider corresponding columns to perform delta deduplication, at a small cost to the deduplication ratio. On the other hand, KSM performs a thorough scan of the system, which manages to reach the maximum possible deduplication ratio for each dataset. One possible enhancement is embracing the holistic strategies of other general-purpose deduplication solutions, like content-based similarity-fingerprinting of pages. This might allow for savings beyond the strict pairwise column search, while still benefitting from the steerability of our matching.



(a) IMDB (b) SSB

Figure 4: Cumulative pages saved over time, datasets are Arrow-encoded

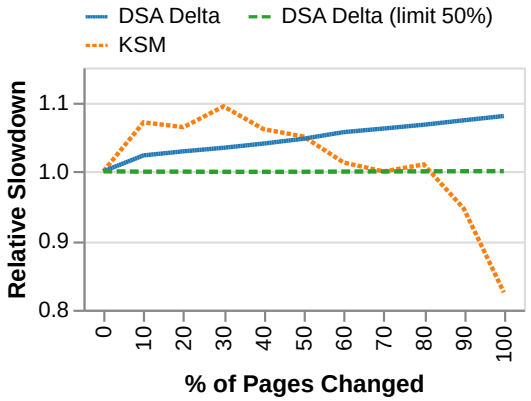


Figure 5: Performance impact of fewer matching pages in the IMDB Arrow dataset.

False Positive Candidates

False positive FQCN matches can occur if two columns exhibit syntactically similar but semantically different names or if the content of one column has been heavily modified, in relation to its origin. In these cases, the work needing to be performed to attempt delta deduplication of a column is higher than for exact matches, since more deltas are written. In Figure 5, the impact of increasingly more mismatching pages on KSM and De^{2}Dup is shown. Clearly, the extra work performed by the DSA manifests itself in slightly slower execution: in the worst case, it finishes the scan 9% slower than deduplicating a good match. However, this slowdown can be completely negated by configuring the delta operations to short-circuit if two pages differ too much, in this case 50%. Conversely to our approach, KSM benefits from very few matching pages in the dataset, likely due to fewer page merges having to be performed, and improving lookup-times for its, now smaller, tree-based data-structures. Ultimately, this experiment highlights the resilience of delta-creation towards false-positive matches. Despite KSM improving its own performance when there are fewer savings opportunities, in absolute terms, it still finishes 24.5x slower than the steered scan.

6. Conclusion and Future Work

In this paper, we expanded on our previous work [3] by introducing a strategy for finding potentially redundant data in multi-tenant database systems. We showed that using a steered search, we are able to outperform existing general-purpose solutions in terms of scanning speed, even in cases where steering yields false-positive matches. There are many more optimizations to be made, such as more scalable column-matching, better heuristics for early returns on the column level, and more resilience to diverse types of column edits besides updates and appends. Taking inspiration from related work,

the use of hash fingerprinting to detect similar content should be investigated as well [2]. However, to further test our steering approach, a suitable dataset representing tables in multi-tenant databases must first be created. Furthermore, due to the deprecation of the DSA's delta operations, future work may focus on creating an independent delta format, optimized for columnar storage. In terms of hardware-acceleration, Intel has shown that it's on-chip integration of specialized accelerators opens up possibilities for low-latency, energy-efficient offloading of workloads [4]. Especially worth further exploring is the In-Memory Analytics Accelerator (IAA), with initial research suggesting its usefulness in columnar databases [26].

Acknowledgments

This work was partly funded by (1) the German Research Foundation (DFG) priority program SPP 2377 under grant no. LE 1416/30-1, (2) the European Union's Horizon research and innovation program under grant agreement no. 101189551 (CHORYS), and (3) the German Research Foundation (DFG) via a Reinhart Koselleck-Project (LE-1416/28-1).

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] D. Jacobs, S. Aulbach, Ruminations on Multi-Tenant Databases, in: Datenbanksysteme in Business, Technologie und Web (BTW 2007) – 12. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Gesellschaft für Informatik e. V., 2007, pp. 514–521. URL: https://dl.gi.de/handle/20.500.12116/31821.

[2] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, A. Vahdat, Difference engine: Harnessing memory redundancy in virtual machines, Commun. ACM 53 (2010) 85–93. URL: https://dl.acm.org/doi/10.1145/1831407.1831429. doi:10.1145/1831407.1831429.

[3] A. Krause, J. Kowalick, J. Pietrzyk, D. Habich, W. Lehner, De^{2}Dup: Extended Deduplication for Multi-Tenant Databases, in: Proceedings of the 21st International Workshop on Data Management on New Hardware, DaMoN '25, Association for Computing Machinery, New York, NY, USA, 2025, pp. 1–9. URL: https://dl.acm.org/doi/10.1145/3736227.3736236. doi:10.1145/3736227.3736236.

[4] Y. Yuan, R. Wang, N. Ranganathan, N. Rao, S. Kumar, P. Lantz, V. Sanjeepan, J. Cabrera, A. Kwatra, R. Sankaran, I. Jeong, N. S. Kim, Intel Accelerators Ecosystem: An SoC-Oriented Perspective : Industry Product, in: 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), 2024, pp. 848–862. URL: https://ieeexplore.ieee.org/abstract/document/10609705. doi:10.1109/ISCA59077.2024.00066.

[5] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, S. Zdonik, C-store: A column-oriented DBMS, in: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05, VLDB Endowment, Trondheim, Norway, 2005, pp. 553–564.

[6] D. Lemire, L. Boytsov, Decoding billions of integers per second through vectorization, Software: Practice and Experience 45 (2015) 1–29. URL: http://arxiv.org/abs/1209.2137. doi:10.1002/spe.2203. arXiv:1209.2137.

[7] L. Xu, A. Pavlo, S. Sengupta, G. R. Ganger, Online Deduplication for Databases, in: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1355–1368. URL: https://dl.acm.org/doi/10.1145/3035918.3035938. doi:10.1145/3035918.3035938.

[8] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, F. Bellosa, {XLH}: More Effective Memory Deduplication Scanners Through Cross-layer Hints, in: 2013 USENIX Annual Technical Con-

ference (USENIX ATC 13), 2013, pp. 279–290. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/miller.

[9] T. Veni, S. M. S. Bhanu, MDedup++: Exploiting Temporal and Spatial Page-Sharing Behaviors for Memory Deduplication Enhancement, The Computer Journal 59 (2016) 353–370. URL: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/bxu149. doi:10.1093/comjnl/bxu149.

[10] G. Miłós, D. G. Murray, S. Hand, M. A. Fetterman, Satori: Enlightened page sharing, in: Proceedings of the 2009 Conference on USENIX Annual Technical Conference, USENIX'09, USENIX Association, USA, 2009, p. 1. URL: https://dl.acm.org/doi/10.5555/1855807.1855808. doi:10.5555/1855807.1855808.

[11] A. Silberschatz, P. B. Galvin, G. Gagne, Operating System Concepts, 10th edition ed., Wiley, Hoboken, NJ, 2018.

[12] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, S. Savage, Scalability, fidelity, and containment in the potemkin virtual honeyfarm, SIGOPS Oper. Syst. Rev. 39 (2005) 148–162. URL: https://doi.org/10.1145/1095809.1095825. doi:10.1145/1095809.1095825.

[13] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, P. Unterbrunner, The Snowflake Elastic Data Warehouse, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 215–226. URL: https://dl.acm.org/doi/10.1145/2882903.2903741. doi:10.1145/2882903.2903741.

[14] J. Aguilar Saborit, R. Ramakrishnan, K. Bocksrocker, A. Halverson, K. Kosinsky, R. O'Connor, N. Poliakova, M. Shafiei, H. M. Ansari, B. Crivat, C. Cunningham, T. Kim, P. Kon Kim, I. R. Madan, B. Matuszyk, M. Miles, S. Mohanan, C. Petculescu, E. R. Wirshing, E. Yousefi Amin Abadi, Extending Polaris to Support Transactions, in: Companion of the 2024 International Conference on Management of Data, SIGMOD '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 321–333. URL: https://dl.acm.org/doi/10.1145/3626246.3653392. doi:10.1145/3626246.3653392.

[15] W. Qiu, M. Copik, Y. Wang, A. Calotoiu, T. Hoefler, User-guided Page Merging for Memory Deduplication in Serverless Systems, in: 2023 IEEE International Conference on Big Data (BigData), 2023, pp. 159–169. URL: https://ieeexplore.ieee.org/abstract/document/10386487. doi:10.1109/BigData59044.2023.10386487.

[16] A. Arcangeli, I. Eidus, C. Wright, Increasing memory density by using KSM, Proceedings of the Linux Symposium (2009). URL: https://www.kernel.org/doc/ols/2009/ols2009-pages-19-28.pdf.

[17] L. Sun, M. J. Franklin, S. Krishnan, R. S. Xin, Fine-grained partitioning for aggressive data skipping, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 1115–1126. URL: https://dl.acm.org/doi/10.1145/2588555.2610515. doi:10.1145/2588555.2610515.

[18] P. A. Bernstein, J. Madhavan, E. Rahm, Generic schema matching, ten years later, Proc. VLDB Endow. 4 (2011) 695–701. URL: https://dl.acm.org/doi/10.14778/3402707.3402710. doi:10.14778/3402707.3402710.

[19] R. Castro Fernandez, E. Mansour, A. A. Qahtan, A. Elmagarmid, I. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, N. Tang, Seeping Semantics: Linking Datasets Using Word Embeddings for Data Discovery, in: 2018 IEEE 34th International Conference on Data Engineering (ICDE), IEEE, Paris, 2018. URL: https://ieeexplore.ieee.org/document/8509314/. doi:10.1109/icde.2018.00093.

[20] P. A. Bernstein, S. Melnik, J. E. Churchill, Incremental schema matching, in: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06, VLDB Endowment, Seoul, Korea, 2006, pp. 1167–1170. URL: https://dl.acm.org/doi/10.5555/1182635.1164235.

[21] E. Rahm, P. A. Bernstein, A survey of approaches to automatic schema matching, The VLDB Journal 10 (2001) 334–350. URL: http://link.springer.com/10.1007/s007780100057. doi:10.1007/s007780100057.

[22] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, T. Neumann, How good are query

optimizers, really?, Proc. VLDB Endow. 9 (2015) 204–215. URL: https://dl.acm.org/doi/10.14778/2850583.2850594. doi:10.14778/2850583.2850594.
[23] P. O'Neil, B. O'Neil, X. Chen, The Star Schema Benchmark (SSB) (2009).
[24] Apache Software Foundation, Apache Arrow, 2025. URL: https://arrow.apache.org, accessed: 2025-07-05.
[25] Apache Software Foundation, Apache Parquet, 2025. URL: https://parquet.apache.org, accessed: 2025-07-05.
[26] A. Baumstark, L. Martins, K.-U. Sattler, Uncore your Queries: Towards CPU-less Query Processing, in: Proceedings of the 21st International Workshop on Data Management on New Hardware, DaMoN '25, Association for Computing Machinery, New York, NY, USA, 2025, pp. 1–10. URL: https://dl.acm.org/doi/10.1145/3736227.3736243. doi:10.1145/3736227.3736243.