SmartLTM: Smart Larger-Than-Memory Storage for Hybrid Database Systems

Paulo R. P. Amora[1], **Elvis M. Teixeira**[1],
Francisco D. B. S. Praciano[1], **Javam C. Machado**[1]

[1]Laboratório de Sistemas e Bancos de Dados (LSBD)
Computer Science Dept – UFC – CEP 60440-900 – Fortaleza – CE – Brazil

{paulo.amora,elvis.teixeira,daniel.praciano,javam.machado}@lsbd.ufc.br

Abstract. *Main-memory DBMS can offer hybrid and evolving storage architectures, instead of the traditional row or column storage layouts. Even if RAM is affordable nowadays, it is still a limited resource concerning available storage space in comparison to conventional storage devices. Due to this space restriction, techniques that leverage a trade-off between storage and query performance were developed and should be applied to data that is not frequently accessed or updated. This work proposes SmartLTM, a data eviction mechanism that considers the decisions previously taken by the DBMS in optimizing data storage according to query workload. We discuss how to migrate data, access it and the main differences between our approach and a row-based one. We also analyze the behavior of our solution in different storage media. Experiments show that cold data access with SmartLTM incurs an acceptable 17% of throughput loss, against 26% of the row-based one, while retrieving only half of the data to answer queries.*

1. Introduction

One of the great challenges in database management is how data should be available. As time passes by, newer data usually has more importance than older data, with a few exceptions, data becomes stale after a period of time. Current database systems setups usually work by having this new, more used data available in an OLTP database and the old, more stale data in an OLAP data warehouse, through the process of data migration and the Extract, Transform, Load mechanism. This setup brings several drawbacks, such as not having actual access to a real-time data analytics, because not all data is present in both places, and the maintenance cost of keeping two separate infrastructures may hinder some applications.

As technology improves, data processing requires more speed because organizations such as businesses or research labs acquire and accumulate data at a very rapid pace and this data must be processed into information promptly, which may be too fast for current database storage engines. This scenario creates a new kind of workload, called HTAP (Hybrid Transactional Analytic Processing), characterized by having both transactional and analytic features [Grund et al. 2010] [Kemper and Neumann 2011] [Alagiannis et al. 2014] [Appuswamy et al. 2017].

Traditional relational DBMS architectures do not handle well this kind of workload, because they adopt a fixed form of storing data contained in tables. Be it the row-based for OLTP databases, where tuples are stored contiguously, and is optimized by

design for workloads that access only a small range of tuples, but many attributes of each tuple; or column-based, where the attributes of several tuples are stored contiguously, and responds well to range and aggregation queries on single attributes. A proposed Flexible Storage Model (FSM)[Arulraj et al. 2016] aims to handle both workloads more optimally, having a mixture of both. The motivation for this FSM is to allow the retrieval of more relevant data, avoiding wasting cache space with data that won't be used in query processing. This model can be generated incrementally, using the query workload and accessed attributes as a clue on how to optimally organize and present data, making better use of upper layers of memory, such as CPU caches.

This data transformation is a costly task to perform on a slow, larger storage device, which is why the databases that execute this kind of transformation, be it a fixed one or an adaptive one, are usually main-memory databases. RAM is still a limited resource, and databases must be mindful not to overuse it, as data is also stored alongside structures like indexes and other auxiliary mechanisms.

Not all data are relevant to database users, in fact, usually, the most recent data is queried and modified, and as time passes by, those tuples become stale, except for a few attributes and only on aggregate queries. Note that this is not true for all entities in the system, for example, in a sales business setting, this behavior can be observed on orders, but not on stock warehouses, which may frequently be updated or items, that are mostly immutable, but often point queried for reads.

This skewed access pattern allows optimization concerning storage space. Data that is not being accessed, nor will be accessed, named cold data, can be moved out to larger, slower storage media, while preserving the working set, also called hot data, not to hinder transactional throughput performance. Data locality should also be transparent to the DBMS upper layers, to avoid specialized code and unnecessary overheads. Project Siberia[Eldawy et al. 2014] divides this problem into 4 categories, cold data classification, cold data storage, cold data access reduction and cold data access and migration.

This work focus on 3 of the 4 categories. It proposes a novel way to store cold data, applies techniques to avoid unnecessary cold data access and discuss how to access cold records. In summary, the main contributions of this work are:

- A cold data storage that considers the hybrid organization
- An application of techniques to avoid cold data access
- A performance study of the impact of storage media on our approach

We prototyped SmartLTM within PelotonDB[Pavlo et al. 2017], a main-memory hybrid database system and performed our evaluation using benchmarks from OLTPBench[Difallah et al. 2013]. With a reasonable amount of cold data access (50%), we find that the performance decrease is around 17% in a high-performance SSD. More experiments and results are presented further in the paper.

This paper is organized, as follows: Section 2 details SmartLTM and discuss design decisions. Section 3 explains how our approach is integrated within the DBMS. Section 4 presents the experimental evaluation. Section 5 briefly discusses related works, and we conclude in Section 6.

2. SmartLTM

2.1. Background

Instead of the traditional main-memory storage, we introduce a cold storage component inside the database architecture. The cold storage is separate from the main memory storage. Data is moved to the cold storage through a process called eviction, which can be defined as the inverse of caching. While caching keeps data frequently used in a faster medium, eviction moves data infrequently used to a slower medium. However, differently from other works that employ eviction, like Anti-caching[DeBrabant et al. 2013] and Siberia[Eldawy et al. 2014], the storage architecture is hybrid instead of row-based. One example of a hybrid storage DBMS is Peloton. In Peloton, data is organized according to the tile architecture[Arulraj et al. 2016], a specific in-memory organization to make data more available to the execution engine, and it has a few important definitions and terms that will be used throughout the article.

A table is composed of a list of tile groups, which can be seen as horizontal partitions within the table. A tile group has a fixed limit of tuples, the same schema as the table but it is composed of a disjoint set of tiles.

A tile is akin to a vertical and horizontal partition of a table, which contains a subset of the attributes as its schema, as well as only the subset of tuples of the table enclosed by the tile group. Different tile groups may have different tile layouts.

2.2. Data Eviction mechanism

Rather than doing data eviction one tuple at a time like Anti-caching, the mechanism uses a coarser granularity, evicting whole tile groups. For a tile group to be a candidate for eviction, it must not have been directly accessed, for read or write queries. Once the tile group is full, it will be accessed only for reads, however, if data that is being frequently accessed is evicted, the DBMS performance will decrease drastically.

The eviction process runs in a background thread and while the data is being written out, read transactions can still access it in memory if needed. Once data is written out, it is removed from main memory as soon as the older transactions cease to use it. It starts when a given threshold is reached.

Data is evicted in a format inspired by works like PAX[Ailamaki et al. 2002] and NoDB[Alagiannis et al. 2012]. It is written to the secondary storage in separate files, following the tile layout generated according to the workload. This approach respects the work previously done by the DBMS in organizing data in an optimal arrangement, allows for parallel retrieval of data in supporting devices, like SSDs, and preserves together data that is accessed together, reducing wasteful I/Os. To be able to skip unnecessary data, we also need to write out the column map, which maps the schema columns to the corresponding tile and offset.

Algorithm 1 details the execution of the eviction process.

2.3. Cold Storage

The cold storage is persistent storage where evicted data resides. It is decoupled from the database storage and non-transactional because it is a file. Tuples present within

Algorithm 1: Eviction Algorithm

```
Data: Tile groups in the table
Result: Tile groups evicted from the table and auxiliary structures
for tile group in table do
    if tile group is marked for eviction then
        write column map to external storage;
        create SMA for tile group;
        for tiles in tile group do
            add data to cuckoofilter;
            write tile to external storage;
        end
        delete tile group from table;
    end
end
```

expelled tile groups are read-only and not modifiable, although they can be invalidated in-memory, in case of deletion. Section 3 describes the behavior of operations with the new architecture.

2.4. Access Filters

Once data is moved to cold storage, it should be accessed only when needed, given that secondary storage operations are expensive in comparison to main memory. Some structures help to avoid unnecessary access to cold storage. Bloom Filters[Bloom 1970] is a non-deterministic structure that can tell if an element is absent or if it may be present, but here we employ Cuckoo Filters[Fan et al. 2014], an evolution of Bloom filters that is more space-efficient and supports delete operations.

Cuckoo Filters are hash-based. Therefore they only support equality comparisons. Another structure called Small Materialized Aggregates (SMA)[Moerkotte 1998], also known as Zone Maps, aids when checking for the presence of data in ranges, to avoid bringing to main memory all the data in the cold storage. It consists of precomputed aggregates (max, min) for tile groups (horizontal partitions), which can tell if a given key or range is present in that partition. There is an implementation of SMAs present in Peloton, for memory resident tile groups.

The CuckooFilters and SMAs are stored in main memory, alongside hot data. The storage space they occupy is negligible in comparison to the space saved.

2.5. Data Retrieval Mechanism

When a query is posed against the DBMS, it must try to answer the query with the memory resident data. For example, if the query asks for an exact match in a unique column. If the answer is not sufficient or not found, the cold storage must be probed, to check if there is a possibility that data present in the cold storage might answer the query. From the probe, two scenarios are possible.

If it's deemed that the cold storage cannot answer the query, then, the DBMS returns an answer to the client, saving an expensive cold access. On the other hand, the

probe returns the candidate tile groups in cold storage that may contain the data. Those candidate tile groups are then retrieved from the cold storage, but not entirely. Only the tiles containing the queried attributes are returned, as the column map links the queried columns and the correct tiles and offsets. After the retrieval, the data is validated against the Cuckoo Filter for deletes, and they are reassembled in a temporary in-memory table, that is disposed of when the transaction is completed. Algorithm 2 clarifies the data retrieval mechanism.

Algorithm 2: Data retrieval algorithm

```
Data: Columns accessed, candidate tile groups
Result: Temporary structure containing requested data
1 for tile group in candidate tile groups do
2 |   retrieve column map;
3 |   tiles = column map[columns accessed];
4 end
5 for tile in tiles do
6 |   retrieve columns accessed;
7 |   create temp table;
8 |   add retrieved tuples to temp table;
9 end
```

While data is being retrieved from the cold storage, the current transaction waits for the data. This retrieval does not bring many concurrency issues because of multi-version concurrency control.

2.6. Discussion

By dividing possibly large files into smaller ones, the I/Os become less costly and devices that allow efficient random access benefit from this. By keeping together data that is accessed together, the I/Os are not wasteful, avoiding the useless retrieval and load of data that is not necessary to the current query. The data retrieval mechanism ensures that cold access is done only when necessary. The synchronous retrieval is preferred according to [Ma et al. 2016].

3. Integration with the DBMS

Inserts. New tuples are always inserted in main memory. It is assumed that because they are new data, they will frequently be accessed and it is not for the benefit of performance to add new tuples directly in the cold storage.

Deletes. Deletes in main memory data happen as usual. When deletes happen in cold storage data, the respective entry is removed from the filter, but no access is made to the cold storage. This removal effectively makes the record inaccessible in cold data while avoiding access.

Updates. Updates with relation to cold data are nothing more than a delete operation followed by an insert. Meaning that the updated record will always be in main memory. This new version may be evicted later to cold data. Updated data is considered hot because new data is always considered hot.

Reads. Reads can be seen as two types of queries: point queries, which are reads done through an equality predicate and range queries, which use a more flexible predicate, like *less than* or *greater than*. A broader view of reads has already been presented in section 2.5. Reads are also benefited from new data being only placed in main memory. When a transaction with a read validates, it checks for changed or new data, which is already present in main memory, avoiding cold storage accesses.
Point queries. Point queries are first posed against the hot data. If the predicate is a primary key or unique, the query may be answered only by looking at data present in main memory. If it is not completely answered, the predicate is evaluated by the CuckooFilter, which can tell if the data is not present in the cold storage. In the end, if the probe determines that data may be present in the cold storage, we move to data retrieval.
Range queries. Range queries are first posed against the hot data. If it is not completely answered, data may be present in the cold storage, which is probed using the SMAs, since they are suitable for ranges, we move to data retrieval.

4. Experimental evaluation

To evaluate our strategy, we prototyped it in Peloton, a hybrid, main-memory, multi-versioned DBMS. Besides adding the new components, a few changes were made in the engine to integrate the new components with the query processing engine. The logging and garbage collection components were disabled to avoid interference with other secondary storage media and ensure tile group immutability.

4.1. Setup

The experiments were executed in an Intel Core I7 7800X with 6 physical cores and hyperthreading, with 64GB of RAM and 8MB of L3 cache with Ubuntu 16.04 LTS as the OS. Two different storage devices were selected as cold storage, a commodity WD Blue SATA 3 7200rpm HDD and an Intel DC P3600 SSD. The HDD has a reported IOPS of 500 and the SSD 230000 for random read operations. To ensure maximum parallelism and avoid interference from context switch, the number of threads executing queries against the database is purposely low.

4.2. Benchmarks

The benchmark selected is YCSB[Cooper et al. 2010], due to the easiness of keeping track of operations. We used OLTPBench to execute the benchmark but modified some operations. While we kept the semantics of the key uniqueness, the primary key constraint was disabled, and no indexes whatsoever were created. The queries were also modified to diversify attribute access. Five queries selecting some columns were placed and randomly selected alongside the values, to allow a better data layout organization and effectively test the different organizations. They are shown below, with the layout organization after execution:

- Q_1: `SELECT f0 WHERE KEY =` X`;`
- Q_1: `[KEY][f0][f1, f2, f3, f4, f5, f6, f7, f8, f9]`
- Q_2: `SELECT f2, f4 WHERE KEY =` X`;`
- Q_2: `[KEY][f0, f1][f2, f3, f4][f5, f6, f7, f8, f9]`
- Q_3: `SELECT f1, f2, f3 WHERE KEY =` X`;`
- Q_3: `[KEY][f0][f1, f2, f3][f4, f5, f6, f7, f8, f9]`

- Q_4: `SELECT f1, f2, f6, f7 WHERE KEY =` X`;`
- Q_4: `[KEY][f0][f1, f2][f3, f4, f5][f6, f7][f8, f9]`
- Q_5: `SELECT f0, f1, f5, f8, f9 WHERE KEY =` X`;`
- Q_5: `[KEY][f0, f1][f2, f3, f4][f5][f6, f7][f8, f9]`

To have more control of cold and hot data accesses, we also added a uniform distribution to select key values, alongside the standard Zipfian. With a uniform distribution, cold storage access can be correctly estimated and is directly proportional to the amount of data evicted to the cold storage.

4.3. Workloads

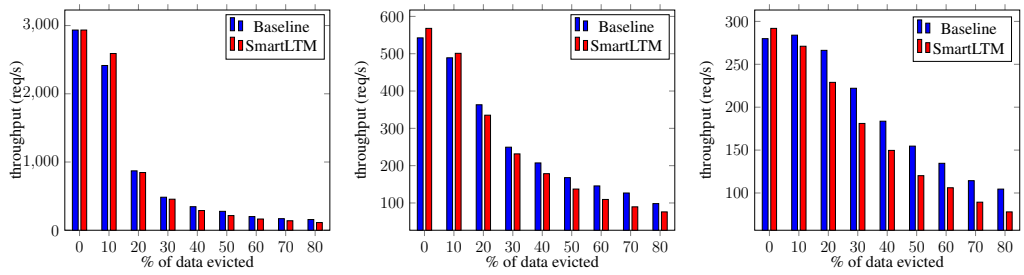
We defined four workloads to be executed, a read-only, an insert-only, a delete-only and an update-only. The scenario of data eviction to a secondary, slower storage medium shifts the burden to read queries, given that all modifying operations, like delete, insert and update, happen only in-memory, according to the mechanism proposed. Therefore, mixed workloads would only alleviate the bottleneck imposed by cold storage reads. Due to the nature of YCSB, all queried values are within the range of data loaded in the table, so there are no missed queries, and every one of them returns an answer, except in the delete workload.

4.4. Experiments and Results

SmartLTM and the baseline are implemented inside Peloton. The main difference between SmartLTM and the baseline is how data is organized and how it is evicted. The baseline evicts the tile group completely, as a row-based layout, while our approach separates the tiles, as described above. Both implementations take advantage of the access filters implemented to avoid cold storage access, to ensure fairness. An effort was also made to ensure direct access to the storage media, trying to avoid buffered reads as much as possible.

4.4.1. Read-only Queries

This experiment evaluates the impact of our approach with a read-only workload. Three different scale factors in YCSB were used: 50, 250 and 500. The access pattern is uniform, to allow accuracy when estimating hot data accesses and cold data accesses.

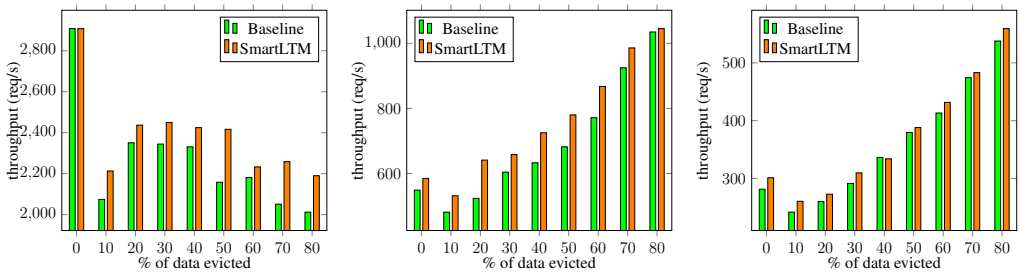


(a) 50k Tuples **(b) 250k Tuples** **(c) 500k Tuples**

Figure 1. HDD results (higher is better)

The HDD results are shown in figure 1. It is observable that the performance decrease is exponential, and that the baseline performs a little better than our proposed approach. The exponential decrease in performance is due to the access speeds of the media (HDD), which becomes the bottleneck in performance.

The baseline performs better due to the nature of an HDD device. When reading a single file which was written in neighboring sectors, the arm only needs a single spin to read all the data. In our proposed approach, the tiles are written separately, keeping the data inside them contiguous, but having no control over how different tiles of the same tile group are recorded. If different tiles are required to answer a query, the device would have to do multiple scans, and it is known that random access in an HDD is costly.



(a) 50k Tuples **(b) 250k Tuples** **(c) 500k Tuples**

Figure 2. SSD results (higher is better)

The SSD results are shown in figure 2. The higher access speeds and random access behavior of the device demonstrates how our proposed approach is an improvement over the baseline, having a throughput loss of 17% vs. 26% of the baseline on figure 2a, at the 50% evicted data mark. It is also noted that the throughput increases as more data is evicted when there is a higher volume of data loaded into the database. From section 2.5, the in-memory sequential scan has a *O(n)* complexity and always happen. As less data is present in memory, the faster this scan happens. Given the SSD device's high speeds and that the cold storage read is also a targeted one, retrieving only the tiles where the queried attributes reside, the most expensive task becomes traversing all the in-memory data to search the value. Querying the cold storage is still an expensive operation, as observed in the 50k graph, where the cold storage read is visibly hindering the performance. It is also observable that the throughput increase follows an approximately linear pattern, more clearly seen in the 500k graph.

The random access optimization of SSDs makes clear that our approach is better than the baseline, especially where it most counts when the bottleneck becomes the cold data access. It can be verified that the throughput with our proposed approach almost doubles the one in the baseline, as more data is evicted, shown in figure 2a.

4.4.2. Insert, Update and Delete queries

This experiment evaluates the impact of SmartLTM with insert, update and delete workloads. Since those operations do not care about the cold storage, only probing it through the access filters, only one scale factor in YCSB was used, 250. The access pattern is

uniform, to allow accuracy when estimating hot data accesses and cold data probes.

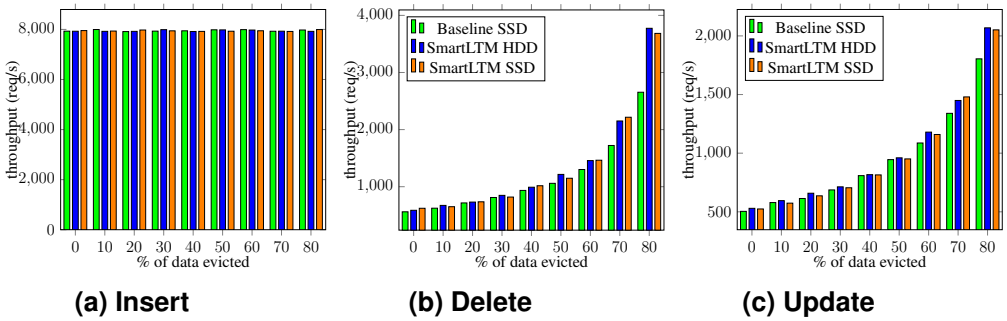


Figure 3. Insert, delete and update results

Figure 3 shows three separate results, one from an insert-only workload, that is observed to be almost constant, independent of how much data was evicted from the database. The proximity of insert results happens because inserts do not care about data that is present or absent in main memory, it only allocates a tuple and inserts the values. The update-only workload shows a throughput increase as data is evicted, clearly because, before updating, a sequential scan must happen to in-memory data. The probe in cold storage is a cheap operation because only the access filters are queried, and all the modifications are done in-memory. The delete-only workload behaves like the update-only, however, deleting a record involves fewer operations than updating, which is why the overall throughput is higher. It also can be observed that the baseline suffers on sequential scans. This behavior is an effect of the tile layout organization, which doesn't happen in the baseline.

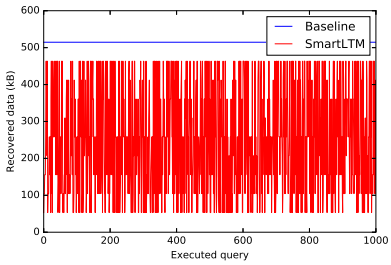
4.4.3. Retrieved data from disk

This experiment evaluates the impact of SmartLTM for retrieved data from disk. The read-only workload is executed with YCSB scale factor 250. The access pattern is uniform.

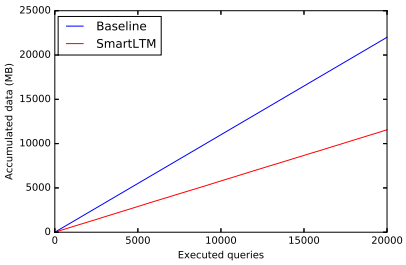
Figure 4 shows three separate results. Figures 4a and 4b are measurements taken with 50% of data evicted, and show how much data is retrieved per executed query and the evolution of how much data is retrieved during the benchmark execution. Figure 4c summarizes how much data was retrieved from disk in each eviction percentage and compares it to the baseline. Baseline retrieves a fixed amount because the tile group is evicted as a whole, while SmartLTM takes into account the previous organization, as described before. This experiment shows clearly that our approach is effective regarding data retrieval, by avoiding useless data to answer queries, based on the data organization.

5. Related Works

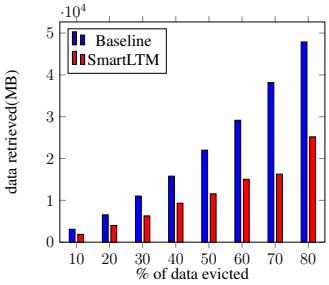
There are a few different solutions to in-memory space saving. Garbage collection (GC) is a well-known approach, adopted in several multi-version DBMS. Wu et al. [Wu et al. 2017] conduct a study of various techniques. GC by itself can save space by reusing invalid tuple slots. However, this brings an unwanted side effect of mixing hot and cold data. If a cold data partition receives one hot record, it may not be considered completely cold anymore.



(a) Data retrieved per query



(b) Accumulated data



(c) Total data retrieved

Figure 4. Data Retrieval results (lower is better)

To keep away from this side effect, the GC may not recycle used tuple slots, but be associated with another technique, called compaction, which can place data together and close any empty spaces. The challenge imposed by compaction in a hybrid storage DBMS environment is that data in different partitions may be organized in different ways, and mixing them would implicate a decision that results in some layouts being suppressed in favor of the many.

Aside from GC and compaction, compression is seen as the next step in space saving. Compressed data occupies a fraction of the original storage space, and while it introduces overhead to decompress data, everything is still present in main memory. Works like HyPer [Kemper and Neumann 2011] present an HTAP database that makes use of compression techniques to store unaccessed data better. An evolution of this storage model, called Data Blocks[Lang et al. 2016] optimizes even further the access speed to compressed data and the compression rates. Compression is the last stand when it comes to main memory storage.

However, main memory storage is finite. When data does not physically fit in main memory, secondary storage media is needed. Works like Anti-caching[DeBrabant et al. 2013] and Project Siberia[Eldawy et al. 2014] provide two different solutions.

Anti-caching tracks the tuple eviction candidates through an LRU chain, and when eviction is needed, the last members of the chain are written in a block back to disk. Evicted tuples are tracked through a specific table, containing the block id and tuple offset to evicted tuples. To access evicted data, a special pointer called a tombstone is placed when a given tuple is evicted. Transactions are processed making use of a pre-pass phase, which checks if any tombstone is accessed. If anything evicted may be accessed, the transaction aborts and a subroutine that brings the evicted tuples back to memory starts.

Then, the transaction is restarted and executes as usual. Since H-Store executes only one thread per execution node, this frees the thread to run other transactions while data is being retrieved.

Siberia tracks the eviction candidates offline by logging record accesses and sampling those logs to extract estimates of eviction candidates. Those are migrated to the cold storage when a user-defined threshold is achieved. Data retrieval from the cold storage is synchronous, and to access evicted data Bloom Filters are used.

LSM Trees [O'Neil et al. 1996] are also a data structure appropriate to larger-than-memory scenarios, but is more applied in key-value DBMS, and may not perform well in relational databases as the main storage mechanism.

Siberia and Anti-caching focus on an OLTP environment, and evict the data as tuples, while SmartLTM focuses on an HTAP environment and preserve the optimization that this kind of DBMS provides.

6. Conclusions and future work

In this work, we propose SmartLTM, a new, smarter way of executing data eviction concerning HTAP databases. Current data eviction solutions are designed for row-based DBMS and perform sub-optimally when adapted as-is to the new architecture, provided good random access secondary storage. Our experiments show that SmartLTM does not affect insert, delete and update response times, and achieve better response times while retrieving fewer data from secondary storage.

As future work, a global cache can be implemented containing the most accessed tile groups. Keeping the cold storage in modern, byte-addressable non-volatile memories (NVRAM) can also drastically improve performance since the retrieval would not need to bring useless data contained in the current storage, which is block-addressable.

Acknowledgements

This research was partially supported by FUNCAP/CE-Brazil (Grant BMD-0008-01237.01.09/17) and LSBD/UFC. I'd also like to thank Prof. Andy Pavlo and the CMU Database Group for his feedback and support during the early conceptual stages of this work.

References

Ailamaki, A., DeWitt, D. J., and Hill, M. D. (2002). Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215.

Alagiannis, I., Borovica, R., Branco, M., Idreos, S., and Ailamaki, A. (2012). Nodb: efficient query execution on raw data files - read. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 241–252.

Alagiannis, I., Idreos, S., and Ailamaki, A. (2014). H2O: a hands-free adaptive store - read. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1103–1114.

Appuswamy, R., Karpathiotakis, M., Porobic, D., and Ailamaki, A. (2017). The case for heterogeneous HTAP. In *CIDR*. www.cidrdb.org.

Arulraj, J., Pavlo, A., and Menon, P. (2016). Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 583–598.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154. ACM.

DeBrabant, J., Pavlo, A., Tu, S., Stonebraker, M., and Zdonik, S. B. (2013). Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953.

Difallah, D. E., Pavlo, A., Curino, C., and Cudré-Mauroux, P. (2013). Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288.

Eldawy, A., Levandoski, J. J., and Larson, P. (2014). Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11):931–942.

Fan, B., Andersen, D. G., Kaminsky, M., and Mitzenmacher, M. (2014). Cuckoo filter: Practically better than bloom. In *CoNEXT*, pages 75–88. ACM.

Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudré-Mauroux, P., and Madden, S. (2010). HYRISE - A main memory hybrid storage engine. *PVLDB*, 4(2):105–116.

Kemper, A. and Neumann, T. (2011). Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206.

Lang, H., Mühlbauer, T., Funke, F., Boncz, P. A., Neumann, T., and Kemper, A. (2016). Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *SIGMOD Conference*, pages 311–326. ACM.

Ma, L., Arulraj, J., Zhao, S., Pavlo, A., Dulloor, S. R., Giardino, M. J., Parkhurst, J., Gardner, J. L., Doshi, K., and Zdonik, S. B. (2016). Larger-than-memory data management on modern storage hardware for in-memory OLTP database systems. In *DaMoN*, pages 9:1–9:7. ACM.

Moerkotte, G. (1998). Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*, pages 476–487. Morgan Kaufmann.

O'Neil, P. E., Cheng, E., Gawlick, D., and O'Neil, E. J. (1996). The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385.

Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T. C., Perron, M., Quah, I., Santurkar, S., Tomasic, A., Toor, S., Aken, D. V., Wang, Z., Wu, Y., Xian, R., and Zhang, T. (2017). Self-driving database management systems. In *CIDR*. www.cidrdb.org.

Wu, Y., Arulraj, J., Lin, J., Xian, R., and Pavlo, A. (2017). An empirical evaluation of in-memory multi-version concurrency control. *PVLDB*, 10(7):781–792.