UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

PAULO ROBERTO PESSOA AMORA

SMARTLTM: LARGER-THAN-MEMORY DATABASE STORAGE FOR HYBRID DATABASE SYSTEMS

FORTALEZA
2018

PAULO ROBERTO PESSOA AMORA

SMARTLTM: LARGER-THAN-MEMORY DATABASE STORAGE FOR HYBRID DATABASE SYSTEMS

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciências da computação. Área de Concentração: Bancos de dados

Orientador: Prof. Dr. Javam de Castro Machado

FORTALEZA

2018

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A541s Amora, Paulo Roberto Pessoa.
SmartLTM: Larger-than-memory database storage for hybrid database systems / Paulo Roberto Pessoa Amora. – 2018.
69 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2018.
Orientação: Prof. Dr. Javam de Castro Machado.

1. Storage. 2. Adaptivity. 3. Self-tuning. 4. Databases. I. Título.

CDD 005

PAULO ROBERTO PESSOA AMORA

SMARTLTM: LARGER-THAN-MEMORY DATABASE STORAGE FOR HYBRID DATABASE SYSTEMS

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciências da computação. Área de Concentração: Bancos de dados

Aprovada em:

BANCA EXAMINADORA

To the researchers that were affected by the National Museum fire in Rio. That this unfortunate episode may open our eyes to the value of science.

ACKNOWLEDGEMENTS

To Prof. Dr. Javam de Castro Machado for advising me during the master's degree research and giving full support.

To Prof. Dr. Andy Pavlo for having me at CMU, and the contributions during the early stages of this work.

To the Comittee, Prof. Dr. Vanessa Braganholo and Prof. Dr. Angelo Brayner, for the time and effort spent in evaluating this dissertation and providing valuable feedback.

To our study group of DBMS at LSBD, for the valuable discussions during the weekly meetings.

To the CMU Database Group, for the warm welcome and very productive internship.

To all the colleagues of LSBD, for the insights and lessons learned during these years.

To my friend Elvis Teixeira, for pioneering with me this field of study in the lab.

To my significant other, for supporting me and listen to my complaints when things go wrong.

To my mother, for I wouldn't be who I am and be where I am without the support and values taught at all the stages of my life.

To PhD Candidate in Electrical Engineering Ednardo Moreira Rodrigues, and his assistant, Alan Batista de Oliveira, undergraduate student in Electrical Engineering, for fitting the template used in this work, by making it in accordance to the Federal University of Ceará library requirements.

To Fundação Cearense de Apoio ao Desenvolvimento (Funcap), represented by President Tarcísio Haroldo Cavalcante Pequeno for sponsoring my master's research through scholarship.

To Lenovo, for sponsoring my master's research through the Brazilian Informatics Law.

“Our greatest glory is not in never falling, but in getting up every time we do.”

(Confucius)

RESUMO

Memória de acesso randômico (RAM) é um recurso valioso em sistemas computacionais, mas com o passar do tempo, mais memória tem sido disponibilizada para estes sistemas, uma vez que seu valor de aquisição tem descrescido ao longo dos anos. SGBDs em memória podem ser projetados com uma arquitetura de armazenamento híbrido, diferentemente de layouts tradicionais de organização dos dados em registros e colunas. Apesar de sua crescente facilidade de aquisição, memória RAM ainda é um recurso limitado em espaço de armazenamento em comparação com os modernos dispositivos de armazenamento persistente. Devido à restrição de espaço das RAMs, estudos tem sido realizados para melhorar o desempenho do processamento de consultas considerando o espaço de armazenamento dos dados, em particular procurando alocar os dados utilizados com menos frequencia em locais de armazenamento de menor desempenho, abrindo espaço para ocupar a memória mais rápida com dados de uso mais frequente. Esta dissertação propoe um mecanismo de despejo de dados que considera a estrutura de armazenamento de um SGBD previamente definida como forma de otimizar o armazenamento dos dados de acordo com a carga de trabalho que lhe é submetida. Nesta dissertação discutimos como migrar, de maneira automática, os dados da memória mais rápida para a memória persistente de maior capacidade mas mais lenta, as estruturas de dados de acesso e as principais diferenças entre a nossa abordagem e aquelas que tem como base o armazenamento de registros. Nós também analizamos o comportamento da nossa estratégia para diferentes dispositivos de armazenamento. Experimentos mostraram que o acesso a dados ditos frios em nossa abordagem leva a uma perda de desempenho de apenas 17% do tempo de acesso enquanto que esta perda é de 26% em abordagens baseadas em armazenamento de registros, ao mesmo tempo em que apresentamos uma taxa de 50% de acesso aos dados em disco para responder às consultas.

Palavras-chave: Armazenamento. Adaptabilidade. Alto-tuning. Armazenamento híbrido. Bancos de dados maiores-que-a-memória

ABSTRACT

Random access memory (RAM) is a valuable resource in computer systems, but as time passes, computer systems allow for more memory and it is becoming more affordable. Main-memory DBMS can offer hybrid and evolving storage architectures, instead of the traditional row or column storage layouts. In spite of affordability, RAM 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 space and query performance were developed and, consequently, they should be applied to data that is not frequently accessed or updated. This work proposes 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 our approach incurs an acceptable 17% of throughput loss, against 26% of the row-based one, while retrieving only half of the data in average to answer queries.

Keywords: Storage. Adaptivity. Self-Tuning. Hybrid Storages. Larger-than-memory databases.

LIST OF FIGURES

LIST OF TABLES

ALGORITHMS LIST

CONTENTS

1 INTRODUCTION

As memory prices get cheaper through time, a new paradigm is developed, geared towards increasing data querying and processing speed and performance in general, by keeping the database completely in main memory, instead of in disk. Main Memory Databases (MMDB) are the representation of this paradigm shift, having the data in main memory, and developing new algorithms that take advantage of main memory characteristics.

Traditional Database Management Systems (DBMS) data storage layers are usually composed of a buffer, that resides in random access memory (RAM), and the data that is stored in a hard drive or solid state drive. However, as technology improves, RAM become cheaper, giving systems more capacity and allowing for entire databases to fit in main memory. This is an advantageous scenario to applications where data processing requires more speed and throughput given that organizations such as businesses or research labs acquire and accumulate data at a very rapid pace. Traditional DBMS were not designed or optimized to deal with the whole database existing in-memory, therefore, they miss out on many optimizations that this scenario can bring. A more detailed discussion on main memory databases can be found on chapter 2.

The usual architecture for a relational DBMS is presented on figure 1. It is composed of the following modules:

- **Network Interface** Provides a connection interface for external applications to communicate with the DBMS.
- **Planning and execution** From the query, designs execution plans, and the optimizer selects the best execution plan, based on the cost to execute different operations. Executes the operations defined in the associated plan. This engine interacts with the storage engine to retrieve data and with indexes to perform the operations stated in the plan.
- **Access methods** Contain data structures created to optimize access to data in the storage layer.
- **Data storage** Responsible for storing data and provide transparent access to it to the upper layers of the database, also contains the buffer manager, which provides faster data access.

In this work, the focus is on the bottom layer, data storage and management. 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 and with a few exceptions, data becomes stale after a period of time. As an example, take an online auction platform, such as *eBay*. A user creates an auction with an end date, let us say a week. During this time period,

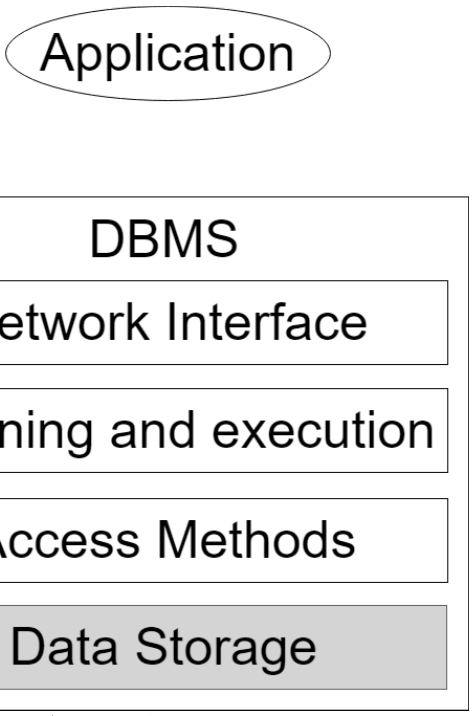


Figure 1 – Architecture of a traditional DBMS

the data corresponding to this auction will be frequently updated, because other users will place bids, ask questions and the user will answer the questions and maybe update the auction, to increase the final price or its chances of selling. After the auction period is over, a few updates related to closing the deal continue to happen, such as payment, shipping and returns or warranty. When the transaction is finished and all return and complaint windows expire, the entries in the database regarding the auction and the deal will never be changed again, only read every once in a while, now, with the purpose of collecting statistics and metrics of the platform.

1.1 Larger-than-memory

While cheap, RAM storage capacity is far from not being a concern. Computer setups that allow for huge amounts of RAM are specific and very expensive. Meanwhile, data can grow faster than in-memory processing speed, causing the need to either give up on the processing optimization of a main-memory database, or develop a solution which can handle data growing larger than memory size.

Given the limitation of RAM, strategies must be adopted to ensure a good use of storage space available. Not all data are relevant to database users, in fact, usually the most recent data are 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 be updated frequently or items, that are mostly immutable, but frequently point queried for reads.

This skewed access pattern allows optimization with respect to 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, avoiding hinder transactional throughput performance. Note that the data that is moved out of memory is still available to the database process, this is not a backup process, instead, it can be seen as an offloading process. Data locality usually is 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, that answer the following questions:

- **Cold data classification.** Which data should be moved to secondary storage?
- **Cold data storage.** How this data should be stored in secondary storage?
- **Cold data access reduction.** What can be done to ensure that the secondary storage is accessed only when necessary?
- **Cold data access and migration.** When and how this data should be migrated and how access should be done?

From these four categories, this work focus on 3. It proposes a novel way to store cold data, considering data layout and organization; applies techniques to avoid unnecessary cold data access and discusses how to access cold records.

1.2 Hybrid store databases

As data grows, the performance decreases and processing times become slower. For businesses, usually the newer, recent data holds more value than data from months or years, which will be used in an aggregated form, for analytic purposes. Current Database Management Systems (DBMS) setups usually work by having this new, more used data available in a traditional, row-based database and the old, more stale data in a different, specialized data warehouse having a different DBMS, specialized for analytics, through the process of data migration and the Extract, Transform, Load (ETL) mechanism. This setup brings several drawbacks, as, for example, not having actual access to a real-time data analytics scenario, because not all data is present in both places. Other disadvantages include the maintenance cost of keeping two separate infrastructures and not having all data accessible at any time, which may hinder some applications.

As the applications evolve, the workload also evolves, having operations that fit both OLTP and OLAP types. These workloads are called Hybrid Transaction Analytical Processing (HTAP), characterized by having both transactional and analytic features.

Traditional relational DBMS architectures do not handle this kind of workload well, because they adopt a fixed form of storing data contained in tables. Be this form rows 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 columns, 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.*, 2016a) aims to handle both workloads in a more optimal way, combining both the row-storage for tuples that are accessed with OLTP-like workloads and a column-store for data accessed as OLAP. The motivation for this FSM is to allow the retrieval of more relevant data, avoiding wasting cache space with data that will not 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.

Other works that deal with the hybrid workload and storage model are (GRUND *et al.*, 2010) (KEMPER; NEUMANN, 2011) (ALAGIANNIS *et al.*, 2014) (APPUSWAMY *et al.*, 2017).

The data layout transformation is a very expensive task to perform on a slow, larger storage device, which is why the database systems that perform this kind of transformation, be it a fixed one or an adaptive one, are usually main-memory databases. Although the price of RAM has been decreasing over the years, it is an expensive resource in comparison to larger and slower devices, like HDDs. Therefore, it is a more limited resource, and databases must be mindful to not overuse it, as data is also stored alongside structures like indexes and other auxiliary mechanisms.

1.3 Contributions and text organization

The hypothesis defended by this dissertation is that, in a larger-than-memory scenario, by considering the data organization provided by the DBMS, based on the access patterns of the workload, data should be evicted and stored in an optimal hybrid fashion, as it was when it was in memory, instead of a fixed row or column store. The question of research is how can a

DBMS autonomously evict part of a database that is in memory, while keeping it available in an optimal way. A separate storage, named cold storage, keeps data that is not being currently used and accessed. The results are evaluated by throughput comparison, as well as how much data is retrieved from the cold storage. While the focus of this work is not on which data must be evicted, but how this data should be evicted and retrieved, a brief discussion on the characterization is also made. However, we start from the premise that the eviction candidates were already selected. In summary, the main contributions of this work are:

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

As a result of the development and evaluation of this hypothesis, the papers in table 1 were submitted and published to the scientific community. The first paper in the list below entirely describes this work and I acted as its main author. I also directly contributed to the others, but they resulted from joint research with other colleagues at the DBMS group of LSBD.

Table 1 – Publications

Title	Authors	Venue
SmartLTM: Larger-than-memory database storage for hybrid database systems	**Paulo Amora**, Elvis Teixeira, Daniel Praciano, Javam Machado	SBBD 2018
FLEXMVCC: Uma abordagem flexível para protocolos de controle de concorrência multi-versão	Eder Gomes, Filipe Lobo, **Paulo Amora**, Javam Machado	SBBD 2018
MetisIDX: From Adaptive to Predictive Data Indexing	Elvis Teixeira, **Paulo Amora**, Javam Machado	EDBT 2018
Adaptive Database Kernels	**Paulo Amora**, Elvis Teixeira, Javam Machado	SBBD 2017

We prototyped SmartLTM within PelotonDB(PAVLO *et al.*, 2017), a main-memory hybrid database system and performed our evaluation using benchmarks from the OLTP-Bench(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.

This dissertation is organized as follows: Chapter 2 describes with more detail the theoretical foundations of this work, such as main memory database management systems, workload types and data organization layouts; Chapter 3 delves in some related works regarding the several areas that this dissertation intersects; Chapter 4 goes into detail on how SmartLTM was designed, the rationale for auxiliary data structures and what the impact for database operations is; Chapter 5 presents our experimental evaluation and a brief discussion on the findings; Finally, chapter 6 summarizes the contributions presented with our experiment results and proposes where efforts to develop new work in the area should be directed.

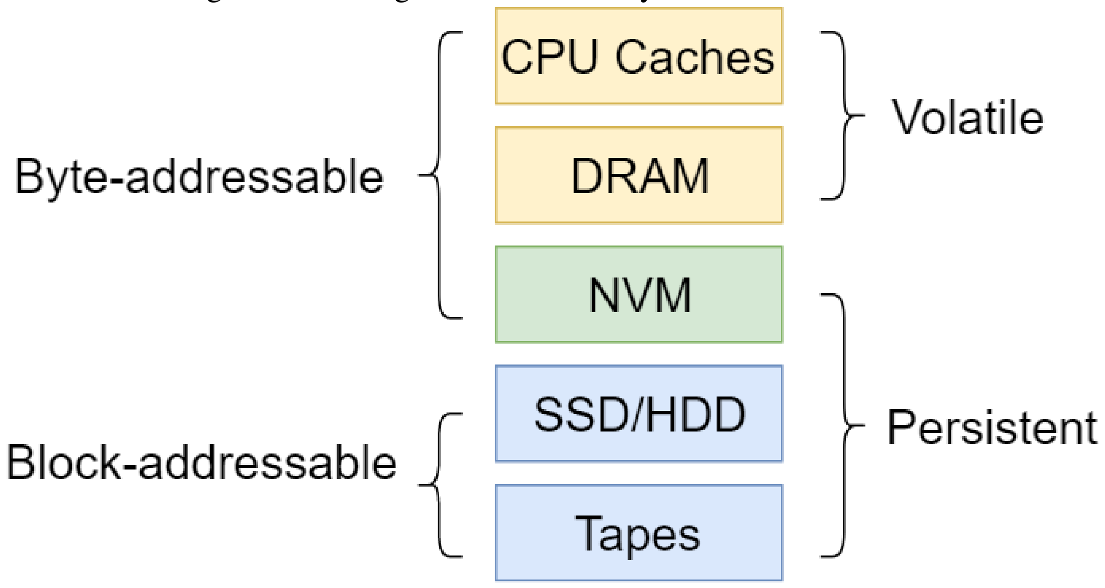
2 THEORETICAL FOUNDATIONS

This chapter presents the theoretical foundations in which this work is based on. It explains memory hierarchies, the special characteristics of in-memory database systems, the types of workloads, the data storage architectures, a brief discussion on the concept of data temperature and an introduction to the access filters used in the work. Section 2.1 describes the memory hierarchies in more detail, in order to showcase the differences between devices. Section 2.2 describes in detail the differences between traditional DBMS and MMDB, by presenting the algorithms used by MMDBs in different stages of the DBMS. Section 2.3 presents the workloads and their characteristics. Section 2.4 explains the different types of data organizations. Section 2.5 introduces the concept of data temperature and finally, section 2.6 gives a brief introduction on access filters and presents the ones that were used in this work.

2.1 Memory hierarchies

Traditionally, storage means in computers are differentiated following an hierarchy, as presented in figure 2.

Figure 2 – Storage devices hierarchy



The storage means have some special characteristics in which they are grouped by, regarding data volatility and data access.

Data volatility is defined as if data will still be present when there is no power feeding the hardware, it is split in two categories:

- **Volatile.** Volatile storage means do not keep data when power is down. More specifically, they do not preserve state.
- **Persistent.** Persistent storage means keep data when power is down. More specifically, they preserve state and are able to store data.

Data access is defined as how data will be accessed, it is split in two categories:

- **Block-addressable.** If data is accessed in a given position, a whole block containing this position must be read and loaded.
- **Byte-addressable.** If data is accessed in a given position, it is read from this position and the exact bytes for the data are retrieved.

We start the discussion from the storage medium with higher storage capacity and less speed, going forward to smaller and faster storage media.

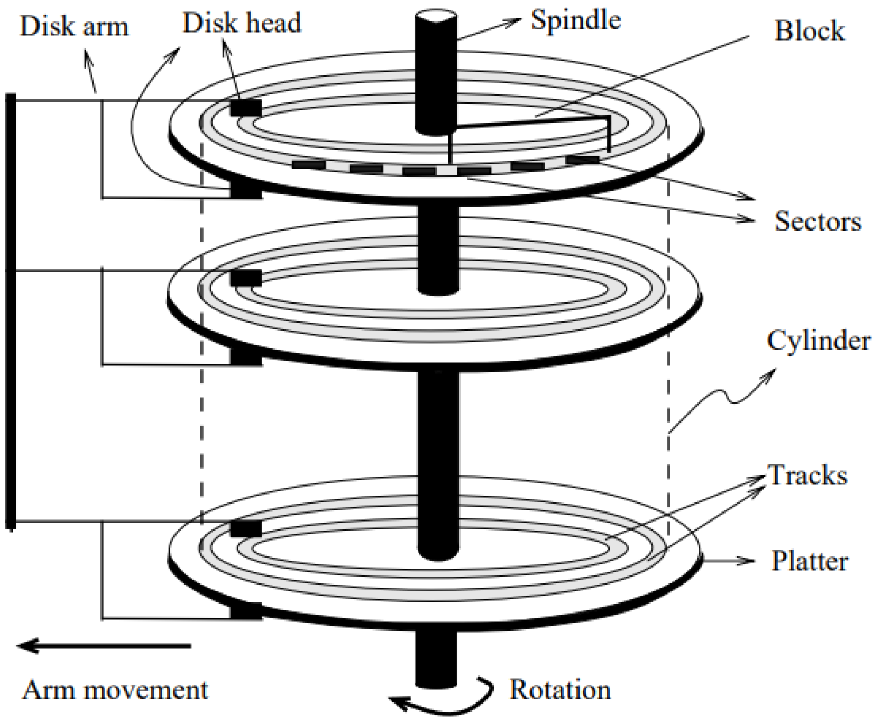
2.1.1 Tape

Tape storage is not used to store data on live systems, but rather for long-term storage of data, such as backups. Tape ribbons can store hundreds of terabytes on a drive, but access to data is sequential, serial, and block-addressed, meaning that for a point read, it is also necessary to retrieve more data. Any random access demands great effort because the tape must be rewound to the position, a process that can take tens of seconds. Thus, applications where these storage media are used should be adequate for the limitations of this medium. One example of an application that takes advantage of these constraints is a log application, where data is added over time, and if it is necessary to retrieve them, this is done from start to finish. Financially speaking, tapes are the cheapest storage medium, taking into account the price per storage ratio.

2.1.2 Hard Disk Drives

Hard disk drives are used as main storage media in most computer systems to date. These are composed of stacked magnetic disks and attached heads that read and write on these disks, moved by arms. Hard drives provide a good trade-off between storage capacity and speed of access. Nowadays, hard drives can store units of terabyte on a drive, and access them in tens of milliseconds. The optimal mode of access is still sequential, since the movement of the arms is unidirectional, but unlike the tapes, it is possible to random access in reasonable times, because as the disks spin at high speeds, the maximum wait for a given section with the head positioned is a rotation of the disc. Access to data on hard disk drives is also done by block, so it is important to consider which data will be stored in the same block. It is in these devices that most of the traditional DBMS data reside. Hard disk drives are economically viable for storage, and offer the best price-per-storage ratio and speed of access. Figure 3 (RAMAKRISHNAN; GEHRKE, 2003) presents the internal components of a hard disk.

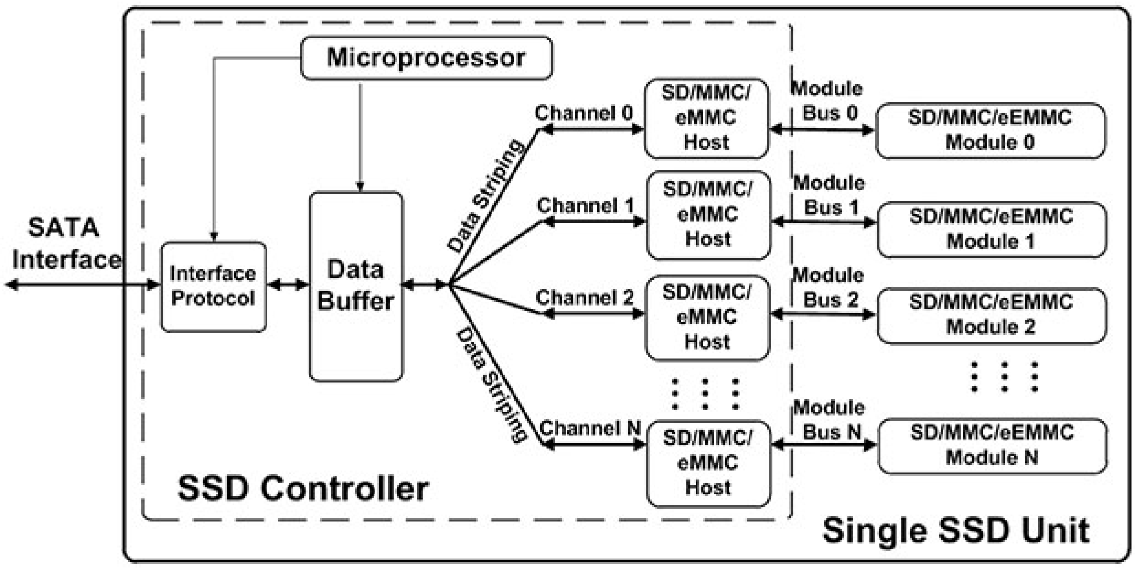
Figure 3 – HDD internal components



2.1.3 Solid-state Drives

With the evolution of FLASH memory technologies, solid-state drives (SSD) have become economically viable and the evolution to hard disk drives. While SSDs have still block-addressed storage, data access is not done mechanically, rather by electrical impulses. An SSD consists of several FLASH memory cells and a controller, which implements various algorithms, such as wear-leveling, to prevent the rapid degradation of cells, a limitation of the technology. As for the speed of access, it is possible to state that an SSD is at least 10x faster than a hard drive, depending on the devices being compared and how they are being compared. SSDs are not recommended for long-term data storage, nor are applications that are write-intensive due to their degradation. Commercially available SSDs have the storage capacity of hundreds of gigabytes. Thus, its high performance makes SSD the best choice for the last persistent layer of data. Compared to RAM, SSDs have around 100x the latency. SSDs are more expensive than hard drives, and incur a higher cost due to low durability in comparison. Figure 4 presents the internal components of a solid state drive. The I/O commands come from the SATA interface and are distributed by the controller in different channels, facilitating random and parallel access.

Figure 4 – SSD internal components



2.1.4 Non-volatile Memories (NVM)

In the near future, there will be non-volatile memories. These promise access speed close to main memory numbers, without its main disadvantage, the volatility of the data because they are durable storage media. In contrast to the storage media presented above, non-volatile memories have byte-addressable storage, meaning that data obtained from this storage medium is obtained by itself, making use of pointer indirection only. Several ways to implement this new paradigm exist, either through phase change memory (PCM) (LEE *et al.*, 2010), which uses a chemical element crystallization structure to characterize the state of a byte, magnetoresistive RAM (Rachel Courtland, 2014), which uses magnetic elements instead of electric current, and memristors (R. Stanley Williams, 2008), a new fundamental element of circuits, able to maintain the last state even without electric current. The target storage capacity of these memories is equivalent to the current SSDs. NVMs however, have a latency of 4x the latency in DRAM, which is fairly close, compared to SSDs. As they are still under development by their manufacturers, it is only possible to experiment with it by means of simulators but, they are already an object of study in academia.

2.1.5 Random Access Memory (RAM)

Also called main memory, random access memory is where most of the data that is in use on the computer resides. These memories have fast access speed and reasonable capacity, in the order of tens or hundreds of gigabytes. However, these memories are volatile, depending on electrical energy to maintain their state. Data access is byte-addressed, thus allowing the timely retrieval of the requested data, almost independently of the location of this data in memory. Applications are usually designed to use main memory when accessing current data. In traditional DBMSs, basically all the application is kept in memory, as well as auxiliary structures, for example indexes, most recently used data stored in buffers, among others. Those DBMSs were designed with not so much RAM available, therefore, the buffer manager needs specific algorithms to handle which data should be in the buffer. The volatility of these memories generates the need for a control of operations on resident data, the so-called log. Through this log, when there is a failure, the database can be restored to a consistent state, even if the latest data was not persisted. Despite the recent downward trend of recent years, RAM has still a high cost compared to other storage media, since a special infrastructure is required to use large quantities.

2.1.6 CPU Caches

Finally, the last layer of memory, the closest to the processor, CPU caches are special memories of very high performance and, because of this, of very small sizes. A typical CPU cache today has 3 layers. The first layer, actually close to the registers stores about 256KB, the second, which serves as support for the first, usually 1 or 2MBs and the third, which serves as a repository of the most used data of the memory, is about 8MB. The speed of these layers is also quite different. The processor is responsible for moving data between RAM and CPU caches, as well as movements between layers.

2.2 Main memory databases

Data from a traditional DBMS typically reside on some durable storage medium, also called secondary storage, whether it is hard disk drives or solid state drives. As described in the previous section, these storage media are in a magnitude below the main memory in relation to the access speed, and the traditional DBMSs have optimizations not to suffer as much the impact of recovering data of a medium of access so slow and with coarser granularity.

Garcia-Molina(GARCIA-MOLINA; SALEM, 1992) describes in their work the characteristics of a DBMS optimized in a scenario where RAM, also called main memory, is used as the primary data storage medium. Immediately we can notice some differences that this storage means brings to the DBMS:

- **Data access speed.** As main memory is faster than secondary storage with relation to access speed, it ceases to be the main bottleneck in query processing.
- **Main memory volatility.** Data stored in main memory are volatile, incurring in a design choice. Either accept the data volatility or a durability mechanism must be implemented, like the ones in traditional DBMS, however, they must restore the database to a consistent state from a blank slate, because in the event of a failure, previous data is lost.
- **Data access form.** In secondary storage means, to retrieve specific data incurs a high, fixed cost, which is the cost to retrieve a whole block from secondary storage. When data resides in main memory, the CPU is able to access it directly, due to main memory being byte-addressable, instead of block addressable.
- **Data organization.** Data organization is crucial in secondary storage, because the access is block based. Therefore, data access together must be together, because it makes use of sequential access, which is faster than random access in hard drives and reduces wasteful I/Os. In main memory, this difference is not as big, because data can be accessed through pointers. Data organization helps avoid unnecessary pointer chasing.

More recent works that address this theme were also considered for the following subsections (LARSON; LEVANDOSKI, 2016) (FAERBER *et al.*, 2017).

2.2.1 Data storage and access

Disk-resident DBMSs usually contain a dedicated component to improve response time when data needs to be accessed by the buffer manager. Since all data in a main memory database is already in the fastest medium, the buffer manager is unnecessary.

Even though the database fits entirely in the buffer manager, making all data memory resident, index structures are still designed to perform optimal disk access, for example, the B+-tree, thus not taking advantage of byte-addressed access that the main memory resident data provides. Main memory resident DBMSs in turn implement index structures optimized for this, such as the Bw-tree (LEVANDOSKI *et al.*, 2013b).

2.2.2 *Concurrency control*

Because the traditional DBMSs bottleneck is disk access, block-oriented pessimistic concurrency control protocols are acceptable as they do not impact the processing speed of queries so much. In main memory DBMSs, this premise can no longer be assumed to be true, so blocking concurrency control is indeed detrimental as it prevents parallelism in data access. The natural shift follows to more optimistic protocols, having as an alternative to avoid blocking the multi-version concurrency control protocols, evaluated and compared by Wu et al. (WU *et al.*, 2017).

2.2.3 *Logging and recovery*

In traditional DBMSs, while a transaction executes, it creates multiple log records, describing the operations performed so that when a failure occurs, the database can be restored to a consistent state. The most famous recovery protocol (ARIES) (MOHAN *et al.*, 1992) uses two steps, an Undo step, which analyzes the changes in memory that have not been committed and a redo step, which replays the changes that have been confirmed. Another optimization that is worth mentioning is called group commit, where several transactions feed a buffer and, if it fills or a certain time passes, it is written, committing several transactions with only one write operation in the storage device.

However, in main memory DBMS there is no data present after a database failure. This fact associated with the effort required to have the logging and recovery protocol motivates alternatives, such as Command Logging (MALVIYA *et al.*, 2014), which instead of registering the state of the data, registers the commands executed by the DBMS and performs them again for recovery. Write-behind logging (ARULRAJ *et al.*, 2016b) leverages in the memory hierarchy non-volatile memories.

It is also noted that although we are dealing with a database which data is completely in main memory, the log records by definition can not be in main memory, as these must be persistent, and the commit operation for transactions with write operations also depends on this record being written to the disc. However, writing records to disk is much less costly than reading and writing data to disk, as it is a unique, sequential operation.

2.2.4 Query processing

Query processing by itself does not have many changes when comparing disk-based and main memory DBMSs. Query processing components do. Access costs now weigh less in the optimizer decision. Optimizers now factor less access costs and more processing costs. Thus, algorithms that were optimized for disk access, such as sort-merge join, are not as effective. The compilation of query plans for machine code is also an optimization for in-memory DBMSs, since the traditional model of interpreting queries carries an inherent cost, and compilation helps avoid indirections.

Discussion

There are other differences between disk-based and main memory DBMSs, such as support for real-time analytics, availability, or distribution, but these differences will either be discussed further below, such as organizing the data to allow an optimization for different types of workload, or are out of the scope of this work, such as distribution and availability.

2.3 Workloads

Applications that make use of relational databases usually fit one of two categories: Online Transaction Processing (OLTP), that groups applications with a profile of many write queries and point queries and Online Analytical Processing, with applications that execute table scans and aggregations over these scans, iterating through big volumes of data. As they are very distinct access patterns, DBMS usually focus on one of the categories.
OLTP. OLTP workloads are characterized by their large volume of write operations, as well as point queries, and usually retrieve all the attributes of a record. Most relational databases and business applications have this profile. OLTP workloads are composed of insertions and updates to records, as well as point queries, which retrieve entire records, with their results based on key equality.
OLAP. OLAP workloads are characterized by their read-only operations and scan all tuples to retrieve values of an attribute. These queries usually execute over the whole table. Special purpose databases have this profile. OLAP workloads are composed of table scans and aggregations, which retrieve a small subset of attributes from all records, with their results based mostly on range intervals.

HTAP. As new one-size-fits-all DBMS are developed, a new definition of workload, called HTAP arose. HTAP workloads, also known as OLXP, have characteristics from both OLTP and OLAP workloads. Databases that are optimized to these kind of workloads usually have a mechanism that applies a change within the data organization, or maintains different organizations for each type of query. This process usually occurs in an adaptive way, with the layout being reorganized according to the workload characteristics, such as which attributes are being currently accessed, if they are accessed together in a projection or as a predicate, among others. These characteristics are extracted from the incoming queries.

2.4 Data organization

Each workload mentioned above accesses data in a different manner, and data organizations can be optimized to better answer a specific kind of workload. Due to their characteristics, OLTP and OLAP queries can be better answered by different data layouts, and the possible optimizations that these layouts allow for. Row storage is better for OLTP queries, since all data pertaining to a tuple is stored together, while column storage excels in OLAP workloads, because their queries care more about retrieving the same attribute from several records. Abadi et al.(ABADI *et al.*, 2008) conducted a study where a native column-store DBMS is compared against a highly-optimized row-based storage DBMS for OLAP queries, and they found that even with every optimization in place, such as vertical partitioning or indexes over a complete column, the native column-store would still overperform a row-based DBMS, justifying the separation between layouts and types.

2.4.1 NSM

The N-ary Storage Model (NSM), also known as row store, is the one where data for a record in the table is stored contiguously, speeding up insert, update and delete queries and queries over a specific tuple, while accessing many of its attributes. This storage model is not favorable for OLAP workloads because all data from a record is stored together and fetched together, there are many wasteful I/O operations of data that will be fetched and not modified.

2.4.2 DSM

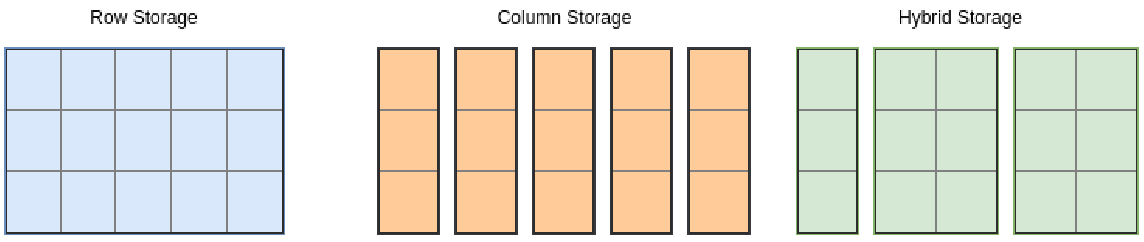
The Decomposition Storage Model, also known as column store, is the one where data for a given attribute of a table is stored contiguously, speeding up queries over single attributes and queries over a range of tuples, accessing a small portion of their attributes. This storage model isn't favorable for OLTP workloads because all data from a record is stored separately, forcing an overhead called tuple stitching when restoring a single tuple, and tuple splitting when a new tuple is inserted.

2.4.3 FSM

For HTAP workloads, a Flexible Storage Model is proposed by Arulraj (ARULRAJ *et al.*, 2016a), although previous works (ALAGIANNIS *et al.*, 2014) make use of a similar concept without naming it. The idea behind this FSM is a model in which data could be represented as NSM, DSM and partly as DSM, as group of columns. A table can have different data layouts, depending on which records are being accessed through the workload and the decision to maintain this hybrid layout may come from a configuration knob set by the DBA, be inferred online or offline. One example of this flexible storage model is the Tile architecture(ARULRAJ *et al.*, 2016a).

Figure 5 presents each of the data organizations.

Figure 5 – Data organizations



2.5 Data Temperature

Many works (FUNKE *et al.*, 2012) (ARULRAJ *et al.*, 2016a) (LANG *et al.*, 2016) draw a clear definition of data temperature, at least the definition of hot and cold data. Hot data are the records that are being currently accessed by the workload, usually for update queries and are mostly recent records. As those records get old, their access ratio decreases and they become cold. Cold data are the records not being currently accessed, and eventually are accessed for scan and aggregation queries, which can be optimized to avoid retrieving record by record, by keeping some pre-computed values. The temperature of records on a table is inherently dependent of the workload, depending on which records are more read or updated.

2.6 Access Filters

Access filters are important when probing if a given value is present within a data collection. For our work, we must avoid accessing the secondary storage unless strictly necessary. In this section we will discuss the access filters used in this work.

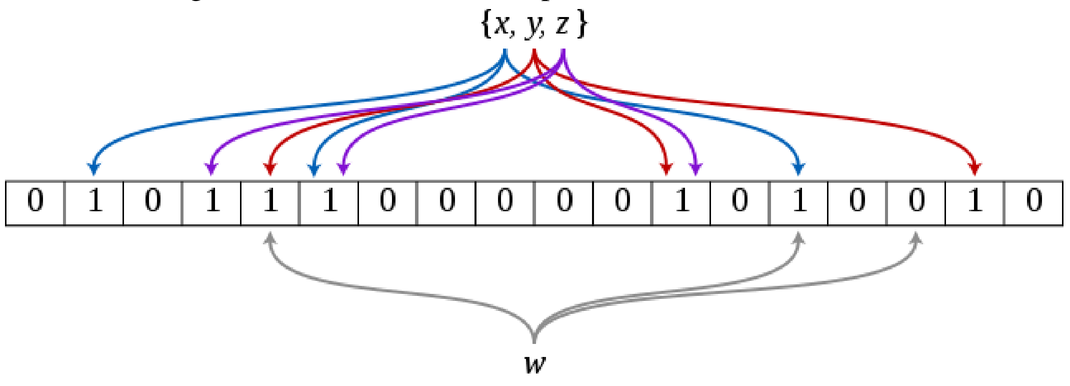
2.6.1 Bloom Filters

Bloom Filter (BLOOM, 1970) is an access filter designed for scenarios where false positives are acceptable, but false negatives are not. A Bloom filter is composed of a bit array initially set to all 0 and a set of hash functions. When an element is added to the filter, it goes through all the hash functions, and the array positions outputted by the functions are flipped to 1. When another element is probed to the filter, the functions output can assert that, if any positions are set to 0, this means that the probed element was never added to the filter. If all the positions are set to 1, this means that the probed element may have been added to the filter, but it is not certain that it is actually there. Bloom filters do not support deletions by nature, because removing an element from the filter (resetting the affected positions to 0) can affect other entries.

Bloom filters are used nowadays on a myriad of applications, because they are a cheap, easy to understand and implement and powerful access filter.

Figure 6 presents the bloom filter operation, with 3 elements, x, y, z added and w looked up.

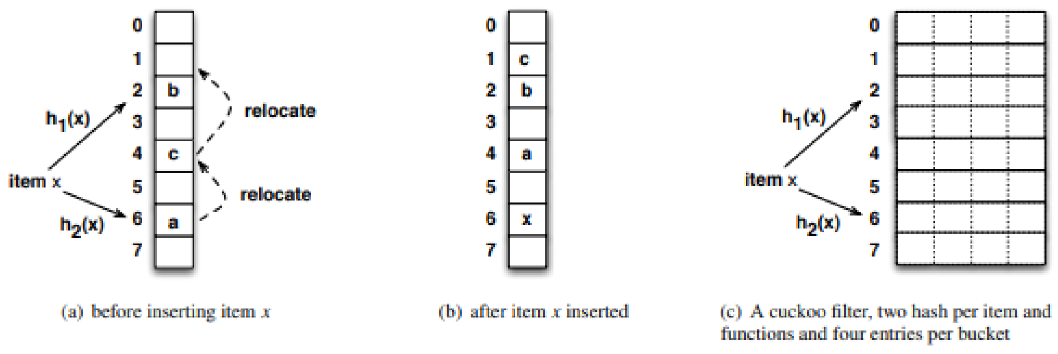
Figure 6 – Bloom filter example



2.6.2 Cuckoo Filters

Cuckoo Filters (FAN *et al.*, 2014) are designed over cuckoo hash tables, and provide a more efficient alternative to Bloom Filters, supporting delete operations and being more space efficient than Bloom filters. A cuckoo filter stores data fingerprints, instead of the complete value. Those fingerprints are of fixed size and are used to densely fill the filter, allocating as much information as possible before having to increase size. The fingerprint size is based on a given false positive acceptance and fingerprints are calculated by applying partial-key cuckoo hashing. In general, for the same filter size, cuckoo filters allow for more keys to be stored, allow 2 misses per lookup, as opposed to the number of hash functions in a regular Bloom Filter, and support deletion. Figure 7 presents the cuckoo hashing operation that bases the cuckoo filter.

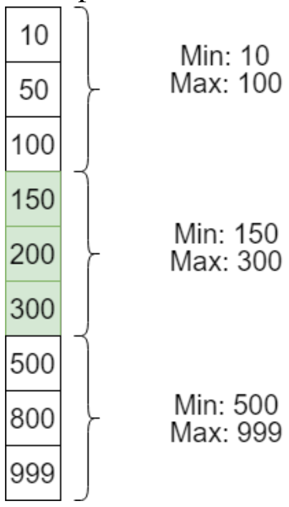
Figure 7 – Cuckoo Filter


0
1
2
3
4
5
6
7
b
c
a
$h_1(x)$
$h_2(x)$
item x
relocate
relocate
(a) before inserting item *x*
c
b
a
x
(b) after item *x* inserted
(c) A cuckoo filter, two hash per item and functions and four entries per bucket

2.6.3 Small Materialized Aggregates(MOERKOTTE, 1998)

Small Materialized Aggregates (SMA), also known as Zone Maps, are pre-computed aggregates over a subset of tuples present in a table. Materialized aggregates ease the work of statistical queries, by avoiding recalculation of some values. The idea of having these aggregates over a subset of the data is to be able to determine if a given value is present within the subset of tuples. For example, if the minimum value among the subset is 10 and the maximum is 100, if the search predicate is > 200, the executor can skip this tuple subset and save time, because the value will certainly not be there. Figure 8 exemplifies this process.

Figure 8 – SMA example

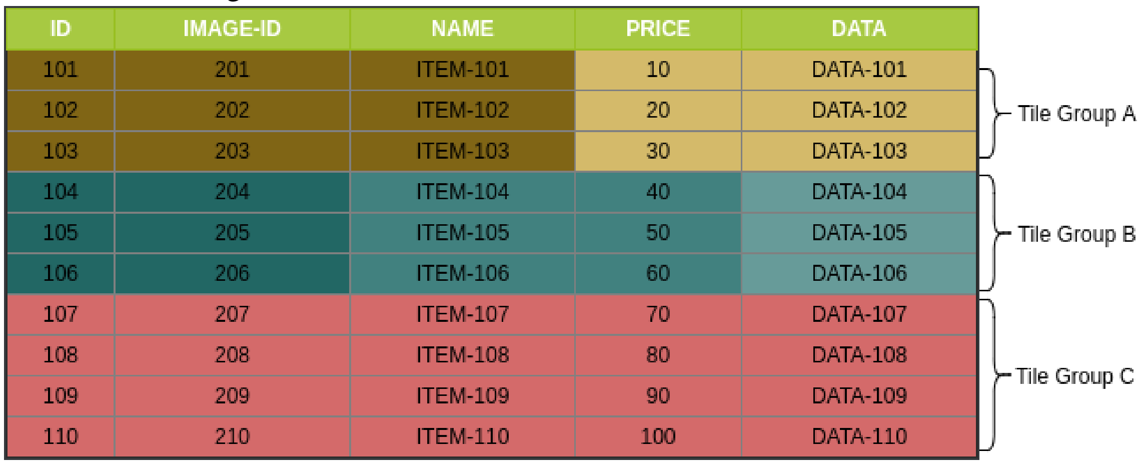


2.7 Peloton(PAVLO *et al.*, 2017)

Peloton is an HTAP multi-version main-memory DBMS developed at Carnegie Mellon University. While more detail about some components will be provided in the next chapter, a few concepts are fundamental to our work, therefore, they must be explained here.

The storage layer is based on the tile architecture (ARULRAJ *et al.*, 2016a), and is composed of the following components, presented in figure 9

Figure 9 – Tile architecture



ID	IMAGE-ID	NAME	PRICE	DATA
101	201	ITEM-101	10	DATA-101
102	202	ITEM-102	20	DATA-102
103	203	ITEM-103	30	DATA-103
104	204	ITEM-104	40	DATA-104
105	205	ITEM-105	50	DATA-105
106	206	ITEM-106	60	DATA-106
107	207	ITEM-107	70	DATA-107
108	208	ITEM-108	80	DATA-108
109	209	ITEM-109	90	DATA-109
110	210	ITEM-110	100	DATA-110

Tile Group A
Tile Group B
Tile Group C

The colored rectangles are named tiles, and are equivalent to a vertical and horizontal partitioning of a table, containing data regarding a subset of the attributes and tuples present in the table. The tiles are grouped in tile groups, representing horizontal partitions of the table. While tiles can have different attributes, the schema of a tile group matches the schema of the table. The table is composed by metadata and an array of tile groups. In the figure the tile group size is 3, and it can be noticed that the different tile groups have different organizations, based on the workload that accessed data inside those tile groups. For example, the first tile group in the figure was accessed by queries that projected the three first attributes while selecting the other two. Queries that grouped different attributes motivated the layout generator to split data accordingly. The last one is equal to a row-based storage given that it is recent data and more probable to be updated than older data.

Tile groups have fixed sizes, and once the active tile group is filled, a new one is added to the table, with a default row-based layout.

Part of the table metadata is a column map for each tile group. This column map links the table schema to the internal tile group organization, in a zero-based form. Taking Tile Group B in the figure for an example, the column map would be presented in the following form: $[0,[0,0];1,[0,1];2,[1,0];3,[1,1];4,[2,0]]$ Meaning that the column 0 of the schema is mapped to tile 0, tile offset 0 of the tile group, column 1 is mapped to tile 0, tile offset 1, column 2 is mapped to tile 1, tile offset 0, and so forth.

2.8 Discussion

This chapter discusses the theoretical foundations of our work, going into concepts that motivate and ground the hypothesis of the work and presents the concepts and the platform in which the hypothesis was evaluated and the algorithms were developed. Next chapter delves deeper on the related works and further discusses their approaches.

3 RELATED WORKS

This section presents related works, grouped by area of interest. Section 3.1 describes strategies to handle the growth of data in memory, and details the main approaches regarding data eviction. Section 3.2 gives an overview of different hybrid storage databases, that while do not implement data eviction, were evaluated for our strategy. Section 3.3 presents different approaches for hot and cold data characterization.

3.1 Strategies to save space

This section describes strategies used on multi-versioned DBMSs with the purpose of saving space and optimizing performance.

3.1.1 Garbage Collection

Multi-versioned Concurrency Control (MVCC) creates new versions every time a tuple is updated or deleted, therefore, there is a need to identify which versions are obsolete and can be safely discarded thus, reclaiming memory. A garbage collection mechanism becomes a necessity. Wu et al. (WU *et al.*, 2017) does an extensive work evaluating garbage collection approaches with MVCC.

Garbage Collection (GC) is performed in three steps: (1) The expired versions are detected, (2) unlink those versions from the indexes and associated chains, (3) reclaim the memory used by these versions. An expired version can be an invalid version or a version that has the end timestamp set to before the current timestamp of all transactions.

This version tracking in main-memory DBMSs can be performed in a coarser-grained epoch-based memory management. This helps avoiding extensive overhead that would occur if the tracking is done by version. The epoch groups several versions and eases the tracking task.

GC can be implemented in two ways: Tuple oriented or Transaction oriented.

The tuple oriented method scans each tuple individually, and can be also performed in two ways: The first one has background threads periodically scanning the entire database looking for expired tuple versions. Its drawback is that it does not scale well with the database size. The second one is cooperative, and when a transaction needs to traverse the version chain, it also uses the traversal to check for expired versions of the tuples. This design can leave expired tuples for a long time in the database if those version chains are never traversed.

The transaction oriented method tracks transactions instead of tuples. The transaction is considered expired when the versions created by it are not visible anymore by any other transactions. When an epoch ends, all the versions created by transactions contained in the epoch can be safely removed.

GC is an essential part on how to reclaim space and allow new data to come in the in-memory DBMS. It is definitely the first strategy to be implemented, but having only GC can incur in a problem for Data Aging. The tuple space reclaimed can be reused as-is by new data. If you reclaim and provide space from a cold block, you might mix hot and cold data, making the characterization harder.

3.1.2 Data Compaction

Data compaction, also known as defragmentation, is a process where scattered data is moved near each other to be stored contiguously. While this might not be an actual concern in random access memory, compaction can help save space avoiding mixing hot and cold data. Take the scenario where GC would free the memory of expired versions, but not provide these spaces as free memory. To the DBMS, the GC would not be doing much, since it cannot access the free memory, although since it would never reuse memory, mixing hot and cold data would not be an issue. By using compaction algorithms, cold data could be reorganized and the free memory would be available again.

This process is very costly to perform in block-addressed devices, especially HDDs due to their design. However, copying memory over to target addresses is an easy task with random access memories.

3.1.3 Data Compression

The next step is having the cold data occupy less space. Since it is not frequently accessed, we can allow a trade-off between access performance and occupied space. Therefore, cold data can be compressed and summarized. Another point is that cold data is rarely accessed by point-queries. There are structures which allow for summarization, helping the DBMS skip irrelevant compressed data blocks and answer queries upon this data without actually having to decompress it.

To summarize compressed data, a structure called Small Materialized Aggregates (SMA) (MOERKOTTE, 1998), as described in chapter 2, is very effective in avoiding access to a compressed block, especially if data is sorted. It works by precomputing several aggregates, like max, min, sum, avg, etc., and when queries, usually OLAP queries try to retrieve these types of aggregation, they can simply query the precomputed aggregates. These SMAs are best used on immutable data, to avoid the overhead of recomputing them. However, when we say that cold data is immutable, this does not apply to deletes. Deletes on old data are rare, but can happen, and the SMAs need to be updated accordingly.

HyPer (FUNKE *et al.*, 2012) initially had this cold data compressed and stored in huge pages, because it was beneficial to their *fork()*-based architecture. Later, they evolved to a structure called Data Blocks (LANG *et al.*, 2016) which contains cold, immutable data in a flat structure and keeps data organized in a columnar layout. Data blocks also make use of an auxiliary mechanism called Positional SMAs to avoid looking for data in the whole data block, even when the outside aggregations are not sufficient to filter access. With the Data Blocks, HyPer maintains the hot data uncompressed and the cold compressed in different ways, considering which is best for each attribute type. To retrieve this data, the PSMA contains a lookup table in which the data position is checked.

3.1.4 Data Eviction

At last, when space runs out and there is nothing left to do with respect to data transformation, it is time to free some space by moving data to another storage. This storage is bigger, slower and usually does not have the same characteristics as RAM. The data that must be moved out should be selected from the least accessed cold data, given that even reads would be delayed by having to access another storage device. Also, since the storage device might not have the same features as main memory, how to store data in this device with regard to data retrieval becomes a concern when moving data out.

Anti-caching (DEBRABANT *et al.*, 2013) is a technique created for OLTP larger than memory databases, which consists in moving cold data to disk, in order to free space in memory. Two structures are used to enable retrieval of tuples in disk, the Evict Table, which contains the block id and tuple id for a given tuple and the Block Table, a disk-resident hash table that stores the evicted tuples in blocks, in a format that resembles the tuple format in memory.

When a tuple is evicted from main memory, all indexes are updated with a special flag that enables the DBMS to recognize that a tuple was evicted, and to look for it in the Evict Table instead. In the table, a special pointer is placed instead of the tuple, therefore, saving space.

This special pointer is called tombstone.

In terms of transaction processing, once a transaction access an evicted tuple, it is aborted and restarted after the DBMS fetches the tuple back from disk and merges it to the table. Another approach is to hold the transaction while data is merged back to the table, which is called synchronous retrieval.

Tuple merging can be done by block (merge the whole block retrieved on disk) or by tuple (merge only the needed tuples). Another approach is when tuples are required only for read-only purposes, they are not merged at all, instead, they are kept in a separate memory space, used and then discarded. This requires a closer monitoring of how these tuples are accessed to merge back those most accessed.

Traditional DBMS systems check the cache in main memory for the required register, and, if they do not find it, they go for the disk to look for it. Anti-caching systems usually have all the registers in main memory, but, in case the register is not present, then anti-caching access the disk to retrieve the register.

Project Siberia (ELDAWY *et al.*, 2014) is another approach to the OLTP larger than memory databases issue. It was proposed to work with Hekaton, which is the SQL Server in-memory database. It differs from Anti-caching in several aspects.

To determine if a tuple was evicted or not, Project Siberia uses a probabilistic data structure called Bloom Filter, which can accurately determine if an element is not present. However, it cannot say for sure if an element is present.

Siberia use synchronous retrieval, halting the transaction as an absent tuple is accessed.

It also uses the *always merge* strategy, which merges the tuples as soon as they are retrieved.

3.1.5 Discussion

Table 2 presents the directly related works in comparison to SmartLTM.

Table 2 – Related Work Comparison.

Authors.	DeBrabant et al, 2013	Eldawy et al, 2014	*Amora et al, 2018*
Research Problem.	Row-based larger-than-memory storage (on-line)	Row-based larger-than-memory storage (of-fline)	*Larger-than-memory storage for hybrid systems*
Access to evicted data.	Tombstones (pre-pass overhead).	Bloom filter-s/Adaptive range filters.	*Cuckoo filter-s/SMA*
Retrieval of evicted data.	Asynchronous, block based.	Synchronous, tuple based.	*Synchronous, tile-based*
Evicted data stor-age.	Block of tu-ples.	Tuple by tuple.	*Store tile data separately, according to DBMS*

3.2 Hybrid and Adaptive storages

3.2.1 PAX - Partition Attributes Across

PAX (AILAMAKI *et al.*, 2002) is a cache oriented approach, with its design inspired by NSM and DSM. On NSM models, the tuple data is stored contiguously into a page, and whenever the page is loaded, many unnecessary data is loaded along. DSM solves this problem, but tuple reconstitution is expensive.

PAX uses an approach in which the storage page is subdivided into minipages. The pages still keep data as the NSM model, all together, but inside, the minipages store data separately, as a DSM model. When data is to be loaded, the page header provides the necessary offsets to allow wasteless loading of data and, by consequence, less page swaps due to useless data being loaded. Since PAX only affects layout inside the pages, no storage penalty is incurred and no I/O operation is affected.

Minipages can be for Fixed or Variable size attributes. If the minipage contains fixed-size attributes, presence bits are used to determine whether data is present. If the minipage contains variable-sized attributes, the offsets are stored to determine where each field ends.

(AILAMAKI *et al.*, 2002) also describes the implementation of PAX, a case study comparing NSM, DSM and PAX performances for various workloads.

PAX uses a hybrid approach, but only in concern to cache. There is neither an intelligence on layout refactoring or on the query processing, nor for specific workloads, only helping optimize cache access.

3.2.2 Data Morphing

Data Morphing (HANKINS; PATEL, 2003) is a flexible data storage technique, that uses a cache-efficient attribute layout, called partition, which is determined through an analysis of query workload. The partition is used as a template for storing data in a cache-efficient way.

The paper contributions are: The flexible page architecture, where attributes of the same tuple can be stored in non-contiguous groups, increasing the spatial locality of memory accesses; two algorithms for calculating the attribute groups that are stored on each page; a naive algorithm that tries all possible layouts, and a hill-climb algorithm that performs a greedy search of the layout space, while not ensuring the optimal organization for data, it finishes in a more feasible time.

Data Morphing consists of two phases, calculating a cache-efficient storage template and reorganizing data into this storage template. To store in a cache-efficient way, the flexible page is described as a generalization of the pages in PAX(AILAMAKI *et al.*, 2002), since PAX stores each attribute individually. Data morphing stores groups of attributes, to avoid cache misses when attributes are required together. Data morphing allow for dynamic reorganization of pages.

The partition calculation takes as input a relation R and a set of queries Q. A query q is an ordered sequence of pairs *(x,y)* where x represents the attributes of R accessed and y the frequency of access. The naive algorithm calculates the optimal partition based on the cost of each possible partition, and the optimal partitions are the ones that result in the fewest number of overall cache misses for given workload. This algorithm is very expensive and impractical for a large number of attributes, given that its time complexity is $O(e^{n\ln(n)} + mn2^n)$ and space complexity is $\Theta(2^n)$. The hill-climb starts as the naive algorithm, but then, the algorithm takes an iterative approach to select a partition, and considers the effect of merging two partitions. The cheapest cost partition is picked for the next iteration. Thus, in each interaction, the number of partitions reduces by one.

The paper describes an experimental setup showing that Data Morphing is 45% faster than the N-ary storage model and 25% faster than PAX.

Data Morphing, although introduces concepts used by the newer approaches (the attribute grouping is similar to groups of columns of H_2O(ALAGIANNIS *et al.*, 2014)), is more of a refining of PAX, by allowing cache to keep more related information than to optimize cache for readings.

3.2.3 HYRISE

HYRISE (GRUND *et al.*, 2010) is a main memory hybrid database system that automatically partitions tables into vertical partitions of varying widths depending on how the columns are accessed. The paper states that it is preferable to use narrow partitions for columns accessed as part of analytical queries. HYRISE stores columns accessed in OLTP-style in wider partitions, in a work closely related to Data Morphing (HANKINS; PATEL, 2003). The contributions are: A detailed cache performance model for layout-dependent costs. An automated database design tool that, given a schema, a query workload and the performance model, recommends an optimal hybrid partitioning. The paper also concludes with an analysis between the architecture proposed and previous hybrid storage schemes.

HYRISE is composed by a storage manager, that is responsible for creating and maintaining the hybrid containers, a query processor that receives user queries and creates a physical query plan, and a layout manager that analyzes a given query workload and suggests the best possible layout.

Each partition is called a container within HYRISE. Containers are physically stored as a list of large contiguous blocks of memory.

3.2.4 HyPer

HyPer (KEMPER; NEUMANN, 2011) is a main memory database that allows for both OLTP and OLAP workloads by shifting the organization of data between row and column stores. It works as a row-based storage, for OLTP workloads. Whenever a OLAP workload comes, a session is opened and a snapshot of data in memory is created for the process responsible for the OLAP workload. This is done by a *fork()* system call on the OLTP process.

The data in the forked process is viewed as a column store. Tables are stored as disjoint sets called partitions. These partitions are composed of vectors, which are the attributes, and a group of vectors creates a chunk, which is a horizontal subset of the partition. This hybrid concept allows for OLTP processing using chunks and OLAP using vectors. The table is divided into partitions to ease the concurrent access to data in a same table.

Modern operating systems do not physically copy all memory in a *fork()* call, they instead use a lazy copy-on-write strategy. New data is merged to the OLAP workload by subsequent calls of *fork()*.

3.2.5 H_2O (ALAGIANNIS et al., 2014)

H_2O is an adaptive storage that designs itself, guided by the query workload. It generates layouts to better fit data given the query workload.

H_2O is designed with the following architecture: A query processor that examines the incoming queries to decide how data should be accessed. Once this is decided, the operator generator creates code for access operators. Data can be stored as a row-major layout, column major layout or a hybrid between the two. The layouts are not definitive, as queries change, new layouts are generated and adapted, given that the generation cost can be amortized by future queries using this layout. To allow that, H_2O uses a dynamic window to monitor the access patterns of incoming queries. This window provides hints on how the new layouts must be generated.

To access these layouts, H_2O provides multiple execution strategies, oriented in column and row-major layouts. For hybrid layouts, the execution is decided based on the query. The query results are materialized in the main memory in a row-major layout. Those strategies are generated on-the-fly, because the layouts are unknown until then. To generate these layout-aware operators, code templates are used, which fit different query plans. These templates take as input the needed data layouts and output the source code, which is compiled externally and dynamically linked in the query execution plan. To minimize this overhead, H_2O keeps an operator cache.

The combination of data layout and execution strategy is how H_2O calculates its query cost, using a model that takes those parameters into account, similar to HYRISE (GRUND *et al.*, 2010).

H_2O, despite being an experimental prototype, shows a lot of promise in the experimental analysis. The approach of composing groups of columns into a hybrid layout beginning from a column-store is smart, given that it is easier to concatenate columns than to split data. However, it is wasteful in terms of memory use and operator generation, given that the layouts are materialized and kept in main memory. This creates an impact on the query workload, because it is geared to unknown OLAP applications. It is difficult to maintain consistency within all data replicas with an insert/update operation. Therefore, it is a great solution for hybrid OLAP workloads, but not hybrid workloads.

3.2.6 Tile Architecture

The tile architecture (ARULRAJ *et al.*, 2016a) aims to provide cache-optimized storage for data, taking into account the access patterns and designing the storage as time passes.

This tile architecture is transparent to the query execution engine, which sees only a row-based representation of data. To give support to this abstraction, the architecture is divided between physical and logical tiles. This architecture is thought taking into account the behavior of this kind of workload, which can be described with applications such as eBay, in where data starts being subject to a heavy OLTP-kind of workload, but after a given time, this data is not modified anymore, being used only for analytical and statistical queries, an OLAP-kind of workload.

The schemas are stored in physical tiles, which partition physically the schema, both horizontally and vertically. Within a table, can reside several physical tiles, each one with its own layout and data. Data starts aligned in a row-based storage, to facilitate insert and update operations. Then, as time passes, this data becomes colder and may be reconstructed in another tile.

The separate tiles would require different execution strategies, given the different layouts. This problem is solved by logical tiles, which are tiles that contain only metadata and offsets. The query engine usually operates on logical tiles, and the output of these operations are also logical tiles. For simplicity, logical tiles do not map to other logical tiles, only physical tiles, which are materialized when needed. A logical tile algebra is defined in the paper, showing how the operations workflow is composed and executed.

The layout reorganization is handled by a background process, composed by a lightweight monitor that tracks the attributes accessed by each query. With this information, the partitioning algorithm can decide how to separate the data.

The tile architecture is part of the PelotonDB (PAVLO *et al.*, 2017) project.

3.2.7 Hekaton

Hekaton (DIACONU *et al.*, 2013) is Microsoft SQL Server's take on a main-memory database. It uses several infrastructure from SQL Server, with a complete reconstruction of the transaction processing and storage related layers. It makes use of lock and latch-free structures, to allow concurrent access to be fast. It is also multi-versioned for that same reason. Indexes are optimized for main memory, as they use a variant of the B+-tree called Bw-Tree (LEVANDOSKI *et al.*, 2013b). The in-memory storage is similar to the SQL Server, in the way that data is stored in tuples.

The storage design differs from other approaches on how data is stored. They argue that there is no benefit in partitioning the data if the workload cannot also be partitioned similarly. Therefore, they avoid this strategy, making the whole data available for any thread.

There are more components of the Hekaton environment that will be explained in further sections.

3.3 Hot and cold data characterization

This section describes strategies to track data that is not being used anymore and has not been used in a while. This data must still be retrievable, but it does not need to be as available as more relevant data. In accordance to recent literature, we will call relevant data hot and obsolete, stale data, cold.

3.3.1 Project Siberia

Hekaton's tracking of hot and cold data is done offline, meaning that data is not tracked by Hekaton itself, but by a separate process.

It keeps an access log (separate from the transaction log) where accesses are recorded. A record in this log contains a record id and the correspondent time slice, which is not counted by an actual clock, but every access count as a tick.

To estimate record accesses, it uses exponential smoothing over the time slices. To reduce logging overhead, not all accesses are recorded, instead, a sample is taken by flipping a biased coin. If heads, the access is recorded, if tails, not.

Siberia (LEVANDOSKI *et al.*, 2013a) is defined as a cold storage for Hekaton. It is based on cheaper, slower memory than RAM and that is where data goes when it gets cold. By making use of the access log and the auxiliary structures, Hekaton is able to move data to and retrieve data from Siberia.

From the access log, classification is done by also taking a constant *K* that represents the number of hot records. The hot record set is the K-est record with the highest access frequency, all the others are considered cold.

Siberia describes two strategies for reading the access log, from beginning to end (forward) and from end to a point where there would be no candidates for the hot set. Parallel algorithms for these two strategies are also presented.

3.3.2 HyPer

HyPer (FUNKE *et al.*, 2012) defines data in four temperatures:

- Hot, which is data currently being accessed, also called Working Set.
- Cooling, which is a section containing cold items and some hot/re-heated items.
- Cold, which has only cold items, if data from a cold block is accessed by a transaction, the whole block is reclassified as cooling, due to the item re-heating.
- Frozen, which are cold, immutable items. Frozen items are created from cold items, compressed and stored differently as well. If something tries to update a frozen item, the frozen entry is invalidated and a new entry is added to the working set.

Do note that access is defined as an OLTP access. OLAP queries do not increase the temperature given that they are read-only.

HyPer tracks these temperatures using an Access Observer, which is implemented asynchronously from the database and uses special instructions from the hardware to perform the tracking. Through the CPU MMU flags *young* and *dirty*, which are placed when a record is accessed and modified respectively, the Access Observer obtains the information about each chunk of data and manipulates these flags through a *mlock* system call, to avoid these entries to be paged out. Since OLAP queries are heavy readers, the *young* flag is only taken into account if an OLAP query did not touch the page.

3.3.3 Anti-Caching

The characterization of a cold tuple in H-Store using Anti-Caching (DEBRABANT *et al.*, 2013) is made based on the LRU policy. To do this, it keeps track of every tuple use in a list of the most recent, and, after a threshold, a special transaction locks the tables to allow a consistent selection of the cold data. This cold data is then copied into pages and these pages flushed into disk. Once they are safely stored, their data in the main-memory storage is substituted by a special 64-bit value called tombstone. The details of the eviction process, alongside the structures will be described in a later section.

This LRU chain is a very naïve solution, because it incurs several overheads to maintain this chain. From memory space to operation complexity, this approach brings a big overhead to the system.

3.3.4 Tile Architecture

The layout organization is performed by a background process that tracks the access patterns of the workload, mainly attributes touched on *SELECT* and *WHERE* clauses of queries. This reorganization process is incremental and is not done on groups that are being used on current OLTP transactions.

This allows for avoiding latency on running transactions while optimizing the storage for OLAP queries. To avoid the issue with oscillating workloads, the DBMS prioritizes older query samples.

3.4 Discussion

The related works present different approaches regarding several points of interest of our work. From different data storages to other strategies that evict data from the main memory, these works inspired in the research question and our proposed solution. Related to the problem discussed, there were no strategies found designed to work with hybrid storages, with respect to the larger-than-memory scenario. SmartLTM leverages the studies on hybrid storages and larger than memory storages to propose a novel way to store data when dealing with HTAP DBMSs. The next chapter explores the design of SmartLTM.

4 SMARTLTM

In this chapter we present SmartLTM, our proposed architecture that leverages the work and decisions taken by the DBMS regarding the optimal data storage layout based on the workload.

Below we describe the SmartLTM architecture and detail its components. Figure 10 presents the interaction between components when a query is posed against the DBMS. We focus only from the execution engine onwards because this architecture is transparent for the other components on the DBMS.

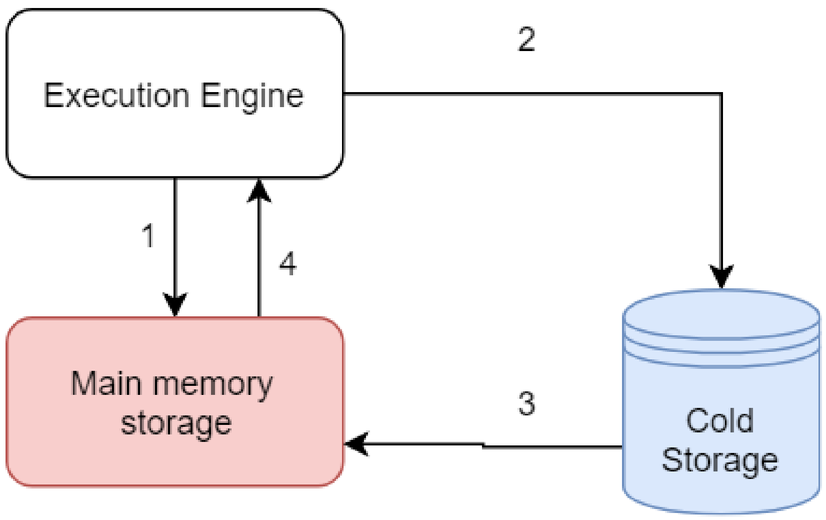


Figure 10 – SmartLTM architecture

First, after the query is processed by the upper layers (parser, planner, optimizer), it begins execution. The execution engine then tries to answer the query only with memory-resident data. If it succeeds, then we move to step 4. If the query cannot be or is not completely answered by in-memory data, it probes the cold storage for the data (step 2). The cold storage then loads the portion of data required to answer the query in the main memory, which then answers completely the query. This data is discarded upon the transaction termination.

4.1 SmartLTM composition

SmartLTM is composed of the cold storage, which keeps data not currently being accessed, the access filters, namely cuckoo filters and SMAs. SmartLTM is implemented in PelotonDB, which follows the tile architecture, therefore, some naming conventions are used, such as tiles, tile groups and column maps, chapter 2 has more detail about those implementations.

4.2 Cold data characterization

While not the focus of the work, cold data can be characterized as data that is not being frequently updated or specifically read. Therefore, range and aggregation queries would not impact as much in data temperature, but point queries and updates will. A form of choosing our candidates could be by counting those accesses at the tile group level, while being careful to not recycle any tuple slots. This recycling could allow mixing of hot and cold data, which is undesirable.

4.3 Data Eviction mechanism

Rather than doing data eviction by tuples 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. From the tile group definition, once the tile group is full, it will be accessed only for reads, because new data cannot be added to it. However, if the data eviction selects randomly which data should be evicted, and data that is being frequently accessed is evicted, the DBMS performance will decrease drastically, for the access to this data becomes more costly, because the data movement between the main memory and secondary storage will need to happen more frequently.

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 avoid accessing unnecessary data when using the secondary storage, we also need to write out the column map, which maps the schema columns to the corresponding tile and offset.

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 the main memory as soon as the older transactions cease to use it. It starts when a given threshold is reached.

Algorithm 1 details the execution of the eviction process.

The eviction process retrieves the eviction candidates from the table (lines 1, 2),

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 in-memory table;
    end
end
```

writes the column map to a file (line 3) and computes the SMA for each attribute from the data present in the tile group (line 4). Then, for each tile (line 5), the data is added in the cuckoo filter, to allow proper filtering (line 6) and the tile is written out in secondary storage (line 7). Then, after this process is executed for each tile in the tile group, this tile group is then deleted from the main-memory storage, not being present in the table.

Figures 11, 12 and 13 present how the algorithm executes. Figure 11 shows a snapshot of the database before the algorithm executes. To aid the example, the tile group size is 5. Notice that the access pattern already divided each tile group in a tile layout, color coded for convenience. The first tile group (key range 1 to 5) is divided according to the colors ([key], [field0, field1, field2], [field3, field4], [field5], [field6, field7]); the second (key range 6 to 10) has the following layout:([key], [field0], [field1, field2, field3, field4, field5, field6, field7]) and so on. When the algorithm is executed, the SMAs are calculated and the data is inserted in the filter, as figure 12 displays. The filter has information about the evicted values, as they were applied to the hash functions, and the SMA table has the information regarding the evicted tile groups, which were deleted from the memory position. The SMA table schema is composed of TBL_ID, which is an internal identifier for the table, COL_ID which represents the offset of the column within the table, TG_ID, which is the tile group id, MIN, which is the minimum value in the data according to the combination and MAX, which is the maximum value in the data according to the combination. For example, the tuple $[1234, 0, 15, 1, 5]$ says that on table whose id is 1234, tile group whose id is 15, the minimum value for column 0 is 1 and the maximum is 5.

Figure 13 shows the organization of data in secondary storage. There are different files for each tile, and a file *_h* that contains the column map, which maps from the table schema column to the internal tile organization attribute. For example, tile group 16 on figure 11 is the second tile group. Following the color coding, there are 4 files. One with the column map, one with the values on the key attribute, one with the values of field0 and another with values of fields 1 to 7.

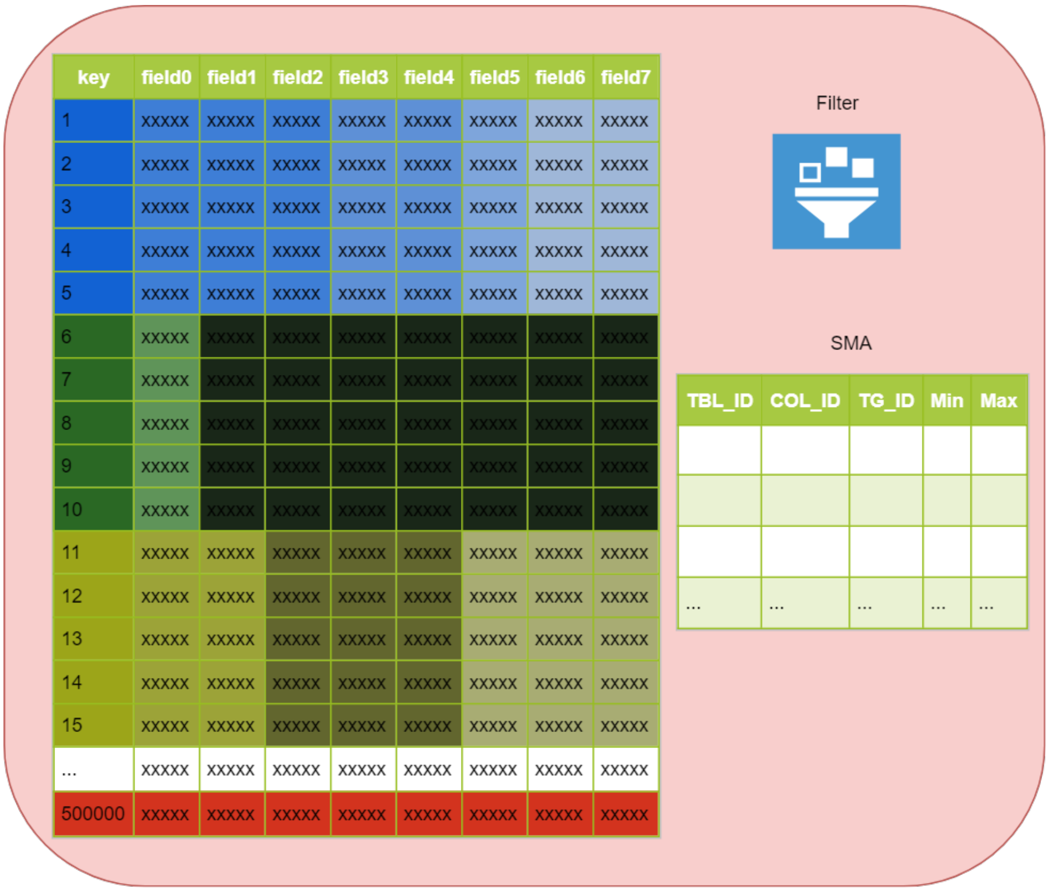
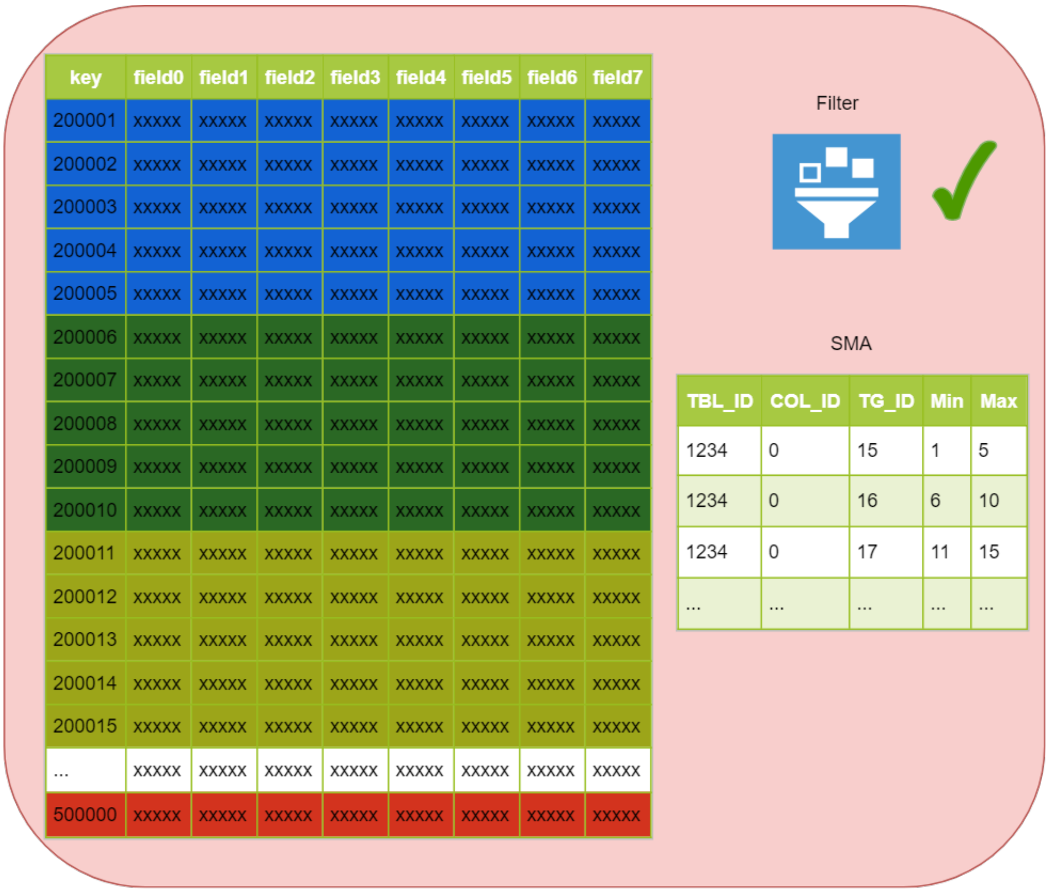


Figure 11 – In-memory table before data eviction

4.4 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 expelled tile groups are read-only and not modifiable, although they can be invalidated in-memory, in case of deletion. Section 4.8 describes the behavior of operations with the new architecture. The data in the tiles is stored together, in a sense that they are to be accessed together, as the workload suggests.



key	field0	field1	field2	field3	field4	field5	field6	field7
200001	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200002	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200003	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200004	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200005	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200006	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200007	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200008	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200009	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200010	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200011	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200012	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200013	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200014	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200015	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
...	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
500000	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx

TBL_ID	COL_ID	TG_ID	Min	Max
1234	0	15	1	5
1234	0	16	6	10
1234	0	17	11	15
...	...	...	...	...

Figure 12 – In-memory table after data eviction

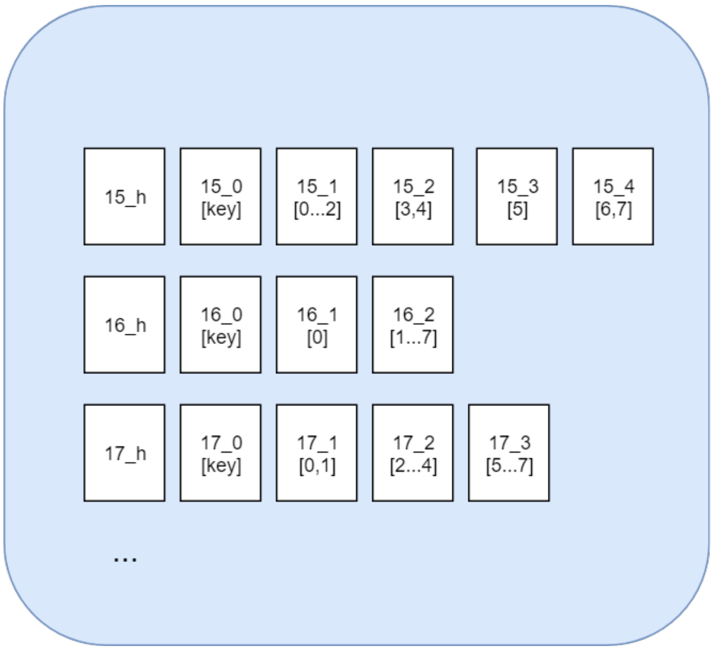


Figure 13 – Evicted data at secondary storage

4.5 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), 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. Further details can be found at chapter 2.

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.

4.6 Data Retrieval Mechanism

When a query is posed against the DBMS, it must try to answer the query with the memory resident data. Take one 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.

To consider an answer sufficient, the database must observe the nature of the query. If it accesses a unique column, or if the answer is only a query to check if a given element exists in the database, querying only part of the database can be sufficient to answer this query. In SmartLTM, a mechanism to evaluate the query was implemented, checking the table constraints to determine if a unique value was retrieved and avoid access to the cold storage.

From the probe, two scenarios are possible: If it is 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, be it excluded from the cold storage through an update or delete, and they are reassembled in a temporary in-memory table, that is disposed of when the transaction is completed. Algorithm 12 clarifies the data retrieval mechanism.

```
Data: Columns accessed, query predicate
Result: Temporary structure containing requested data
1  if predicate in filter then
2      candidate_tile_groups = SMA_table[predicate]
3  end
4  for tile_group in candidate_tile_groups do
5      retrieve column map;
6      tiles = column map[columns accessed];
7  end
8  for tile in tiles do
9      retrieve columns accessed;
10     create temp table;
11     add retrieved tuples to temp table;
12 end
```

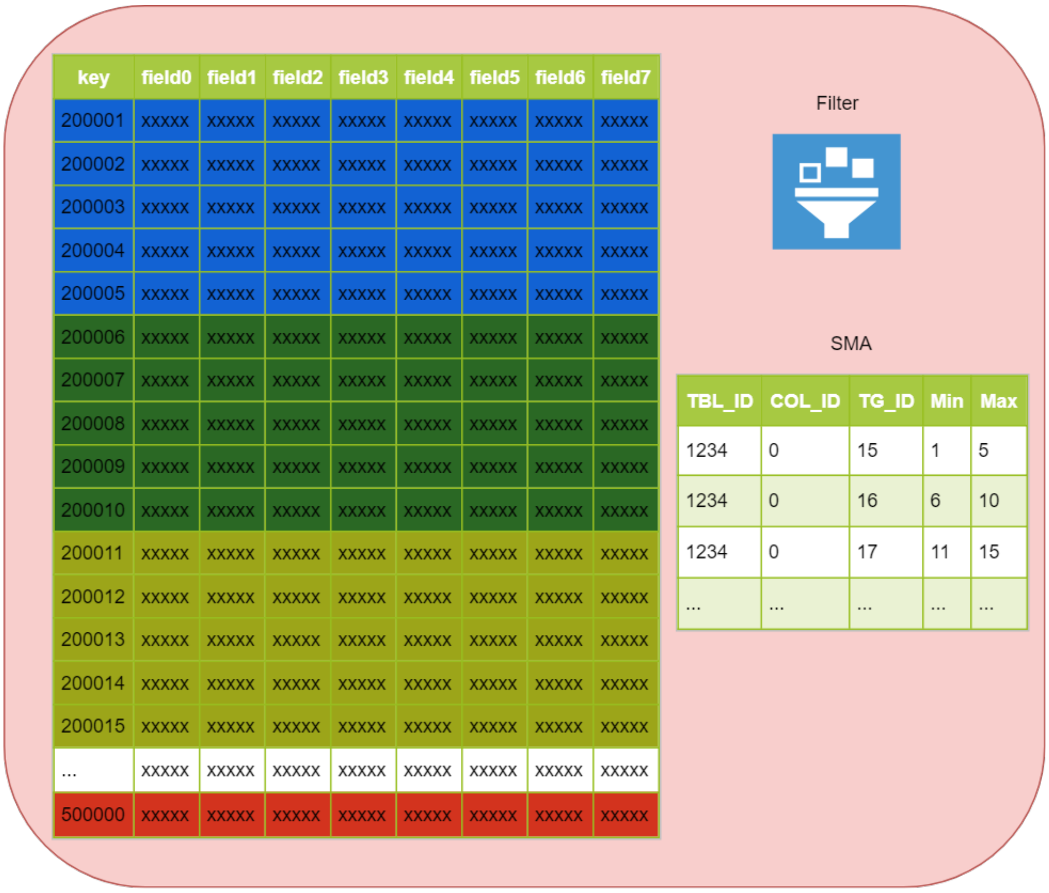
After the execution of the access filters, the candidate tile groups' column maps are retrieved (lines 1 to 3) and the required records are moved to memory, in a temp table (lines 4 to 6). While data is being retrieved from the cold storage, the current transaction waits for the data.

Figures 14, 15, 16, 17 present the retrieval process, using as example the following query:

```
SELECT field2, field3, field4 WHERE KEY = 13;
```

Figure 14 shows a current snapshot of in-memory data, and the engine cannot find the tuple with the given predicate. Then it probes the filter to check if this data may be in the cold storage. The filter answers positively, then, the SMA table is queried to find the candidate tile groups which may contain the requested data. From the query, the tile group with id equals to 17 is the only candidate, since the predicate $key = 13$ can only be in the $[11, 15]$ boundary. This process is presented in figure 15.

Then, the requested tile group column map is retrieved, to assess exactly which tiles should be retrieved back to memory. Figure 17 displays which tiles were selected to be retrieved, only the necessary data to answer this query.



key	field0	field1	field2	field3	field4	field5	field6	field7
200001	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200002	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200003	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200004	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200005	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200006	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200007	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200008	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200009	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200010	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200011	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200012	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200013	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200014	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200015	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
...	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
500000	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx

TBL_ID	COL_ID	TG_ID	Min	Max
1234	0	15	1	5
1234	0	16	6	10
1234	0	17	11	15
...	...	...	...	...

Figure 14 – In-memory table as query is posed

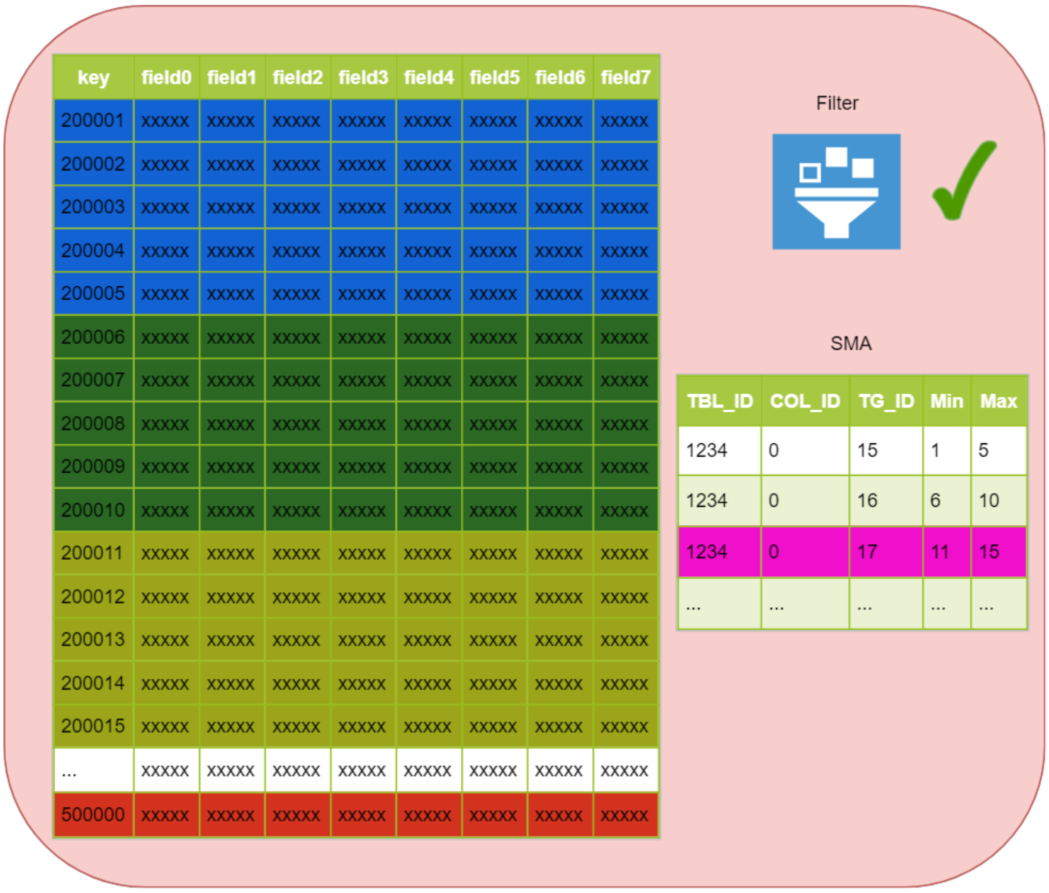
4.7 Discussion

By dividing possibly large files into smaller ones, following the organization from the DBMS, 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).

4.8 Integration with the DBMS

This section describes how the modifications added with SmartLTM affect the more basic operations of 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.



key	field0	field1	field2	field3	field4	field5	field6	field7
200001	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200002	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200003	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200004	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200005	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200006	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200007	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200008	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200009	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200010	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200011	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200012	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200013	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200014	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
200015	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
...	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
500000	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx

TBL_ID	COL_ID	TG_ID	Min	Max
1234	0	15	1	5
1234	0	16	6	10
1234	0	17	11	15
...	...	...	...	...

Figure 15 – Filter detection and candidate retrieval

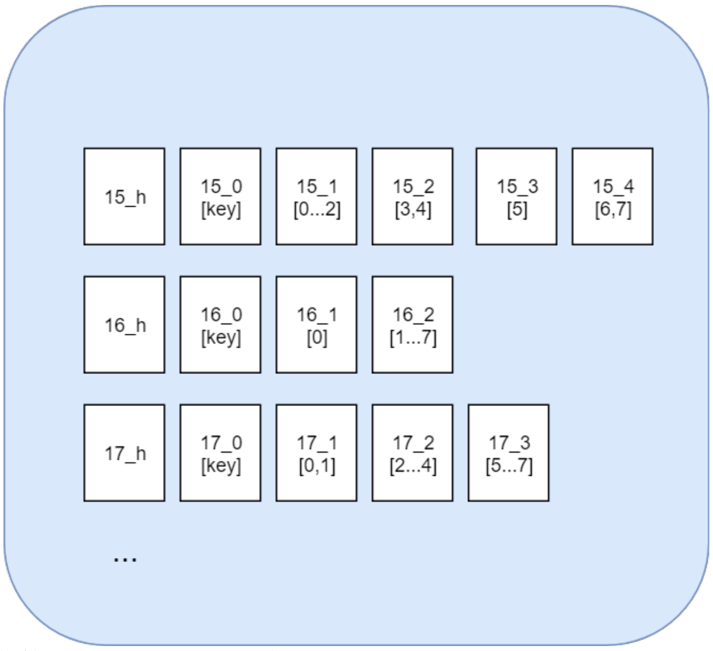


Figure 16 – Evicted tile groups at secondary storage

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

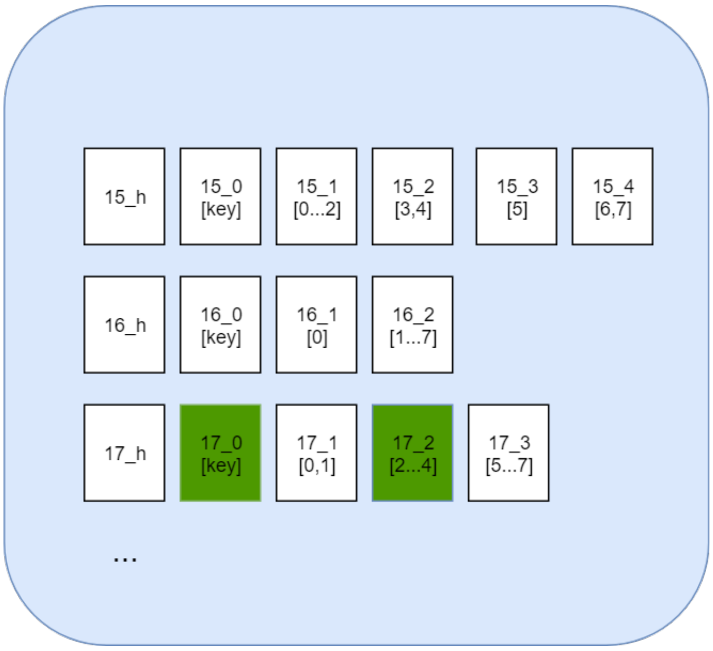


Figure 17 – Required data to be retrieved (in green)

secondary storage 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 4.6. Reads are also benefited from new data being only placed in main memory.

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.

4.9 Discussion

This chapter presents the concept for SmartLTM, the architecture, the algorithms for data eviction and retrieval and the interactions of the DBMS operations with this new architecture. Following, the experimental evaluation is presented, in which we compare SmartLTM with a row-based storage layout, present in the related works.

5 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.

5.1 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.

5.2 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 the following 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 FROM t WHERE KEY =` X`;`
- Q_1: `[KEY][f0][f1, f2, f3, f4, f5, f6, f7, f8, f9]`
- Q_2: `SELECT f2, f4 FROM t WHERE KEY =` X`;`
- Q_2: `[KEY][f0, f1][f2, f3, f4][f5, f6, f7, f8, f9]`
- Q_3: `SELECT f1, f2, f3 FROM t WHERE KEY =` X`;`
- Q_3: `[KEY][f0][f1, f2, f3][f4, f5, f6, f7, f8, f9]`
- Q_4: `SELECT f1, f2, f6, f7 FROM t WHERE KEY =` X`;`
- Q_4: `[KEY][f0][f1, f2][f3, f4, f5][f6, f7][f8, f9]`

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

For example, the layout generated by query Q_1 is similar to the one in figure 11, in the second tile group. Key and f0 are stored separately while the other fields are stored contiguously.

To have more control of cold and hot data accesses, we also added a uniform distribution to select key values, alongside the standard Zipfian (ZIPF, 1949). With a uniform distribution, cold storage access can be correctly estimated and it is directly proportional to the amount of data evicted to the cold storage.

5.3 Workloads

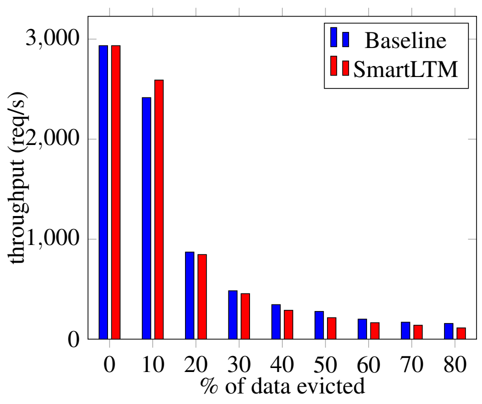
We defined four workloads to be executed, a read-only, an insert-only, a delete-only and an update-only. The scenario of data eviction to a secondary, slower storage medium shifts the burden to read queries, given that all modifying operations, like delete, insert and update, happen only in-memory, according to the mechanism proposed. This is also discussed in section 4.7. 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.

5.4 Experiments and Results

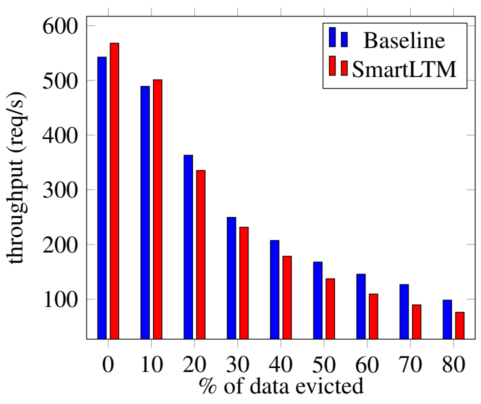
SmartLTM and the baseline are implemented inside Peloton. The main difference between SmartLTM and the baseline is how data is organized and how it is evicted. The baseline evicts the tile group completely, as a row-based layout, while our approach separates the tiles, as described above. Both implementations 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.

5.4.1 Read-only Queries

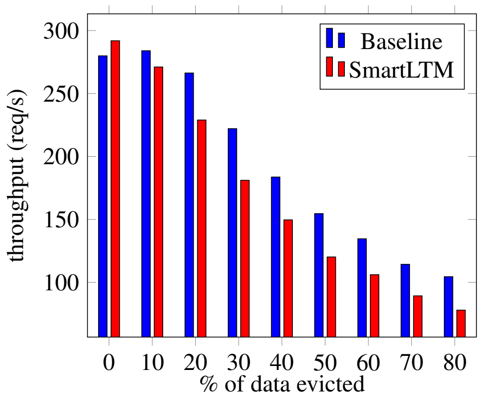
This experiment evaluates the impact of our approach with a read-only workload. Three different scale factors in YCSB were used, to evaluate how would the proposed approach scale: 50, 250 and 500. The access pattern is uniform, to allow accuracy when estimating hot data accesses and cold data accesses.



(a) 50k Tuples



(b) 250k Tuples



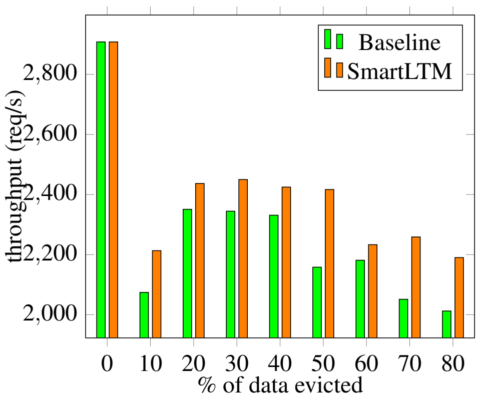
(c) 500k Tuples

Figure 18 – HDD results (higher is better)

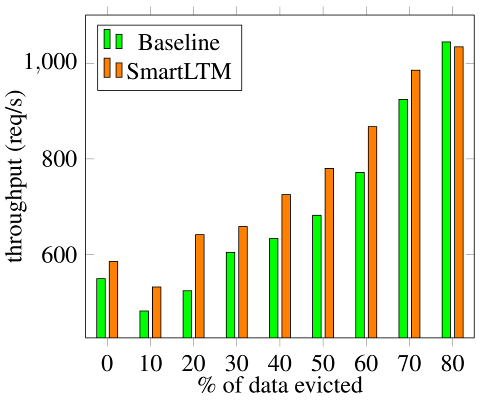
The HDD results are shown in figure 18. The x axis refers to the percentage of data evicted from the database and the y axis refers to the throughput measured. It is observable that the performance decrease is exponential, and that the baseline performs a little better than our proposed approach. The 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.

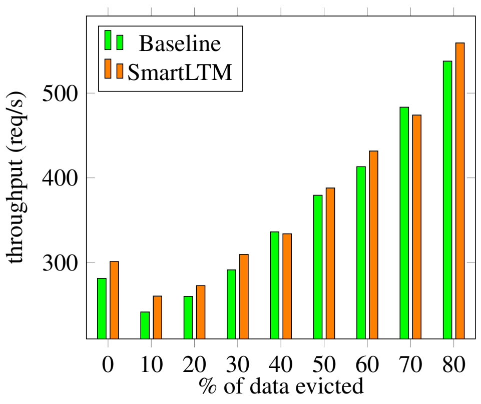
The SSD results are shown in figure 19. The x axis refers to the percentage of data evicted from the database and the y axis refers to the throughput measured. The higher access



(a) 50k Tuples



(b) 250k Tuples



(c) 500k Tuples

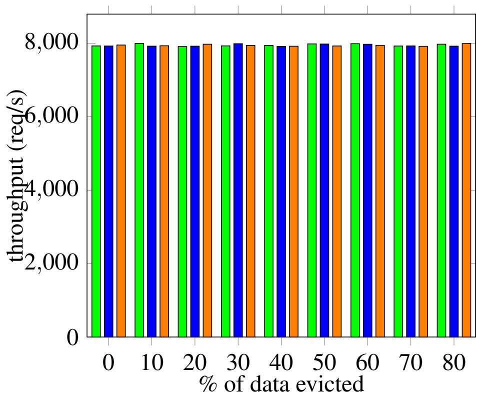
Figure 19 – SSD results (higher is better)

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 in figure 19a, 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 4.6, 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 chart, 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 chart.

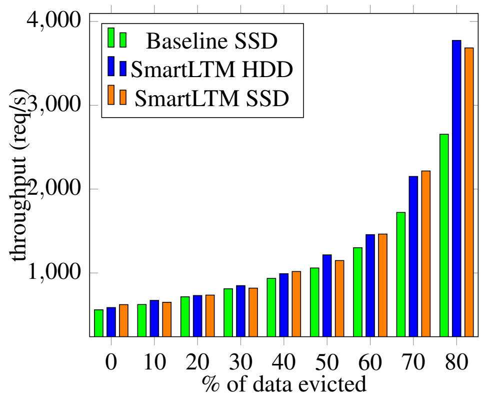
The random access optimization of SSDs makes clear that our approach is better than the baseline, especially where it most counts when the bottleneck becomes the cold data access. It can be verified that the throughput with our proposed approach almost doubles the one in the baseline, as more data is evicted, shown in figure 19a.

5.4.2 *Insert, Update and Delete queries*

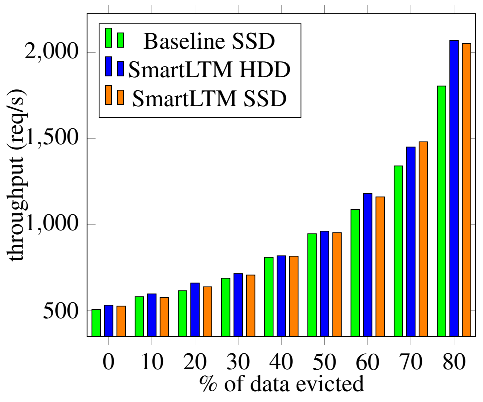
This experiment evaluates the impact of SmartLTM with insert, update and delete workloads. Since those operations do not care about the cold storage, only probing it through the access filters, only one scale factor in YCSB was used, 250. The access pattern is uniform, to allow accuracy when estimating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



(a) Insert



(b) Delete



(c) Update

Figure 20 – Insert, delete and update results

Figure 20 shows three separate results, the x axis refers to the percentage of data evicted from the database and the y axis refers to the throughput measured. The first 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 ignore 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 does not happen in the baseline.

5.4.3 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.

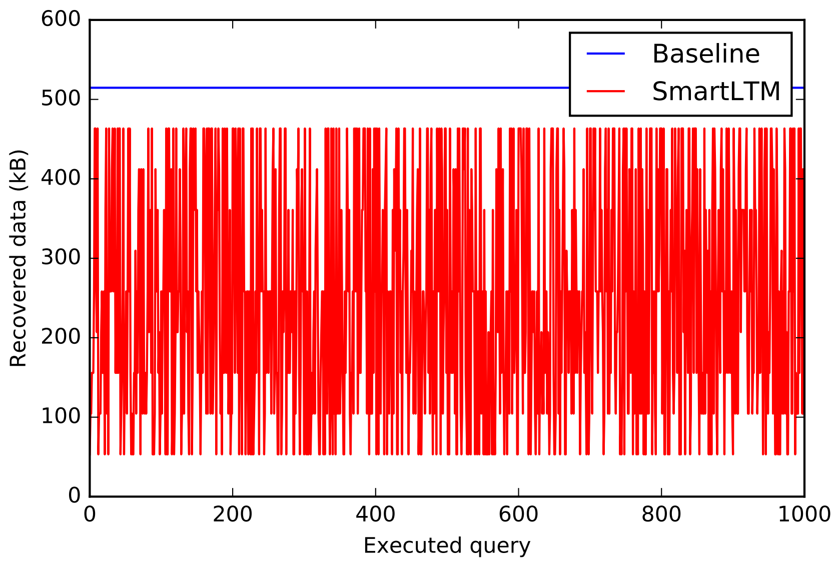


Figure 21 – Data retrieved per query (lower is better)

Figure 21 presents how much data is retrieved from the secondary storage per query when the queries posed require this data. The eviction percentage is 50% The x axis contains the number of the query and the y axis how much data was recovered to answer the query. It can be observed that for the baseline, the result is constant because all data is stored contiguously, in a row-store fashion. For SmartLTM, there are fluctuations depending on which query access each data, but no queries require all the data in a tuple to be retrieved in order to be answered, presenting the effectiveness in avoiding wasteful I/Os.

Figure 22 presents how much data is retrieved from the secondary storage over the execution of a workload. The eviction percentage is 50% The x axis represents how many queries were posed against the database that incurred in data retrieval and the y axis how much data was transferred. It is observable that during the execution, SmartLTM needs to retrieve less data to answer the queries. Over time, the effectiveness is visible.

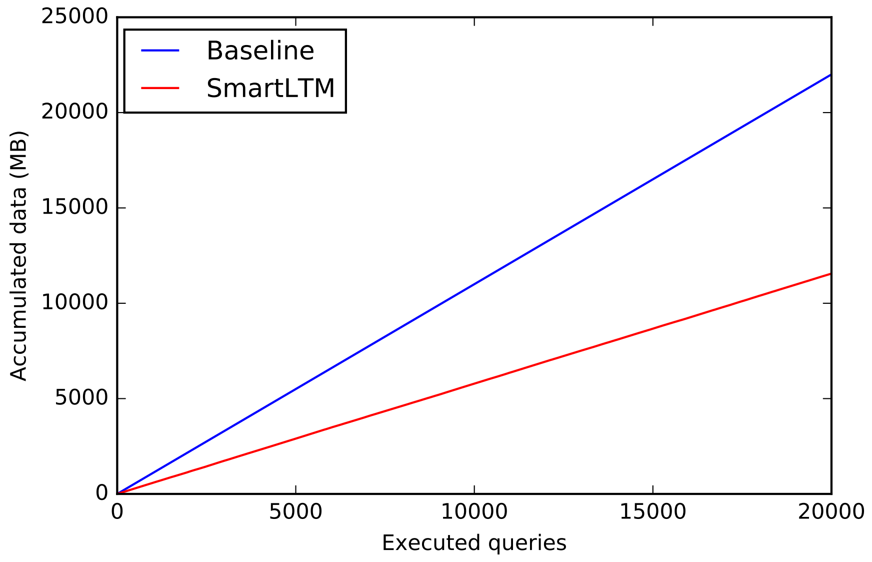


Figure 22 – Accumulated data (lower is better)

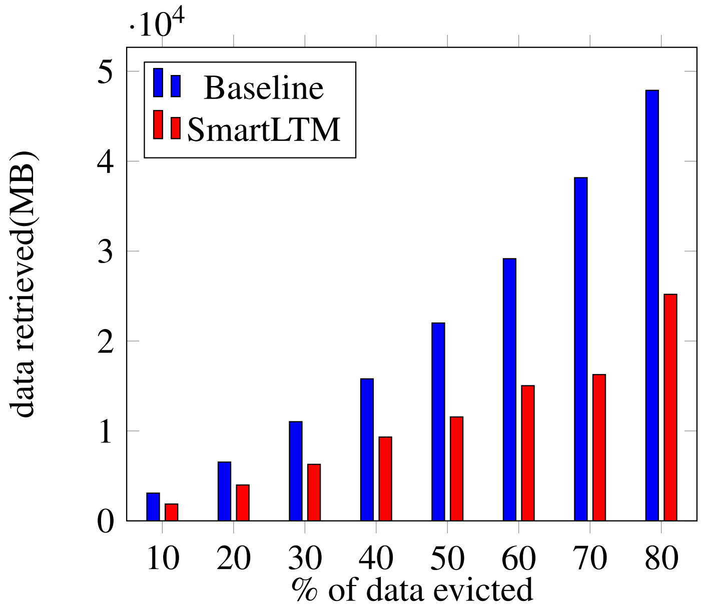


Figure 23 – Total data retrieved (lower is better)

Figure 23 summarizes how much data was retrieved from disk in each eviction percentage and compares it to the baseline. The x axis is the percentage of data evicted from the DBMS and the y axis is how much data was retrieved over the course of the benchmark. It is possible to see that overall, SmartLTM retrieve 50% less data to answer the queries, moving only meaningful data to the main memory.

This experiment shows clearly that our approach is effective regarding data retrieval, by avoiding useless data to answer queries, based on the data organization.

5.5 Conclusion

In this chapter we detailed the experiments and results obtained. Our starting hypothesis was that with an appropriate strategy, it would be possible to avoid wasteful I/O operations and incur an acceptable loss of throughput. Surprisingly, our experiments shown an increase of throughput in some scenarios, which can be explained by the reduction in search space, and, when evaluated with respect to the amount of data retrieved, our approach succeeded in retrieving around 50% less data to answer the same queries.

6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The main idea behind SmartLTM is that, if data must be evicted and was transformed beforehand by the DBMS, that the eviction mechanism should be aware and evict it in an optimal fashion, to facilitate data retrieval, avoiding wasteful I/Os. Besides that, the effective use of access filters, helps spare unnecessary operations in secondary storage. Our proposal is a new, smarter way of executing data eviction concerning HTAP databases. Those databases reorganize data layouts inside the table based on the query workload, to better perform. 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 experimental results show that SmartLTM is effective in its proposal, retrieving the data necessary to answer queries, without useless data, which does not implicate in extraneous overhead. By leveraging the work previously done by the DBMS, it is possible to better store this data.

Some restrictions and limitations were considered in this work. The most important is that we assume as a premise that the eviction candidates were selected by other component of the DBMS. Some works (ELDAWY *et al.*, 2014) (LEVANDOSKI *et al.*, 2013a); (FUNKE *et al.*, 2012) (LANG *et al.*, 2016) separate the two contributions, in a sense that while one depends on the other, they are independent in terms of evaluation. Other restriction was not using indexes. Although secondary indexes could be pruned of the removed main-memory entries, primary key and unique indexes would still require a way to ensure uniqueness in main memory and secondary storage, the reason why these entries cannot be simply erased and handled by the retrieval mechanism.

6.2 Future Work

As future work, a global cache can be implemented containing the most accessed tile groups, avoiding further accesses to cold storage. Keeping the cold storage in modern, byte-addressable non-volatile memories (NVRAM) can also dramatically improve performance, since the retrieval would not need to bring useless data contained in the current storage, which is block-addressable. Deletions happening in the cold storage can bring a scenario where we have mostly hollow tile groups. A strategy to compact tile groups with different tile layouts can also be developed, regarding which layout should be preserved when a compaction happens. Data compression can also be explored, incurring possible changes in the eviction algorithm, because of the different data type and organization.

Instead of making the upper layers completely unaware of the larger than memory architecture, the optimizer could leverage the information of data locality to better evaluate plans and suggest better execution.

Another avenue to be explored is failure recovery. Main memory databases start from the premise that all data was lost on a failure, however, with the larger than memory scenario, this premise is not true, because some data still exists in a persistent storage. New algorithms that take advantage of this effect can be explored as future work.

REFERENCES

ABADI, D. J.; MADDEN, S.; HACHEM, N. Column-stores vs. row-stores: how different are they really? In: **Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008**. [S.l.: s.n.], 2008. p. 967–980.

AILAMAKI, A.; DEWITT, D. J.; HILL, M. D. Data page layouts for relational databases on deep memory hierarchies. **VLDB J.**, v. 11, n. 3, p. 198–215, 2002.

ALAGIANNIS, I.; BOROVICA, R.; BRANCO, M.; IDREOS, S.; AILAMAKI, A. 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**. [S.l.: s.n.], 2012. p. 241–252.

ALAGIANNIS, I.; IDREOS, S.; AILAMAKI, A. H2O: a hands-free adaptive store - read. In: **International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014**. [S.l.: s.n.], 2014. p. 1103–1114.

APPUSWAMY, R.; KARPATHIOTAKIS, M.; POROBIC, D.; AILAMAKI, A. The case for heterogeneous HTAP. In: **CIDR**. [S.l.]: www.cidrdb.org, 2017.

ARULRAJ, J.; PAVLO, A.; MENON, P. 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**. [S.l.: s.n.], 2016. p. 583–598.

ARULRAJ, J.; PERRON, M.; PAVLO, A. Write-behind logging. **PVLDB**, v. 10, n. 4, p. 337–348, 2016.

BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. **Commun. ACM**, v. 13, n. 7, p. 422–426, 1970.

COOPER, B. F.; SILBERSTEIN, A.; TAM, E.; RAMAKRISHNAN, R.; SEARS, R. Benchmarking cloud serving systems with YCSB. In: **SoCC**. [S.l.]: ACM, 2010. p. 143–154.

DEBRABANT, J.; PAVLO, A.; TU, S.; STONEBRAKER, M.; ZDONIK, S. B. Anti-caching: A new approach to database management system architecture. **PVLDB**, v. 6, n. 14, p. 1942–1953, 2013.

DIACONU, C.; FREEDMAN, C.; ISMERT, E.; LARSON, P.; MITTAL, P.; STONECIPHER, R.; VERMA, N.; ZWILLING, M. Hekaton: SQL server's memory-optimized OLTP engine. In: **SIGMOD Conference**. [S.l.]: ACM, 2013. p. 1243–1254.

DIFALLAH, D. E.; PAVLO, A.; CURINO, C.; CUDRÉ-MAUROUX, P. Oltp-bench: An extensible testbed for benchmarking relational databases. **PVLDB**, v. 7, n. 4, p. 277–288, 2013.

ELDAWY, A.; LEVANDOSKI, J. J.; LARSON, P. Trekking through siberia: Managing cold data in a memory-optimized database. **PVLDB**, v. 7, n. 11, p. 931–942, 2014.

FAERBER, F.; KEMPER, A.; LARSON, P.; LEVANDOSKI, J. J.; NEUMANN, T.; PAVLO, A. Main memory database systems. **Foundations and Trends in Databases**, v. 8, n. 1-2, p. 1–130, 2017.

FAN, B.; ANDERSEN, D. G.; KAMINSKY, M.; MITZENMACHER, M. Cuckoo filter: Practically better than bloom. In: **CoNEXT**. [S.l.]: ACM, 2014. p. 75–88.

FUNKE, F.; KEMPER, A.; NEUMANN, T. Compacting transactional data in hybrid OLTP &amp; OLAP databases. **PVLDB**, v. 5, n. 11, p. 1424–1435, 2012.

GARCIA-MOLINA, H.; SALEM, K. Main memory database systems: An overview. **IEEE Trans. Knowl. Data Eng.**, v. 4, n. 6, p. 509–516, 1992.

GRUND, M.; KRÜGER, J.; PLATTNER, H.; ZEIER, A.; CUDRÉ-MAUROUX, P.; MADDEN, S. HYRISE - A main memory hybrid storage engine. **PVLDB**, v. 4, n. 2, p. 105–116, 2010.

HANKINS, R. A.; PATEL, J. M. Data morphing: An adaptive, cache-conscious storage technique. In: **VLDB**. [S.l.: s.n.], 2003. p. 417–428.

KEMPER, A.; NEUMANN, T. 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**. [S.l.: s.n.], 2011. p. 195–206.

LANG, H.; MÜHLBAUER, T.; FUNKE, F.; BONCZ, P. A.; NEUMANN, T.; KEMPER, A. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: **SIGMOD Conference**. [S.l.]: ACM, 2016. p. 311–326.

LARSON, P.; LEVANDOSKI, J. J. Modern main-memory database systems. **PVLDB**, v. 9, n. 13, p. 1609–1610, 2016.

LEE, B. C.; IPEK, E.; MUTLU, O.; BURGER, D. Phase change memory architecture and the quest for scalability. **Commun. ACM**, v. 53, n. 7, p. 99–106, 2010.

LEVANDOSKI, J. J.; LARSON, P.; STOICA, R. Identifying hot and cold data in main-memory databases. In: **ICDE**. [S.l.]: IEEE Computer Society, 2013. p. 26–37.

LEVANDOSKI, J. J.; LOMET, D. B.; SENGUPTA, S. The bw-tree: A b-tree for new hardware platforms. In: **ICDE**. [S.l.]: IEEE Computer Society, 2013. p. 302–313.

MA, L.; ARULRAJ, J.; ZHAO, S.; PAVLO, A.; DULLOOR, S. R.; GIARDINO, M. J.; PARKHURST, J.; GARDNER, J. L.; DOSHI, K.; ZDONIK, S. B. Larger-than-memory data management on modern storage hardware for in-memory OLTP database systems. In: **DaMoN**. [S.l.]: ACM, 2016. p. 9:1–9:7.

MALVIYA, N.; WEISBERG, A.; MADDEN, S.; STONEBRAKER, M. Rethinking main memory OLTP recovery. In: **ICDE**. [S.l.]: IEEE Computer Society, 2014. p. 604–615.

MOERKOTTE, G. Small materialized aggregates: A light weight index structure for data warehousing. In: **VLDB**. [S.l.]: Morgan Kaufmann, 1998. p. 476–487.

MOHAN, C.; HADERLE, D. J.; LINDSAY, B. G.; PIRAHESH, H.; SCHWARZ, P. M. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. **ACM Trans. Database Syst.**, v. 17, n. 1, p. 94–162, 1992.

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.; ZHANG, T. Self-driving database management systems. In: **CIDR**. [S.l.]: www.cidrdb.org, 2017.

R. Stanley Williams. **How We Found the Missing Memristor**. 2008. <\https://spectrum.ieee.org/semiconductors/processors/how-we-found-the-missing-memristor>. Accessed: 2018-09-03.

Rachel Courtland. **Spin Memory Shows Its Might**. 2014. <\https://spectrum.ieee.org/semiconductors/memory/spin-memory-shows-its-might>. Accessed: 2018-09-03.

RAMAKRISHNAN, R.; GEHRKE, J. **Database management systems (3. ed.)**. [S.l.]: McGraw-Hill, 2003.

WU, Y.; ARULRAJ, J.; LIN, J.; XIAN, R.; PAVLO, A. An empirical evaluation of in-memory multi-version concurrency control. **PVLDB**, v. 10, n. 7, p. 781–792, 2017.

ZIPF, G. K. **Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology**. [S.l.]: Addison-Wesley, 1949.