Enabling Efficient OS Paging for Main-Memory OLTP Databases

Radu Stoica
École Polytechnique Fédérale de Lausanne
radu.stoica@epfl.ch

Anastasia Ailamaki
École Polytechnique Fédérale de Lausanne
anastasia.ailamaki@epfl.ch

ABSTRACT

Even though main memory is becoming large enough to fit most OLTP databases, it may not always be the best option. OLTP workloads typically exhibit skewed access patterns where some records are hot (frequently accessed) but many records are cold (infrequently or never accessed). Therefore, it is more economical to store the coldest records on a fast secondary storage device such as a solid-state disk. However, main-memory DBMS have no knowledge of secondary storage, while traditional disk-based databases, designed for workloads where data resides on HDD, introduce too much overhead for the common case where the working set is memory resident.

In this paper, we propose a simple and low-overhead technique that enables main-memory databases to efficiently migrate cold data to secondary storage by relying on the OS's virtual memory paging mechanism. We propose to log accesses at the tuple level, process the access traces offline to identify relevant access patterns, and then transparently re-organize the in-memory data structures to reduce paging I/O and improve hit rates. The hot/cold data separation is performed on demand and incrementally through careful memory management, without any change to the underlying data structures. We validate experimentally the data re-organization proposal and show that OS paging can be efficient: a TPC-C database can grow two orders of magnitude larger than the available memory size without a noticeable impact on performance.

1. INTRODUCTION

Database systems have been traditionally designed under the assumption that most data is disk resident and is paged in and out of memory as needed. However, the drop in memory prices over the past 30 years led several database engines to optimize for the case when all data fits in memory. Examples include both research systems (MonetDB[6], H-Store[13], Hyper[14], Hyrise [11]) and commercial systems (Oracle's TimesTen[2], IBM's SolidDB [1], VoltDB[4], SAP Hana[7]). Such systems assume all data resides in RAM and explicitly eliminate traditional disk optimizations that are deemed to introduce an unacceptable large overhead.

Today storage technology trends have changed - along with the large main memory of modern servers, we have solid-state drives (SSDs) that can support hundreds of thousands of IOPS. In recent years, the capacity improvements of NAND flash-based devices outpaced DRAM capacity growth, accompanied by corresponding trends in cost per gigabyte. In addition, major hardware manufacturers are investing in competing solid-state technologies such as Phase-Change Memory.

In OLTP workloads, record accesses tend to be skewed: some records are "hot" and accessed frequently (the working set), others are "cold" and accessed infrequently. Hot records should reside in memory to guarantee good performance; cold records, however, can be moved to cheaper external solid-state storage.

Ideally, we want a DBMS that: a) has high performance, characteristic of main-memory databases, when the working set fits in RAM, and b) the ability of a traditional disk-based database engine to keep cold data on the larger and cheaper storage media, while supporting, as efficiently as possible, the infrequent cases where data needs to be retrieved from secondary storage.

There is no straightforward solution as DBMS engines optimized for in-memory processing have no knowledge of secondary storage. Traditional storage optimizations, such as the buffer pool abstraction, the page organization, certain data structures (e.g. B-Trees), and ARIES-style logging are explicitly discarded in order to reduce overhead. At the same time, switching back to a traditional disk-based DBMS would forfeit the fast processing benefits of main-memory systems.

In this paper, we take the first step toward enabling a main-memory database to migrate data to a larger and cheaper secondary storage. We propose to record accesses during normal system runtime at the tuple level (possibly using sampling to reduce overhead) and process the access traces off the critical path of query execution in order to identify data worth storing in memory. Based on the access statistics, the relational data structures are re-organized such that the hot tuples are stored as compactly as possible, which leads to improved main memory hit rates and to reduced OS paging I/O. The data re-organization is unintrusive since only the location of tuples in memory changes, while the internal pointer structure of existing data-structures and query execution remain unaffected. In addition, the data

DaMoN '13

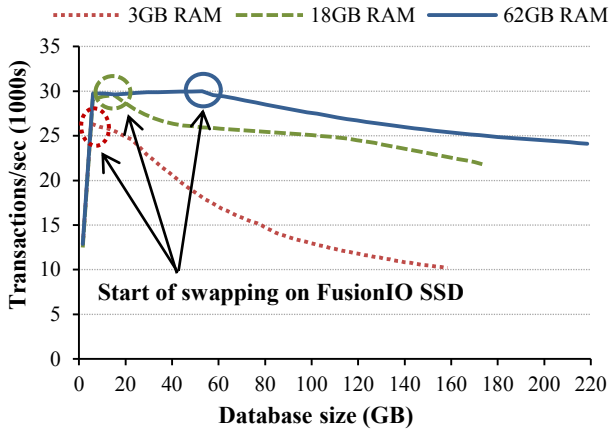


Figure 1: VoltDB throughput when paging to a SSD

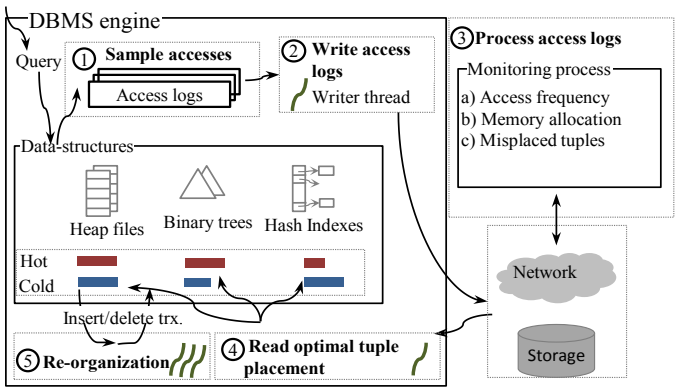


Figure 2: System Architecture

re-organization is performed incrementally and only when required by the workload.

We implement the data re-organization proposal in a state-of-art, open source main-memory database, VoltDB [4]. Our experimental results show that a TPC-C database can grow 50× larger than the available DRAM memory without a noticeable impact on performance or latency, and present micro-benchmark results that indicate that a system can deliver reasonable performance even in cases where the working set does not fully fit in memory and secondary storage data is frequently accessed.

In this paper we make the following **contributions**:

1. We profile the performance of a state-of-art main-memory DBMS and identify its inefficiencies in moving data to secondary storage.

2. We propose an unintrusive data re-organization strategy that separates hot from cold data with minimal overhead and allows the OS to efficiently page data to a fast solid-state storage device.

3. We implement the data re-organization technique in a state-of-art database and show that it can support a TPC-C dataset that grows 50× bigger than the available physical memory without a significant impact on throughput or latency.

The remaining of this document is organized as follows. Section 2 details the motivation for our work; Section 3 describes the architecture of the data re-organization proposal, which is validated experimentally in Section 4; Section 5 surveys the related work and, finally, Section 6 concludes.

2. MOTIVATION

Our work is motivated by several considerations: i) by hardware trends that make storing data on solid-state storage attractive; ii) by the workload characteristics of OLTP databases; and iii) by the inefficiencies of existing systems, either traditional disk-based databases or newer main-memory optimized DBMS.

2.1 Hardware trends

It is significantly cheaper to store cold data on secondary storage rather than in DRAM. In previous work [17] we computed an updated version of the 5-minute rule [10] and found that it is more economically to store a 200B tuple on a SSD if it is accessed less than approximately once every 100 minutes. In addition to the price, the maximum memory capacity of a server or the memory density can both pose challenges. Today, a high-end workstation can fits at most 4TB of main memory, servers can handle around 256GB of DRAM per CPU socket, while a 1U rack slot hosts less than 512GB of DRAM.

2.2 Skewed Accesses in OLTP workloads

Real-life transactional workloads typically exhibit considerable access skew. For example, package tracking workloads for companies such as UPS or FedEx exhibit time-correlated skew. Records for a new package are frequently updated until delivery, then used for analysis for some time, and after are accessed only on rare occasions. Another example is the natural skew found on large e-commerce sites such as Amazon, where some items are much more popular than others are or where some users are much more active than the average. Such skewed accesses may change over time but typically not very rapidly (the access skews might shift over days rather than seconds).

2.3 Existing DBMS Architectures

2.3.1 *Disk-based DBMS.*

One possible option is to use a disk-based database architecture that optimizes for the case data is on secondary storage. Traditional DBMS pack data in fixed-size pages, use logical pointers (e.g. page IDs and record offsets) that allows the buffer pool abstraction to move data between memory and storage transparently, and have specialized page replacement algorithms to maximize hit rates and reduce I/O latency.

However, given current memory sizes and the lower latency of SSDs compared to HDDs, such optimizations might not be desirable. Stonebraker et al. [20] introduced a main-memory database that is two orders of magnitude faster than a general purpose disk-based DBMS, while Harizopoulos et al. [12] showed that for transactional workloads the buffer pool introduces a significant amount of CPU overhead: more than 30% of execution time is wasted due to extra indirection layer. This overhead does not even include latching buffer pool pages at every tuple access, or the overhead of maintaining page-level statistics to implement the page replacement algorithm.

2.3.2 Default OS paging

At the other extreme, a straightforward way of extending available memory is to keep main-memory databases unchanged and simply use the default OS paging. Unfortunately, we find such an approach to cause unpredictable performance degradation.

We show in Figure 1 the throughput of the VoltDB DBMS when runing the TPC-C benchmark [3] (please see Section 4 for the detailed experimental setup). The working set size of the TPC-C benchmark is constant, while the size of the database grows over time as new records are inserted in the *Order*, *NewOrder*, *OrderLine*, and *History* tables. We vary the amount of physical DRAM available to the OS and enable swapping to a 160GB Fusion ioDrive PCIe SSD [9]. The scale factor is 16 (the initial database occupies 1.6GB) and in the beginning all data is memory resident. As the database size grows, the RAM available is exhausted, and the OS starts paging virtual memory pages that it deems cold to the SSD.

Figure 1 shows the trade-off between extending the memory size of a system by using flash memory and the performance penalty incurred. When extending the memory budget from 62GB to 220GB (3.5× increase), the throughput drops by 20%; increasing the main memory from 18GB to 176GB (9.8× increase), the throughput decreases by 26%; finally, when extending the memory size from 3GB to 161GB (53× increase), the throughput penalty is 66%. The results are sub-optimal as the TPC-C working set fits in memory even if the database grows. Clearly, the performance impact of paging is significant and needs consideration.

3. DATA RE-ORGANIZATION

Our data re-organization proposal, as depicted in Figure 2, is composed of five independent steps: First, tuple level accesses are logged as part of query execution (phase one); then, the access logs are shipped (phase two) for the processing stage that identifies which tuples are hot or cold (phase three); finally, the output of the processing stage is propagated back to the database engine (phase four) that performs the data re-organization when needed (phase five). The rest of this section details each step of the data re-organization process.

① **Sample accesses.** In the first phase, each worker thread samples probabilistically tuple accesses and writes the access records to a dedicated log-structure (a circular buffer). Each worker thread has one such log-structure in order to avoid any synchronization-related overhead. Time is split in discreet quanta, each time quantum receiving a single time stamp to further reduce overhead and reduce the access log size.

Each access log quantum has the following syntax: <*TimeStamp, AccessRecord**>, while an *AccessRecord* is composed of <*ObjectID,* [Key | $TupleID$], *FileOffset*>. The *ObjectID* is an identifier that represents the relational object data-structure containing the tuple (can be either a table or an index). The next field, [Key | $TupleID$], identifies the tuple accessed; we use the primary table key or indexing key, denoted as Key, for this purpose. However, the primary key can be replaced with the tuple ID in systems that such support such identificators. The last field, *FileOffset*, presents the memory offset of the tuple relative to the beginning of the relational data-structure. *FileOffset* is used during the processing phase to determine if the location of a tuple matches its access frequency (i.e. whether the position of a tuple matches its cold/hot status). The total length of an *AccessRecord* is $1B + 4B + 8B = 13B$.

A quantum of 1s is used as it is sufficiently fine grained to differentiate between tuples with different access frequencies at the time granularity of interest (as mentioned, a cold tuple should be moved to a SSD if accessed less than once every tens to hundreds of minutes). We experimented with multiple sampling probabilities and decided to use a default sampling probability of 10% (we log at most 10% of the total tuple accesses) that offers a good tradeoff between logging overhead and precision.

② **Write access logs.** A single dedicated thread writes the access logs either to a file or to the network. The writer thread has three functions. Firstly, it decouples transaction execution from any other I/O or communication overhead. Secondly, it throttles the rate at which the logs are generated to insure minimal system interference. If the access logs are not flushed fast enough and fill up, the backpressure will cause worker threads to reduce the sampling frequency. Thirdly, the design allows us to decouple the database engine from the later processing stage of the access logs.

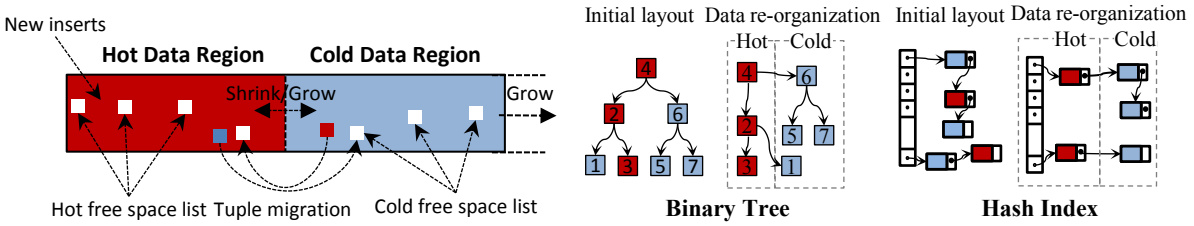
③ **Process access logs.** The access traces are processed offline to identify data placement inefficiencies. The analysis step takes place ideally on a separate server to avoid interference with ongoing query execution. The output of the processing step is: a) a memory budget for each data-structure (table or index), and b) two lists of hot/cold misplaced tuples for each object. A hot misplaced tuple list identifies which tuples are accessed often enough to justify storing them in memory and are not yet placed in the hot memory region of the object, while the cold misplaced tuple list represents the tuples that are the best candidates for eviction from the hot memory region of each object.

The access traces are processed as follows:

Step 1 - estimate access frequency. The access frequency is estimated using a variant of the Exponential Smoothing algorithm [17], although any other standard algorithms such as ARC[18], CAR [5], or LRU-K [19] variants can be used. The advantages of the Exponential Smoothing algorithm over other frequency estimation algorithms are twofold: i) it estimates more accurately access frequency, resulting in higher hit rates, and ii) it is efficient as it supports parallelization and minimizes the number of access records processed (by scanning the access logs in reverse time order).

Step 2 - compute optimal memory allocation. The access frequencies alone are not sufficient to decide what data should be stored in memory – we also need to consider the size of each tuple to maximize the *access density*. We use the average tuple size corresponding to each object to normalize the tuple access frequency. After the normalization, the hot tuples of each object can be identified. The memory budget of each object is simply the number of tuples qualifying as hot multiplied by the average tuple size.

Step 3 - identify misplaced tuples. Finally, we identify misplaced tuples based on the memory allocation of each object. As mentioned, each tuple access contains the relative memory offset of the tuple. The relational data structures are allocated contiguous in memory (as described below), thus we can identify if the access frequency of a tuple matches its location by simply comparing the tuple's relative memory offset with the hot memory budget of the relation object.



(a) Memory allocation strategy (b) Node re-organization for binary trees and hash indexes

Figure 3: Memory allocation and hot/cold data placement

④ **Read optimal tuple placement.** A dedicated thread reads the output of the processing step. It first reads and sets the target memory budget for each object, then incrementally reads the lists of misplaced tuples for each object as needed by the tuple re-organization.

⑤ **Re-organization.** Each data-structure is allocated memory sequentially in the virtual address space of the database process using the facilities of the OS. The memory layout of a data-structure is presented in Figure 3(a). The beginning portion of the data file is logically considered "hot" and is pinned in memory. The rest of the file is left to contend for system RAM and uses the default OS paging policy. The OS is left to manage at least 10% of available memory to insure there is enough memory for other processes.

The tuple re-organization is performed without changing the pointer structure or record formant of the relational objects – we change only how memory is allocated as depicted in Figure 3(b). The memory allocator of each data-structure has two free-space lists, one for the hot memory region, and the other for the cold memory region. When inserting, the data-structure can request either hot or cold memory; when deleting, memory is reclaimed by comparing the memory offset with the hot/cold threshold and appending the memory region to the corresponding free space list. A tuple movement can be logically thought of as a delete operation of the misplaced tuple followed by its re-insertion in the appropriate memory region.

Overall, the key insights of our technique are as follows: i) we decouple, to the extent possible, all data re-organization operations (logging, analysis, re-organization) from the critical path of query execution, ii) we optimize data locality such that hot tuples are stored compactly in a contiguous memory region, and iii) we restrict OS paging decisions by preventing code memory sections and the hot data regions of relational objects from being paged out.

4. EVALUATION

In this section, we evaluate experimentally how effective is the data re-organization technique in reducing OS paging overhead. We first demonstrate the end-to-end performance of a main-memory DBMS when running a TPC-C benchmark where the majority of the database resides on an SSD. We show that the data re-organization strategy and paging-related I/O have little impact on the overall system throughput or latency even when the database size outgrows the RAM size by a factor of more than 50×. As a TPC-C database has a predictable working set and data growth, we then explore the impact of a full workload change on the overall system performance through a micro-benchmark.

4.1 Experimental Setup

DBMS. We use the open-source commercial VoltDB [4] DBMS (version 2.7) running on Linux to implement the data re-organization technique. VoltDB uses data partitioning to handle concurrency and insure scaling with the number of cores. Each worker thread has its own set of data structures (both tables and indexes) that it can access independently of the other worker threads. Thus, each logical table is composed of several physical tables, each worker thread being assigned one such physical table. In all experiments, the database is partitioned to match the number of available cores.

Memory management. The core VoltDB functionality is implemented in C++, while the front-end (network connectivity, serialization of results, query plan generation, DBMS API) are implemented in Java. The memory overhead of the front-end is independent of the database size and we found it crucial to keep memory resident the front-end related objects and code. We pin in memory all code mappings, front-end data structures, and Java heap and allow only the large relational data-structures (tables, hash and binary tree indexes) to be paged out to the SSD. We track which address ranges correspond to code sections or to other data-structures by examining the */proc* process information pseudo-file system. For relational objects, we allocate memory sequentially in the virtual address space of the database process by using the *mmap()*-related system calls and pin/unpin pages in memory using the *mlock()*/*munlock()* system calls.

We note that the VoltDB core engine already optimizes memory allocation. VoltDB allocates memory in large contiguous chunks for each object in order to reduce *malloc()* call overhead and reduce memory fragmentation. Each relational data-structure has its own memory allocation pool and memory chunks never contain data from more than one table or index, which insures that virtual memory pages contain data of the same type. Therefore, the performance benefits of the data re-organization stem only from taking into account access frequency rather than from other memory management optimizations.

Query execution. Transactions run as stored procedures (no query optimization is performed at runtime), with query execution and result externalization being handled by separate threads. The benchmark client, responsible for submitting transactions, is placed on a different machine on the same 10Gb local network. For throughput results, we make sure the server is fully loaded by maintaining sufficient in-flight transactions and measure throughput every second in all experiments. The network bandwidth is never saturated and the network latency does not affect any throughput re-

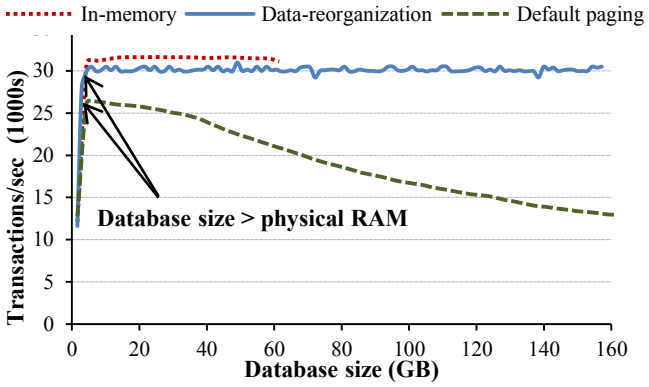


Figure 4: TPC-C throughput.

Trx.Name	Server latency(μs)				Client latency(μs)			
	No Paging		Paging		No Paging		Paging	
	avg	σ	avg	σ	avg	σ	avg	σ
NewOrder	283	135	318	182	3270	5430	3301	5509
OrderStatus	82	26	126	58	2950	5486	2990	5499
Payment	149	52	183	107	3027	5458	3094	5492
Delivery	314	152	347	214	3317	5331	3358	5513
StockLevel	797	243	898	348	3531	5372	3611	5570

avg = average, σ = standard deviation

Table 1: TPC-C transaction latency

sults due to the batch-style processing architecture.

Hardware. In all our experiments we use a 4 socket Quad-Core AMD Opteron (16 cores in total) equipped with 64GB of physical DRAM (the amount of memory visible to the OS varies according to each experiment). The paging device is a single 160GB FusionIO PCIe SSD that can support, according to our measurements, up to 65,000 4kB random read IOPS and between 20,000 (long term throughput) and 75,000 (peak throughput) 4kB random write IOPS.

4.2 TPC-C Results

We use the TPC-C benchmark [3] to validate the overall efficiency of the data re-organization strategy. The scaling factor is set to 16 to match the number of cores and the database is partitioned on the warehouse ID (standard TPC-C partitioning), where each worker thread is responsible for executing all transactions related to its given warehouse. To maximize the number of growing tables and the database growing speed, we disable for the *Delivery* transaction the deletions of fulfilled orders from the *NewOrder* table.

4.2.1 Throughput

We measure the TPC-C transaction execution throughput in three scenarios. In the first case, we run the unmodified VoltDB database with all of the 64GB of RAM available to the OS (∼62GB were useful for actual data storage). In the second scenario, we run TPC-C on the same unmodified engine but restrict the amount of physical memory to 5GB (∼3GB useful memory) and turn on swapping to the SSD. In the third case, we run VoltDB modified with our data reorganization technique and maintain the same memory configuration (∼3GB of useful memory); the memory mapped data files are backed directly by the SSD without an in-between file-system.

As shown in Figure 4, the hot/cold data separation stabilizes throughput within 7% of the in-memory performance, although the actual data stored grows 50× larger than the amount of RAM available. On the other hand, the throughput of the unmodified engine drops by 66% when swapping to the SSD.

4.2.2 Latency

Paging potentially introduces I/O operations in the critical path of a transaction and a relevant question is how SSD I/O changes transaction latency. We repeat the same TPC-C experiments, only this time we throttle the maximum number of transaction at 50% of the maximum throughput (to prevent transaction latency from including queuing times).

We show in Table 1 the transaction latency as measured from both the server- and client- side and compute for each transaction its standard deviation. As shown, paging increases the average transaction latency by ∼50μs and its standard deviation by ∼100μs. However, the variability increase is small relative to the total transaction execution time and becomes insignificant from the client perspective. The transaction latency, as experienced by the client, is dominated not by the actual transaction execution but rather by the software overhead of submitting a transaction to the server, by the overhead of the network I/O, and finally by transaction management on the server side. To put the values into perspective, the advertised I/O latency for the Fusion ioDrive SSD is 25μs and the network latency (round-trip wire latency plus switching) is ∼30μs.

4.3 Response to workload changes

The TPC-C working set size is fixed (the size of the hot data does not change over time) and only 4 tables out of 9 are growing. In addition, accesses show a predictable time correlation: the newest tuples in the growing tables are also the hottest. We expect that in most workloads such a time correlation exits, nonetheless we want to understand the stability of the system if the working set shifts in unpredictable ways or if the working set does not fully fit in RAM.

To answer these questions, we develop a micro-benchmark with a dual purpose: to measure performance for the case where the working set does not fully fit in memory, and to test how fast the system reacts to workload changes. We generate a database composed of 10^8 200B tuples indexed by a 4B integer primary key. The total memory footprint of the VoltDB DBMS (the table, index, plus VoltDB memory overhead) is 26.2GB. The database size is constant and the available memory is to 5GB (physical memory is ∼20% of the total database size). We execute a single query that retrieves single tuples of the form:

SELECT * FROM R WHERE PK = <ID> ;

The *ID* values for the select query are generated according to a Zipfian distribution with skew factor 1 (80% of accesses target 20% of data). To measure how fast the system adapts to a workload change, we randomize the set of hot tuples

while preserving the same access distribution and measure how throughput varies over time.

We show in Figure 5 the impact of the working set shift on the throughput of the system. We distinguish five distinct phases. The initial steady-state (phase 1) represents the long term behavior of the system when the table and index data structures are fully optimized according to the access pattern. Once the workload shift occurs, the performance of the system is immediately affected (phase 2) and drops by 96% from 75,000 to 4,500 queries/sec. The OS page replacement algorithm first reacts to the workload shift (phase 3) as the hottest pages from the cold file portions are identified and buffered in memory. As the access distribution contains significant skew, even the small memory budget available to the OS is enough to capture the most frequently accessed pages. As a result, throughput improves to around 25,000 queries per second. The throughput remains stable (phase 4) until the data re-organization starts to relocate tuples (with the configurable delay of 60s). Once, the data relocation commences, the throughput quickly stabilizes to the original level of 65,000 queries/sec (phase 5).

We note that the unmodified VoltDB system is unable to execute any queries. The high memory pressure combined with the lack of a clear working set cause a high OS paging activity that renders the database unresponsive.

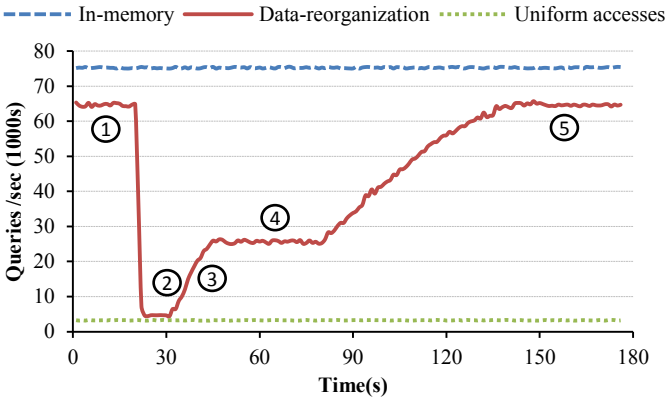


Figure 5: Throughput for a shifting Zipfian working set.

5. RELATED WORK

There has been much work on exploring OLTP engine architectures optimized for main-memory access. Research prototypes include [6, 13, 14, 11, 15] and already there are a multitude of commercial main-memory offerings [2, 1, 4, 7]. Such engines assume that the entire database fits in memory, completely dropping the concept of secondary storage. We diverge from the memory-only philosophy by considering a scenario where a main-memory database may migrate cold data to secondary storage by using OS paging and investigate how to efficiently separate hot from cold data in order to improve memory hit rate and reduce I/O.

The work closest to ours is HyPer's cold-data management scheme [8]. In Hyper, hot transactional data is identified and separated from cold data that might be only referenced by analytical queries. Once identified and separated, the cold data is compressed to reduce its memory footprint in a read-optimized format suitable for the analytical queries. However, Hyper's data re-organization scheme has a different goal, namely to reduce the overhead of creating new OLAP threads. This is achieved by minimizing the number of pages that are actively modified by the OLTP threads and by compressing and storing the OLAP-only data on large virtual memory pages. Hyper also takes a different approach for implementing the data re-organization: it keeps access statistics at a larger granularity (at the VM page level) by overriding the OS' virtual memory infrastructure, and uses an online heuristic to separate hot from cold data as an integral part of query execution.

Compression techniques can reduce the memory footprint of a database and can help in supporting larger in-memory datasets. Compression, however, is an orthogonal topic. Compressing relational data, without changing the tuple order inside the underlying data-structure, does not create a separation of cold from hot data, but rather can be viewed as a technique for expanding the available system memory. The incentives of separating data based on the access frequency remain the same. In addition, if data is stored based on its access frequency, the hot and cold datasets can be compressed through different techniques. A similar idea is used in [8] where only the cold data is compressed.

6. CONCLUSIONS

Our data re-organization proposal is generally applicable as it does not change the pointer structure of the physical data-structures, or the concurrency mechanism of the database engine. The hot/cold data separation can be simply thought of a better way of allocating memory by taking into account access frequency, while the tuple re-organization can be modeled as short search/delete/insert transaction of individual tuples. At the same time, our proposal can be suboptimal: some data structures maintain invariants that might force accessing cold data along the way of retrieving hot tuples. For example, the hash index proposed in [16] uses linear hashing, i.e. maintains tuples sorted on keys inside the hash buckets to allow incremental expansion of the hash index. Our data re-organization technique cannot distinguish between logically cold and hot data as it relies on physical accesses; cold tuples in a hash bucket accessed on the way to the target hot tuple are considered as hot as the target tuple. Possible solutions for such cases could be either to change the original data-structures or to use two different data stores, one data store for the hot and the other store for the cold data. Unfortunately, both approaches have deeper architectural implications.

We proposed a simple solution for a main-memory database to efficiently page cold data to secondary storage. Our technique logs accesses at the tuple level, processes the access logs offline to identify hot data and re-organizes accordingly the in-memory data structures to reduce paging I/O and improve memory hit rates. The hot/cold data separation is performed on demand and incrementally through careful memory management, without any change to the underlying data structures. Our experimental results show that it is feasible to rely on the OS virtual memory paging mechanism to move data between memory and secondary storage even if the database size grows much larger than the main memory. In addition, the hot/cold data re-organization offers reasonable performance even in the cases where the working set does not fully fit in memory and can adapt in a time frame of a few minutes to full workload shifts.

7. REFERENCES

[1] IBM SolidDB. Available: http://www.ibm.com.

[2] Oracle TimesTen In-Memory Database. Information available at: http://www.oracle.com.

[3] Transaction Processing Performance Council TPC-C Standard Specification. Available: `http://www.tpc.org/tpcc/spec/tpcc_current.pdf` .

[4] VoltDB In-Memory Database. Available: http://www.voltdb.com.

[5] S. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *FAST*, 2004.

[6] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[7] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *ACM Sigmod Record*, 2012.

[8] F. Funke, A. Kemper, and T. Neumann. Compacting transactional data in hybrid OLTP & OLAP databases. In *VLDB*, 2012.

[9] Fusion IO. Technical specifications. Available: `http://www.fusionio.com/PDFs/Fusion%20Specsheet.pdf`.

[10] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. 1987.

[11] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: a main memory hybrid storage engine. In *VLDB*, 2010.

[12] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[13] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. In *VLDB*, 2008.

[14] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[15] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. In *VLDB*, 2011.

[16] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. In *VLDB*, 2011.

[17] J. Levandoski, P.-A. Larson, and R. Stoica. Identifying Hot and Cold data in Main-Memory Databases. In *ICDE*, 2013.

[18] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, 2003.

[19] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. 1993.

[20] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *VLDB*, 2007.