OS Paging and Buffer Management

Using the Virtual-Memory Paging Mechanism to Balance Hot/Cold Data in Memory and HDD

Jana Lampe

TU Kaiserslautern

LGIS Seminar SS 2014
Optimizing Data Management on New Hardware

1 Introduction

1.1 Motivation

Traditional database systems follow the design principle that all data is stored on hard disk where records are accessible via indexing structures and loaded into main-memory as soon as they are requested. Since memory prices have dropped significantly during the last three decades, new database systems have been designed that allow faster access to the data by storing all data inside the main-memory, e.g. H-Store [9].

Notwithstanding the above, also durable storage technologies have made considerable progress towards larger capacity, more input/output operations per second as well as reduced cost per storage unit at the same time. Solid-state drives (SSDs) have entered the market and, using the non-volatile NAND-flash technology, convince by their reading and writing speed, although they are still about ten times more expensive than common HDDs. Another promising technology is phase-change memory (PCM), but it is still in research state.

Now one could ask, why all the data is stored in memory. If there are records that are not accessed that frequently, would it not be cheaper to store them on a secondary storage device such as HDD or SSD? Moreover, energy consumption would decrease and more space would be available to temporarily store intermediate results thereby accelerating query processing.

To answer this question, Gray and Putzolu proposed in [5] the 5-minute rule, which says, that a record should be kept in memory if it has accessed at least every 5 minutes (the break-even interval). The n-minute rule is defined as follows:

$$BEInterval = \frac{StoragePrice[\$]}{IO/s} \cdot \frac{1}{ItemSize[byte] \cdot RAMPrice[\$/byte]} \quad (1)$$

Levandoski et al. derived in [7] an updated version of the 5 minute rule, where the break-even interval is increased to 60 minutes given a record size of 200 bytes, when an SSD is used. Under the assumption, that SSD prices will continue to fall, the 60 minutes are not a final value, but will still diminish as well.

Let us now consider OLTP workloads. OLTP is an acronym for "online transaction processing" and means that these applications are characterized by relatively short running transactions, which access in general only very limited data volume [6].

These workloads often have a certain characteristic: on one side, there are datasets that are accessed recurrently (so called "hot" records), on the other side, there are datasets which are accessed only sporadically ("cold" records) and in between some "lukewarm" records [7].

As an example, imagine a flight booking agency, where customers, e.g. individuals and companies make reservations for flights of a certain airline. If a new flight is inserted into the database, it will typically by "hot" in the beginning when reservations are made and cancelled. Once the flight is accomplished, no reservations are performed anymore. Hence, after a while this flight will become a "cold" record.

In brief, an OLTP workload seems to be a good application for the use of secondary storage in main-memory database systems.

1.2 Content Overview

The aim of this paper is to investigate to what extent we can improve the efficiency of main-memory databases by means of the virtual paging mechanism to balance hot and cold data in memory and HDD. In order to answer this central question, we will mainly concentrate on the approach taken by Stoica et al. in [8].

Therefore, we will first consider the architecture of the database system, developed by Stoica et al., and the basic technologies they use in their approach. In the second part, we will examine the mode of operation of this database engine, thereby introducing the "exponential smoothing algorithm" developed by Levandoski et al. [7] and its application. The last part is about the experimental results of Stoica et al.'s approach.

1.3 Related Work

In contribution [4] Funke et al. present a technique that separates hot from cold data. As HyPer is a one-size-fits-all database system, their goal is a different one. They define cold data as data which is not modified, but still read by OLAP queries. The classification is done by hardware-assisted monitoring in order to keep the number of modified pages as small as possible. Secondary storage is only used if the database size exceeds the memory's capacity. Thus, they do not want to migrate cold to secondary storage, instead, they compress it in order to enlarge the available memory.

Hekaton [7] is an extension of SQL Server that enables the storage of entire relations in memory. The focus of "Project Siberia" is to extend the Hekaton system in a way that cold records can be relocated to secondary storage. But this still is ongoing research.

DeBrabant et al. [3] switched the traditional storage hierarchy around, i.e., they come up with a new model, where main-memory is the principal storage and only cold records are moved to a so-called "Anti-Cache", if space is needed. The main difference is that they do not make use of the OS virtual paging mechanism.

2 Architecture and Basic Technologies

Stoica et al. built a new database architecture by adding secondary storage to a main-memory DBMS and using the OS paging mechanism to transfer the data. Since they used VoltDB, we are going to take a closer look at this DBMS in the first part of this chapter. Then we will discuss the virtual paging mechanism in general and how it affects a database system. In the third part, we are going to discuss the extensions that are necessary to enable the efficient use of secondary storage as proposed in [8].

2.1 VoltDB

VoltDB [1] is an open-source commercial relational database management system, which utilizes main memory as storage and can be installed on a single machine or distributed on a cluster.

Data access is serialized, hence there is no need for locking, latching and buffer management. Instead to improve scalability and performance, VoltDB makes use of vertical partitioning and replication, e.g., small read-only tables can be replicated to other partitions, whereas large (logical) tables, where single records are mostly accessed via primary key, can be partitioned into several different physical tables on different nodes. Finally, there exists one worker thread for each physical table, which maintains its own data structures, namely index structures and physical tables. If data from different partitions is needed, one node has to coordinate the transaction by distributing the requests to the other nodes, collecting the intermediate results and finally computing the complete result.

Durability is guaranteed by a snapshot mechanism which can be configured by the user, as writing snapshots to a disk inhibits the database's performance. In addition to that, all commands can be logged to disk. So whenever the database crashes, it can be restored by using the latest snapshot and replaying the command log.

Furthermore, there exists support for realtime analytics, Hadoop integration and many more features, but these are not in the scope of this paper.

2.2 The Virtual Paging Mechanism

If the memory is exceeded, the operating system starts to look for areas that have not been accessed for a while and migrates their content to hard disk (or more general, to the next larger storage device). The benefit is, that space is freed in memory, However, if this is not done in an efficient way, the OS has to start copying pages from memory to hard disk and back. This process heavily slows down the performance, since much more additional I/O operations have to be performed on disk, where they are much more expensive than in memory.

In databases, pages contain numerous tuples which are not uniformly accessed. Therefore, a standard virtual paging mechanism would not be sufficient in case that the database size is bigger than the memory size. In consequence, Stoica et al. implemented a data reorganization strategy, that places hot tuples

next to the other hot tuples and cold tuples next to cold tuples respectively. How this strategy works in detail is discussed in section 3.

2.3 The modified VoltDB

First and foremost, Stoica et al. added an SSD as secondary storage device to the database. Therefore they had to modify the memory management in such a way that only tables and indexes can be moved to the secondary storage. VoltDB provides an interface to get information about the address ranges of code sections and other data structures. To apply the data reorganization strategy, it is crucial that tables and indexes are stored sequentially in the virtual address space. VoltDB supports this by assigning large memory chunks to the relational objects and forbidding that these contain records of other data structures. Thus, the pages of the virtual memory always contain equally typed data. The *mlock()/munlock()* methods then enable the user to make the system hold or release appointed pages in memory. They are therefore used by Stoica et al. to keep the hot pages and the front-end-related data structures in memory.

Pages, that are not fixed in memory, are then left to the OS virtual paging mechanism.

Besides the memory management, Stoica et al. implemented an additional component, that analyzes which records are classified as hot and which not using an additional log. This component is independent of the DBMS engine and should run on a seperate machine.

The resulting architecture is shown by figure 1.

3 Mode of Operation

As already mentioned in the beginning, the main idea of Stoica et al. [8] is to store infrequently used (cold) data on cheaper secondary storage. In the first part of this section, the utilized classification algorithm is presented. The second part comprises its application to a given DBMS engine, as proposed in [8].

3.1 How to identify hot and cold data in main memory?

In other words, the issue is to retrieve the K hottest records out of a large database in an efficient way. To achieve this, Levandoski et al. [7] present two different approaches, namely a *forward* and a *backward* algorithm. In a second step, these two algorithms are parallelized in order two improve performance.

Before the two basic algorithms are presented, some preparatory issues have to be discussed, starting with the database engine on which the algorithms have been developed. Levandoski et al. used Hekaton, an engine which has to be combined with SQL server. Its aim is to enable efficient handling of OLTP workloads. It is up to the user to decide, if he wants to have his table managed by Hekaton which means, that it is then entirely stored in main memory. Moreover, the records are arranged on record level – not on page level – and can be accessed by means of hash indexes or ordered indexes using range scans or index lookups [7]. The Hekaton engine allows to combine Hekaton tables with normal tables. In contrast to VoltDB, Hekaton does not make use of partitioning. Instead, it allows any thread to access any record without the use of locks and latches but guaranteeing no disturbance between transactions by "a new optimistic, multi-version concurrency control technique" [7].

Another question that has to be answered is how to perform the analysis of the record accesses. There are two different possibilities: *inline* and *offline*. If the analysis' results, i.e., the estimations of the record's access frequency, are kept in main memory and updated each time a record is accessed. Alternatively, the record accesses are written to a seperate log and analyzed later. Levandoski et al. preferred this approach, since they considered caching to impose to much overhead, whereas a simple logging mechanism does not have much impact on the database's performance. In addition to that, the database engine does not have to be changed and there are more possibilities to adapt the system to special requirements, as it is not prescribed "when, where and how to analyze the log and estimate access frequencies". They suggest to do the analysis on a separate machine. Stoica et al. took up this idea, as we have already seen in section 2.3.

However, it might not be necessary to log every single access. As an alternative, one could speed up the analysis computation by sampling. This could for example be done by defining a fixed probability p. Each worker thread has to perform a Bernoulli experiment every time it recognizes record access, to log this access with probability p or not (with probability $1 - p$). This will result in a sampling rate of p. Levandoski et al. derived from experiments that a sampling rate of 10% still delivers good results being 2,5% less precise.

Finally, in order to compute the estimated access frequency $ext_r(t_n)$ of a record r at time slice t_n, Levandoski et al. made use of expoential smoothing as depicted in equation 2, because in their experiments, this approach was more accurate than the caching techniques LRU-2 and ARC and thus led to less misplaced records and better performance of the system.

$$ext_r(t_n) = \alpha \cdot \chi_{t_n} + (1 - \alpha) \cdot est_r(t_{n-1}) \tag{2}$$

In order to understand, this equation, we first have to understand the notion of "time slice" used by Levandoski et al. According to them, a time slice $[t_n, t_{n+1})$ is defined as the interval between two record accesses, i.e., the record accesses determine when a new time slice begins. In the equation above and in the following, the time slice $[t_n, t_{n+1})$ is denoted by the starting timestamp t_n for simplicity. In the log, the record ids are stored in the order of access, time markers delimit the time slices' boundaries. χ_{t_n} is the characteristic function of the event "r has been accessed during t_n", α is a parameter which weights how significant new observations are towards old estimates, the higher it is, the more relevant are recent accesses. Typically, α is bounded by $[0.01, 0.05]$ [7].

Forward algorithm The "Forward algorithm" is the simplest approach to determine the K hottest records. It scans the log from a beginning time slice t_0 into forward direction and applies the exponential smoothing equation in a slightly modified way:

$$est_r(t_n) = \alpha + est_r(t_{prev}) \cdot (1 - \alpha)^{t_n - t_{prev}} \tag{3}$$

Instead of always considering the previous time slice for all record, overhead is reduced by using only the time slice t_prev when r was accessed last. Hence, the estimated value of a record is only updated when this record is updated and the characteristic function χ_{t_n} becomes superfluous as it always evaluates to 1 in this setting. The other summand of the equation is normalized by the exponent $t_n - t_{prev}$.

In a last step, the estimations of the records are ranked and the K highest-estimated records are classified as "hot", all others as "cold".

To improve the performance of this algorithm, parallelization is taken into account. Levandoski et al. present two different approaches.

On the one hand, the log could be split into several partitions by a hash function on the record ids. The estimation and the ranking are then done on all partitions independently and, finally, the best K results of all these partitions together are classified as "hot".

On the other hand, the log could by partitioned based on the time slices. To understand this idea, we examine the non-recursive version of the exponential smoothing equation

$$est_r(t_n) = \alpha \sum_{i=t_0}^{t_n} \chi_{t_{n-1}} (1 - \alpha)^{i-1} + (1 - \alpha)^{t_n} \tag{4}$$

We can see that, by partitioning the log such that each partition consists of successive time slices, only one summand is dependent on a result of the previous partition. The rest can be be computed by a dedicated worker thread on each partition in the serial manner. These partial results must then be aggregated in an additional step and the ranking step finishes the estimation.

Backward algorithm The "Forward Algorithm" has two major disadvantages: "it requires a scan of the entire log and it requires storage proportional to the number of unique record ids in the acces log" [7]. To overcome these, Levandoski et al. developed a backward algorithm that scans the log in reverse order and updates a record's backward estimation $estb_r$ at each time slice t_n where it is accessed. Once more, the exponential smoothing equation has to be adapted.

$$estb_r(t_n) = \alpha(1-\alpha)^{t_e - t_n} + estb_r(t_{last}) \tag{5}$$

$estb_r(t_{last})$ denotes the previous estimation at time t_{last} when an access of r was recognized in the log last before t_n, so $t_{last} > t_n$. t_e designates the last time slice in the log, where the analysis is started.

Furthermore, two estimates have to be maintained which are shown by the equations 6 and 7.

$$upEst_r(t_n) = estb_r(t_n) + (1-\alpha)^{t_e - t_n + 1} \tag{6}$$

$$loEst_r(t_n) = estb_r(t_n) + (1-\alpha)^{t_e - t_0 + 1} \tag{7}$$

Equation 6 adds the current estimation of record r to the value the forward estimation would have, if r was accessed in all time slices, i.e., the time slices, that have not been evaluated by the backward algorithm so far. What we get, is the highest, still possible access frequency that record r can have.

Analogously, equation (7) computes the lowest access frequency estimation value, that record r can still have at time slice t_n, assuming that is not contained in the log any more.

So why are these estimates necessary? If the upper bound of a record r_i is lower than the lower bounds of min. K other records, the log entries of r_i can be skipped. Furthermore, if only K records remain, the algorithm can stop without having analyzed the whole log and knowing the exact estimation values.

Still, the algorithm can be accelerated by means of parallelism. In this case, partitioning by time slices would not make sence, as one main goal of the "Backward Algorithm" is not to scan the whole log. Therefore, Levandoski et al. suggest to partition the log by means of a hash function by record id. Several worker threads perform the backward algorithm on the different partitions. One dedicated controller thread then manages the execution, requesting the upper and lower bounds and telling the workers which record ids to skip and when to stop processing the log.

3.2 The data reorganization technique by Stoica et al. [8]

The data reorganization technique consists of five separate steps shown by figure 1. Its main goal is to prevent that hot records are paged out to the secondary storage by the operating system.

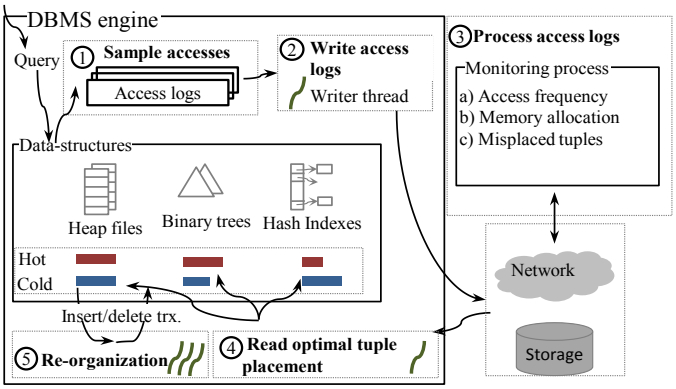

Fig. 1. System Architecture [8]

1st step: Sample accesses In order to be able to perform the analysis, tuple accesses have to be sampled. As the system is based on VoltDB, where each worker thread is responsible for its own partition, Stoica et al. decided that every worker thread should have its own log structure – in their implementation a circular buffer. Like Levandoski et al. they decided to split time into discrete slices, here called time quanta. Each time quantum is indicated by a time stamp.

An access log entry consists of a time stamp and some information about the accessed record, i.e., the id of the data structure that contains this record, the key of it, given by the primary key or indexing key and a so called "FileOffset". Instead of a primary key, also a direct tuple id can be used, if this is supported by the database system. This offset field contains the memory offset from the record to the beginning of the respective data structure. It is needed to determine if the record's position matches it's assignment to be part of the hot or the cold region.

Stoica et al. confirmed the 10% sampling rate suggested by [7] by their own experiments. Nevertheless, they made some slight modification: the sampling rate can be reduced, if too many log entries have to be written and the worker thread is at risk to be braked by a full log.

2nd step: Write access logs One dedicated thread is responsible for writing the access logs obtained from the 1st step either to a file or to the network. In

section 2.3, we have already mentioned that the processing of the access log is done by a separate component, which is independent from the database engine and performs the analysis offline, as suggested by Levandoski et al. In brief, this writer thread passes the log data to the external component in a manner that the transactions executed on the database are not disturbed by additional I/O overhead.

3rd step: Process access logs Stoica et al. suggest that the analysis should be done on an external server in order to avoid that the computations might reduce the query execution performance.

This step uses the exponential smoothing algorithm that we have already discussed in the previous section, more precisely the parallel backward algorithm is used, as Stoica et al. considered it to be more accurate than other caching algorithms and because it is more efficient for the reasons already mentioned in section 3.1.

Having the estimations of the access frequencies, a normalization with respect to the average tuple size of each object is done. This way, the so called "access density" is maximized. Subsequently, the hot tuples of each object can be distinguished and the memory budget for each object can be computed as the product of the number of hot tuples and the average tuple size.

A misplaced tuple is then determined by comparing its memory offset to the memory budget of the object, the tuple is contained in, since the memory is allocated adjacently for relational objects.

The output of this step is a list containing the memory budget of each data structure, i.e., tables and indexes, and for each object a list of hot misplaced tuples (tuples that are not stored in memory though their access frequency estimation is high enough) and respectively a list of cold misplaced tuples list.

4th step: Read optimal tuple placement The goal of this step is to prepare the re-organization step by first reading and setting the target memory budget of all objects. Afterwards, the lists of misplaced records are analyzed and passed to the next step object by object.

5th step: Re-organization Figure 2 depicts how the data structures are allocated in the virtual address space.

The head of a data structure is always part of the hot region and therefore always stays in memory. The operating system controls then the remaining part of this data structure according to its native paging policy.

Each data structure has a memory allocator that maintains a so called hot free-space list and a cold free-space list. If a tuple is inserted, it can either be placed inside the hot or inside the cold region. On a tuple deletion, a comparison of the tuples offset with the hot/cold threshold determines, whether the tuple was part of the hot or the cold region and the memory budget of this tuple as well as its location is added to the respective free-space list.

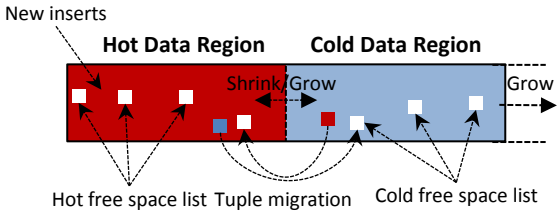

Fig. 2. The memory allocation strategy [8]

From a logical point of view, moving a tuple inside this data structure is a deletion from the wrong region followed by a re-insert operation into the right region by help of the free-space lists.

Figure 3 illustrates that the pointer structures of the logical index structures are not altered, the only thing that changes is the placement of their entries inside the memory.

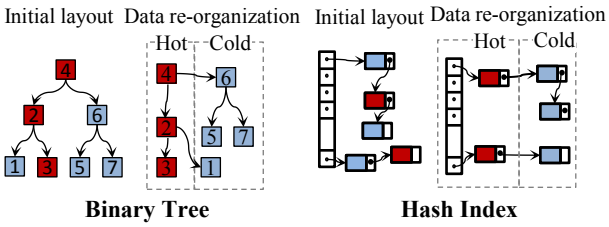

Fig. 3. Data re-organization for binary trees and hash indexes [8]

4 Evaluation

The new system architecture proposed by Stoica et al. has been tested against the TPC-C benchmark and a micro-benchmark they developed on their own.

4.1 The experiment's configuration

The DBMS used by Stoica et al. together with its extensions has already been considered in section 2.

In the implementation, all transactions are executed as stored procedures and, hence, queries are not optimized while running. There are different dedicated threads for query execution and for externalizing the results.

The system was run on a 4 socket Quad-Core AMD Opteron with 16 cores having a physical DRAM of 64GB. As secondary storage, a 160GB FusionIO PCIe SSD was used. According to [8], it can handle up to 65,000 4kB random read IOPS and 20,000 to 75,000 4kB random write IOPS.

Benchmark client and database are located within one 10Gb local network, but run on different machines. Stoica et al. assured that this did not affect the measurement results.

4.2 The TPC-C benchmark

The TPC-C benchmark [2] has been designed to measure the performance of OLTP systems. As application example, it simulates an order-entry environment of a company. Figure 4 gives an overview of what the TPC-C database scheme looks like.

The benchmark itself consists of 5 different transaction types, which can be executed in parallel on the database, for example the "New-Order Transaction": Each of these transactions inserts a complete new order with 5 to 15 items into the database. Each of these items has to be checked for disposability in the *Stock*-table. The other four transactions types are "Payment Transactions" (entering payments of a customer, thereby updating the statistics in the *District*- and the *Warehouse*-table), "Order-Status Transactions" (read-only transactions that check a particular customer's order status) , "Delivery Transactions" (process ten orders from the *New-Order* relation in batch mode, and delete them afterwards) and "Stock-Level Transactions" (read-only transactions that supervise the stock of recently ordered products) [6].

The TPC-C benchmark uses then the two following metrics.

1. The Performance: This metric is measured by the so called "Maximum Qualified Throughput" (MQTh) which is given by the throughput of *new-order*-transactions per minute.
2. The Price/performance: Here the overall cost for the system within three years (i.e., initial price, maintanance, energy consumption) is divided by the performance.

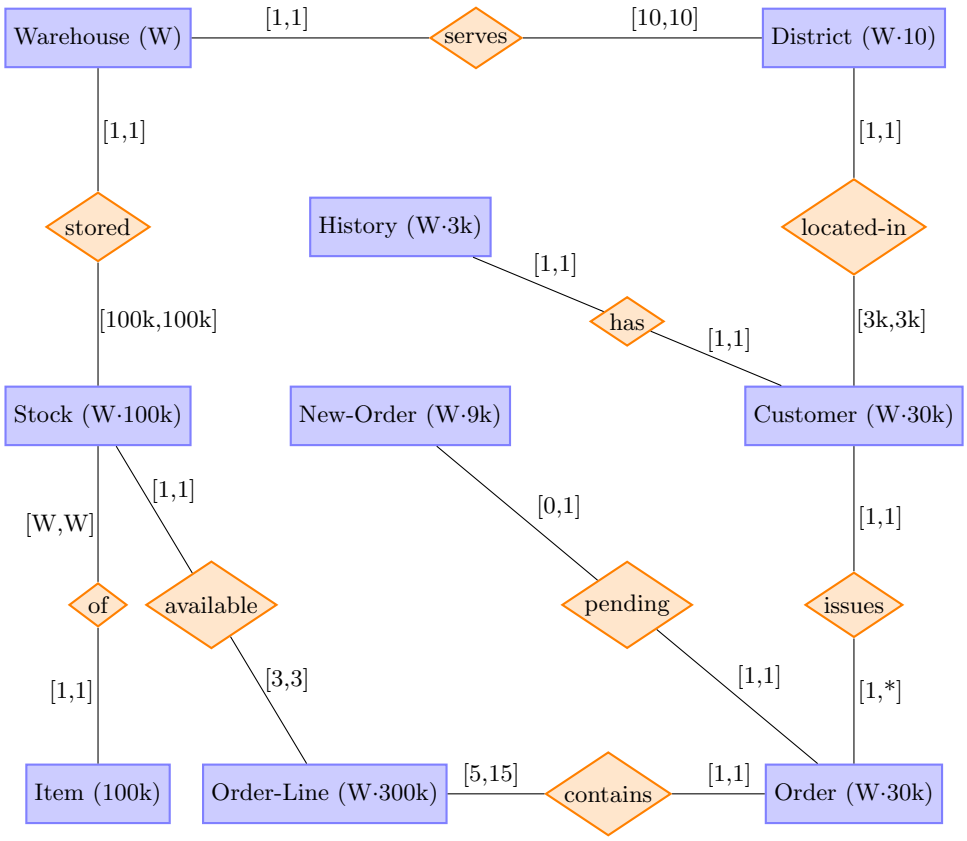

Fig. 4. Entity-relationship-diagram of the TPC-C benchmark based on [6]

Stoica et al. used the TPC-C benchmark to measure throughput and latency of their extended VoltDB. As the number of cores is 16, the scaling factor is set to the same value. In this setting, the database can be divided into one partition for each warehouse, each partition having one worker thread that handles all transactions that access this warehouse. However, Stoica et al. decided to modify the benchmark a little bit: the deletion of the "Delivery Transactions" is omitted, since they intended to have as much growing tables as possible.

Throughput The throughput is measured in three different settings:

1. In-memory: The native VoltDB is run using the whole RAM (64GB, thereof ∼62GB usable)
2. Default Paging: Still the unmodified version of VoltDB is used, but the RAM is reduced to a size of 5GB (∼3GB usable), so in case the database size exceeds the RAM's capacity, data is paged out to the SSD.

3. Data reorganization: The memory configuration is the same as in the second setting, but now the modified version of VoltDB is used.

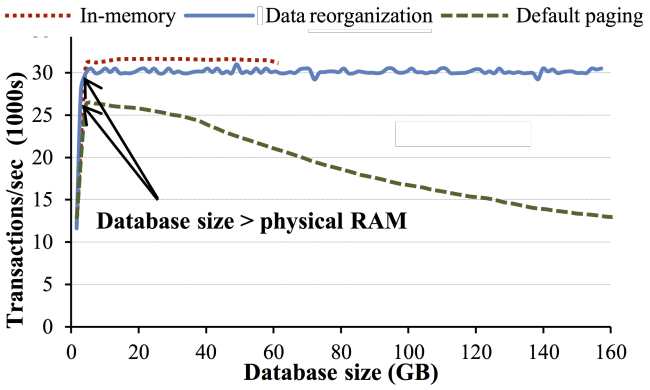

Fig. 5. TPC-C throughput [8]

Figure 5 illustrates, that in the third setting only ca.7% of the throughput is lost compared to the first setting. Furthermore, the throughput remains stable even when the database size grows to 160GB which is more than 50 times the size of the available RAM. In contrast, at a database size of 160GB the throughput in the reference setting (2) falls down to less than half the throughput of the modified VoltDB.

Latency Figure 6 shows the result of the latency measurements, that have been determined in a similar environment as the throughput measurements. However, the throughput has been decreased to 50% of the maximal throughput in order to avoid that waiting times falsify the results, since the only thing that shall be measured is the overhead of the paging process.

Stoica et al. measured the latencies both at the client side and the server side and calculated the average latency and the standard deviation for each of the 5 transactions.

More precisely, the "server latency" measures the time elapsed when a transaction is started directly on the database server until it is completed. In contrast, the "client latency" indicates the latency in a client-server scenario, where the transaction is initialized at the client, i.e., the time span between begin until its end as percieved by the client is measured.

The results show that paging increases the average latency by up to $\sim 101 \mu s$, the standard deviation by $\sim 60 \mu s$ in average. But from the client side, the increasing latencies are relatively small compared to the network latencies and the additional I/O overhead and the server side transaction manangement. So

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

Fig. 6. TPC-C transaction latency [8]

paging does not add more than 2.27% (in case of the "Stock-Level Transaction") to the average latency under the bottom line.

4.3 The mircro-benchmark and its results

The TPC-C benchmark does not really focus on the workloads, where there are hot and cold records that evolve over time. For that reason, Stoica et al. decided to initiate their own simple benchmark. They generated a database of size 26.2 GB that consists of 10^8 200B tuples, which are identified by 4B integer primary keys. Then a simple select query is executed, and, given a primary key, returns the corresponding tuple including all its values. The distribution of the selected primary keys follows a Zipfian distribution which means that the probability p of the selection of a tuple is inversely proportional to the position of its key in the list of all primary keys ordered by their number of occurences. The scaling is implemented such that 80% of the selected primary keys access 20% of the data.

As Stoica et al. try to measure the adaptivity of their system, they do a workload shift, that means, at a given point of time, they select at random a new set of hot tuples and measure how this affects the throughput.

The evaluation then takes place in the same three settings as the TPC-C benchmark, but in the second and the third setting, the database size is five times larger than the memory size, which is increased such that 5GB are usable.

Figure 7 shows how the three different systems react to such a workload change. The native VoltDB that holds all data in memory still performs best and is not affected, its throughput remains constant. In contrast, the VoltDB of the second setting suffers from the high paging operation and "is unable to execute any queries" [8].

The behavior of the modified VoltDB can be divided into five phases. In the initial phase ①, all data structures are adapted to the current set of hot tuples and ca. 65,000 queries/sec can be executed. When the workload change happens, the throughput falls down by 94% to about 4,500 queries/sec because most of the new hot tuples are stored on SSD (phase ②). In phase ③, the OS paging

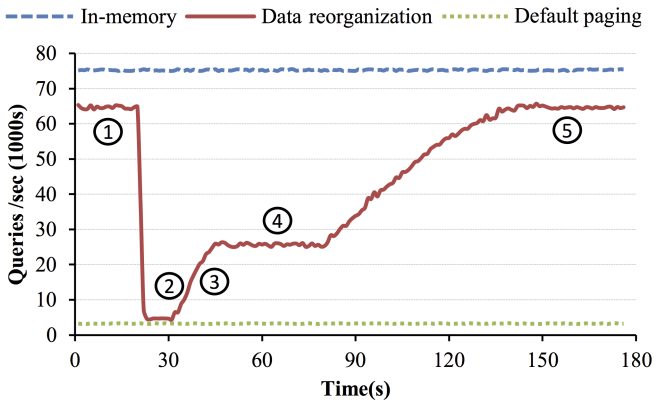

Fig. 7. Throughput for a shifting Zipfian working set[8]

starts buffering the most hot pages from SSD in memory. As a consequence, the throughput increases to ∼25,000 queries/sec and remains on this level (phase ④) until, after preconfigured 60 seconds, the data reorganization process starts. Within about one minute, the throughput becomes stable again at the initial throughput level of 65,000 queries/second (phase ⑤).

5 Conclusion

In the previous sections, we have discussed a technique for OLTP databases that make use of the virtual paging mechanism for the purpose of storing cold records on secondary storage. Its key aspects are that the reorganization process is separated as much as possible from the query execution process, that hot tuples are stored densely and in adjacent sections of the memory and finally, that designated, important memory regions and also hot memory regions are not paged out to secondary storage [8].

On the one side, an advantage of Stoica et al.'s approach is that it is portable since it does neither modify the physical data structures nor the database engine's concurrency mechanism [8]. Aside from that, the benchmark results (section 4) show that, in a given OLTP scenario, the standard paging mechanism is outperformed as soon as the database size exceeds the memory's capacity. In case of the TPC-C benchmark, the database could even grow to a size of about a factor of fifty of the memory size without any loss of performance, which is quite remarkable.

On the other side, one might criticize that this approach is too OLTP centered – if the records cannot be divided into hot and cold, e.g., if the access frequencies are uniformly distributed, the technique will not work anymore. Instead, the reorganization technique would increase the number of I/O operations in addition to the already performed ones due to the OS paging mechanism performed ones. Furthermore, the backward algorithm would have to scan the whole log, since the variance of the access frequencies is much more limited.

Page faults in general are a problem of Stoica et al.'s approach. As in most in-memory database engines, data access is serialized, a page fault delays all other transactions in the queue. In VoltDB, the different partitions help to reduce this problem, nevertheless, in the worst case, a transaction that accesses tuples on all partitions, still stalls all other transactions, when it faces a page fault.

Another major drawback is the use of access frequencies in order to determine which tuples are hot and which are not. Let us consider the following scenario: in a binary search tree, one leaf is requested very frequently, whereas all the other nodes are not. To find this node, all nodes from the root to this leaf are accessed and are thus approximately as frequently accessed as the leaf itself. As a consequence, the system considers the pages of the whole access path to be "hot", too, and the whole path is stored in memory, though only one tuple is hot.

For the purpose of evaluating how efficient Stoica et al.'s implementation is, some measurements of the energy consumption in order to compare the three settings (in-memory, default OS paging, data reorganization) would have been interesting, since this factor has been one of the reasons in favor of the use of secondary storage.

Moreover, all the measurements have only been performed on a system based on SSDs. Though it is likely, that the same technique could also be applied on a system using HDDs as secondary storage, where the perfomance gains compared to native virtual paging would probably be even higher, some reliable

experiments and a comparison between these two storage devices would have brought more insights, e.g., if it is more advisable to use HDD or SSD when the database size is in a range of terabytes or petabytes.

So to sum up, the idea of using the data reorganization technique proposed in [8] can enhance the performance and the capacity of in-memory databases to a large extent, under the premise, that the workload equals to the one assumed in [8] and [7].

6 Prospects

As already mentioned in the conclusion, Stoica et al.'s approach concentrates only on OLTP workloads, where hot and cold records can be separated. But what about OLAP (online analytical processing) workloads? In contrast to OLTP workloads, they are characterized by complex queries such as aggregations and joins in order to analyze large amounts of data. In consequence, a separation into hot and cold data cannot be done so easily.

One possible solution would be to store only the query results in memory and move tuples, that have been processed by all OLAP transactions, to secondary storage. The results can then be updated incrementally. However, this approach is hardly feasible by use of the virtual memory paging mechanism and it can only be applied, if the OLAP queries are static and known, since loading the records from secondary storage into memory would impact the system's performance to a large extent.

Briefly, a good solution to apply the idea of secondary storage to OLAP workloads has not been found yet, thus, more research has to be done in this domain.

In [3], another problem of Stoica et al.'s approach is adressed: page faults.

In this paper, DeBrabant et al. developed a system that integrated a so called "Anti-Cache" into the main-memory database H-Store [9]. Like Stoica et al. the database system runs in memory and cold tuples are migrated to secondary storage, but in contrast to [8] without the use of virtual memory paging.

To achieve a replacement strategy at record level, each partition of the database is assigned a so called "anti-cache storage manager". Its task is to maintain a "Block Table", an "Evicted Table" and an "LRU Chain" of records for each relation.

The "Block Table" contains the serialized tuples that have been removed from memory. All blocks have the same size and are identified by a 4-byte key. Furthermore, the name of the relation that contains the tuples and the creation time are stored in the block. The metadata is stored in memory, the serialized tuples with their size as prefix on disk.

The "Evicted Table" contains block id and tuple id of evicted tuples. The tuple id is computed by its offset inside the block it is stored in. So as soon as a tuple is ejected from memory to disk, the system creates dynamically a new block which contains this tuple and appends it to the "Block Table". The new block id, as well as the tuple's offset are then also added to the "Evicted Table".

In contrast to [8] and [7], De Brabant et al. do not make use of an external component in order to determine hot and cold tuples – H-Store keeps an "LRU Chain" for each table, i.e., a tuple that has been accessed, is removed from the chain and added at its tail. So whenever eviction has to be performed, the first tuples of the LRU chain will be moved to disk first. For performance reasons, the tuple accesses are sampled and such an "LRU Chain" is only maintained, if the table is marked as "evictable". In this way, tuples of relations that are known to be entirely hot, are never transferred to disk.

A transaction is then executed as follows: First the selected keys are examined. If all selected records are in memory, the transaction is executed immediately. Otherwise, it is aborted and enters a so-called "pre-pass phase" during which the required evicted tuples are determined by the system and all write operations, that have been performed, are rolled back. The transaction is restarted as soon as all evicted records are reloaded into memory. The advantage of this approach is that the following transactions do not have to wait. Thus, a higher throughput can be achieved. The disadvantage is that, for a table scan, the whole database has to be loaded into memory, which is not always possible.

References

[1] VoltDB In-Memory Database, May 2014. http://www.voltdb.com.

[2] Transaction Processing Performance Council. TPC-C Standard Specification, May 2014. http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[3] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.*, 6(14):1942–1953, September 2013.

[4] Florian Funke, Alfons Kemper, and Thomas Neumann 0001. Compacting Transactional Data in Hybrid OLTP & OLAP Databases. *PVLDB*, 5(11):1424–1435, 2012.

[5] Jim Gray and Franco Putzolu. The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time. *SIGMOD Rec.*, 16(3):395–398, December 1987.

[6] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 7. Auflage*. Oldenbourg, 2009.

[7] Justin J. Levandoski, Per-Ake Larson, and Radu Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, pages 26–37, 2013.

[8] Radu Stoica and Anastasia Ailamaki. Enabling Efficient OS Paging for Main-memory OLTP Databases. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, pages 7:1–7:7, New York, NY, USA, 2013. ACM.

[9] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.