How to Efficiently Snapshot Transactional Data: Hardware or Software Controlled?

Henrik Mühe
TU München
Boltzmannstr. 3
85748 Garching, Germany
muehe@in.tum.de

Alfons Kemper
TU München
Boltzmannstr. 3
85748 Garching, Germany
kemper@in.tum.de

Thomas Neumann
TU München
Boltzmannstr. 3
85748 Garching, Germany
neumann@in.tum.de

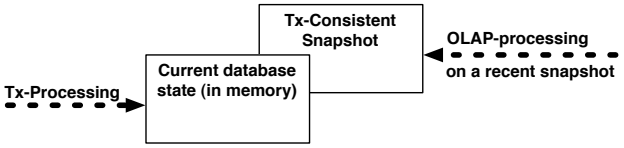
ABSTRACT

The quest for real-time business intelligence requires executing mixed transaction and query processing workloads on the same current database state. However, as Harizopoulos et al. [6] showed for transactional processing, co-execution using classical concurrency control techniques will not yield the necessary performance – even in re-emerging main memory database systems. Therefore, we designed an in-memory database system that separates transaction processing from OLAP query processing via periodically refreshed snapshots. Thus, OLAP queries can be executed without any synchronization and OLTP transaction processing follows the lock-free, mostly serial processing paradigm of H-Store [8]. In this paper, we analyze different snapshot mechanisms: Hardware-supported Page Shadowing, which lazily copies memory pages when changed by transactions, software controlled Tuple Shadowing, which generates a new version when a tuple is modified, software controlled Twin Tuple, which constantly maintains two versions of each tuple and HotCold Shadowing, which effectively combines Tuple Shadowing and hardware-supported Page Shadowing by clustering update-intensive objects. We evaluate their performance based on the mixed workload CH-BenCHmark which combines the TPC-C and the TPC-H benchmarks on the same database schema and state.

1. INTRODUCTION

Harizopoulos et al. [6] investigated the performance bottlenecks of traditional database management systems and found out that 31% of the time is spent on synchronization and 35% on page and buffer management. Consequently, a new generation of main memory DBMS has been engineered to remove these bottlenecks. Overhead caused by page and buffer management can be removed in in-memory DBMS by relying entirely on virtual memory management instead of devising costly software controlled mechanisms. One of the most prominent examples of this generation of in-memory database systems is VoltDB [15], a commercial product based on the H-Store research prototype [8]. There, costly mechanisms like locking and latching for concurrency control are avoided by executing all transactions sequentially, therefore yielding serializability without overhead. In order to increase the level of parallelism, the database can be logically partitioned to accommodate one transaction per partition [3].

Lockless, sequential execution has proven to be very efficient for OLTP workloads, specifically short transactions that modify only a handful of tuples and terminate within microseconds. With the need for "real-time business intelligence" as advocated by Plattner of SAP [13] among others, serial execution is bound to fail. Long-running OLAP queries cannot be executed sequentially with short OLTP transactions since transaction throughput would be severely diminished every time a long-running query stops OLTP transactions from being executed. To solve this dilemma, we advocate the use of consistent snapshots for the execution of long-running OLAP queries which yields both: High OLTP throughput by executing OLTP transactions sequentially as well as a way of executing OLAP queries on a fresh, transaction consistent state of the database, as sketched:



This mechanism has previously been demonstrated in our research prototype, HyPer [9]. For this approach to work, being able to efficiently maintain and create a consistent snapshot of the database at short intervals of seconds is paramount. In this work, we will examine different techniques for maintaining a consistent snapshot of the database without diminishing OLTP performance. We go beyond what has been done in previous work, for instance by Lorie [11] or Cao et al. [1]. First, we extend the mechanisms to enable high performance query execution on snapshots as their most important use – instead of just recovery as previously suggested, e.g., by Molina et al. [5] or in [14]. Therefore, our approaches yield higher order snapshots for query processing as opposed to those snapshots primarily used for recovery. Second, we adapt the mechanisms for use in main memory database systems: In case of Lorie's shadowing approach, we show how the limitations when used in on-disk database systems can completely be alleviated in main memory. With Cao et al.'s twin objects approach (called ZigZag approach in [1]), we extended the implementation for use in a general purpose database system instead

Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN 2011), June 13, 2011, Athens, Greece.

of a specialized application. Third, we offer a thorough evaluation of the different approaches taking both OLTP as well as OLAP throughput into account.

The remainder of this paper is structured as follows: In Section 2, we introduce the concept of virtual memory snapshots. In Section 3, we will reexamine different techniques for maintaining a consistent snapshot of the database and discuss our implementations and the improvements we added to make them viable for OLAP query execution. Section 4 offers a classification of these snapshotting techniques. In Section 5, all techniques are evaluated using a combination of the TPC-C and TPC-H benchmarks called the CH-BenCHmark [4]. Section 6 concludes this paper.

2. HARDWARE SNAPSHOTTING

In this section, we will focus on hardware supported virtual memory snapshotting as proposed in [9]. Purely software-controlled as well as hybrid approaches will be described in the next section.

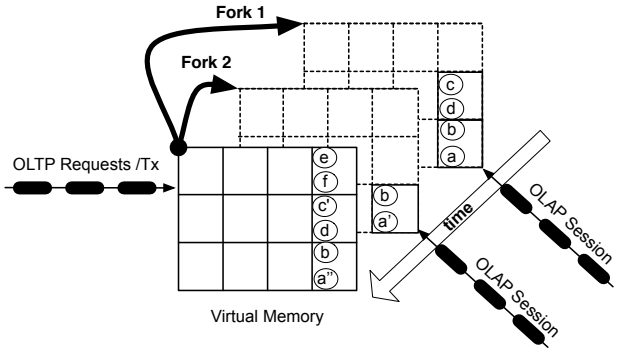


Figure 1: Hardware Page Shadowing with multiple snapshots taken at different transaction consistent states of the database

Hardware Page Shadowing is a new snapshotting technique that we developed in our HyPer main memory database system. It creates virtual memory snapshots by cloning (forking) the process that owns the database. In essence, the virtual memory snapshot mechanism constitutes an OS/hardware supported shadow paging mechanism as proposed by Lorie [11] decades ago for disk based database systems. However, the original proposal incurred severe costs as it had to be software-controlled and it destroyed page clustering on disk. Additionally, virtual memory is not replaced by Lorie's approach, but instead an additional layer of indirection is added, further decreasing performance. None of these drawbacks occurs in virtual memory snapshotting as clustering across RAM pages is not an issue – as we examined in microbenchmarks. Furthermore, the sharing of pages (between OLTP and OLAP snapshots) and the necessary *copy-on-update* is managed by the operating system, efficiently supported by the MMU hardware (memory management unit). This way, translations between virtual and physical addresses via the page table as well as page replication (*copy-on-update*) do not need to be implemented in the database management system. Replicating a page is highly efficient, taking only 2μs for a 4kb page as we measured in a microbenchmark on a standard processor.

Virtual memory snapshots exploit the OS facilities for memory management to create low-overhead memory snapshots. All modern operating systems and hardware support and widely use virtual memory management. This means that physical memory is not directly assigned to a process that requests memory but is mapped through a layer of indirection called virtual memory. All memory accesses use virtual addresses and do not need to know which physical memory pages back a given virtual address. Translations from virtual to physical addresses are done in hardware by providing a lookup table to the memory management unit of the CPU as per the specifications of the processor vendor.

In unix environments, new processes are created by cloning an existing process using the `fork` system call. Since an identical copy of an existing process has to be created, all memory used by this process has to be copied as well. With virtual memory, an eager copy of the memory pages used by the process issuing the fork system call (parent process) is not required, only the table used to translate virtual to physical addresses used by the parent process needs to be copied. Therefore, the physical pages backing both the parent's as well as the child's virtual address space are shared at first. All shared physical pages are marked as read-only during the execution of the fork system call. When a read-only page is modified by either process, a CPU trap is generated causing the operating system to copy the page and change the virtual memory mapping to point to the copy before any modifications are applied. Effectively, this implements a hardware controlled copy-on-write mechanism.

Applied to main memory database systems, we use the fork to generate a lazy copy of the database system's memory with little delay. In order for this snapshot to be consistent, we execute the fork system call in between two serial transactions. This is not strictly necessary though, since undo information is also part of the memory copy. Therefore it is possible to quiesce transaction execution without waiting for transactions to end, execute the fork system call and clean the action-consistent snapshot using the undo log.

The forked child process obtains an exact copy of the parent processes' address space, as exemplified in Figure 1 by the overlaid page frame panel. This virtual memory snapshot will be used for executing a session of OLAP queries – as indicated on the right hand side of Figure 1.

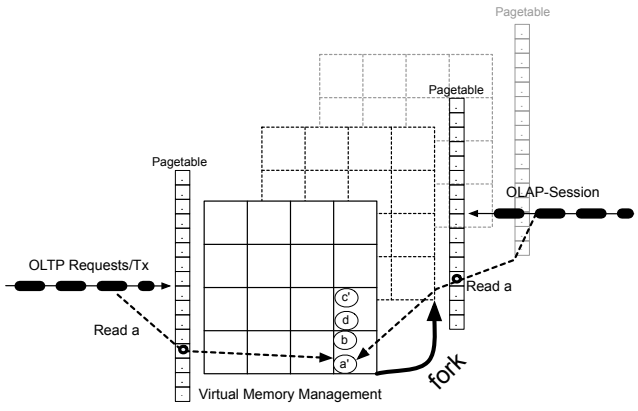


Figure 2: The page table after invoking the `fork` system call.

The snapshot stays in precisely the state that existed at the time the `fork` took place. Fortunately, state-of-the-art operating systems do not physically copy the memory segments right away. Rather, they employ a lazy *copy-on-update* strategy – as sketched out in Figure 2. Initially, parent process (OLTP) and child process (OLAP) share the same physical memory segments by translating either virtual addresses (e.g., for object a) to the same physical main memory location. The sharing of the memory segments is highlighted in the graphics by the dotted frames. A dotted frame represents a virtual memory page that was not (yet) replicated. Only when an object, like data item a', is updated, the OS- and hardware-supported copy-on-update mechanism initiates the replication of the virtual memory page on which a' resides as is illustrated in Figure 3. Thereafter, there is a new state denoted a'' accessible by the OLTP-process that executes the transactions and the old state denoted a', that is accessible by an OLAP query session. As shown in Figure 1, multiple snapshots representing different consistent states of the database can be maintained with low overhead. Here, an older snapshot is shown which was taken before data item a was modified to a'. The page on which data item a lies is a copy denoted by the solid border of the page, most other pages are shared between all snapshots.

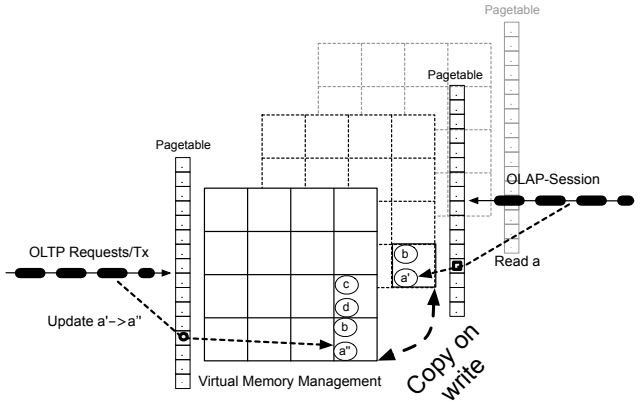


Figure 3: The page table after a page was modified, causing a page copy on update.

Unlike the figure suggests, the additional page is really created for the OLTP process that initiated the page change and the OLAP snapshot refers to the old page – this detail is important for estimating the space consumption if several such snapshots are created. It can be expected that most pages and objects on those pages are of the nature of the page inhabited by e and f. That is, they contain older, no longer mutated objects.

3. SNAPSHOTTING IMPLEMENTATIONS

We implemented a total of four snapshot mechanisms, each anchored deeply within the storage scheme of our main memory database system prototype. All storage backends integrate a mechanism which maintains a consistent snapshot that can be periodically refreshed within short intervals of seconds to minutes.

3.1 Hardware Page Shadowing (VM-Fork)

In Figure 4, the backend implementation of the virtual memory snapshotting approach is illustrated on a more detailed level. Here, a relation spanning multiple pages as well as SQL statements executed on this relation are shown. Statement A) deletes a tuple which is deleted in-place in the memory accessed by the OLTP process. Since the memory used by OLAP queries is lazily shadow copied, the deletion causes an actual copy of the modified page, denoted by the gray memory used by OLAP queries in the background of the figure. Statement B) modifies a tuple, again causing the page that has previously been shared between OLTP transactions and OLAP queries to be copied. Insertions – as exemplified by statement C) – are performed at the end of the vector which – contrary to the depiction in Figure 4 – is not maintained in any specific order to allow for high OLTP throughput without any added level of indirection for newly inserted tuples. The architecture sketched here has been successfully implemented in our main memory database research prototype HyPer [9].

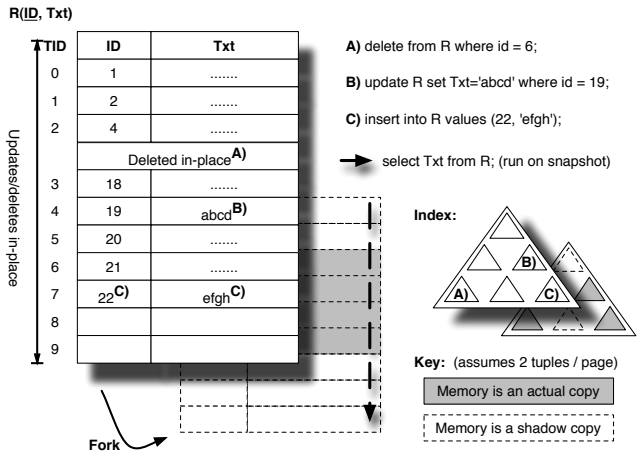


Figure 4: Query execution when using hardware page shadowing

3.2 Tuple Shadowing

Instead of shadowing on page level, shadow copies can be created on tupel level, thus possibly lowering the memory overhead of keeping a consistent snapshot. To manage per tuple shadow copies, we have to resort to software control. Thus, a more complex indirection has to be established since for each access, the current version of the data being used has to be determined on a per-tuple level. No hardware mechanism can be specifically exploited to speed-up tuple-shadowing, thus all indirection has to be dealt with in software.

A straight forward implementation of tuple shadowing is shown in Figure 5. There, a consistent snapshot was taken after the insertion of the tuple with TID 7. When a tuple with TID lower or equal to 7 is modified, modifications are not done in-place but rather by copying the tuple to the end of the relation and establishing a link between the original and the shadow copy. Statement B) is an example of such an update. Instead of changing the tuple with TID 5, the tuple is copied to the end of the relation and modified there. The original tuple is annotated with the TID of its shadow copy

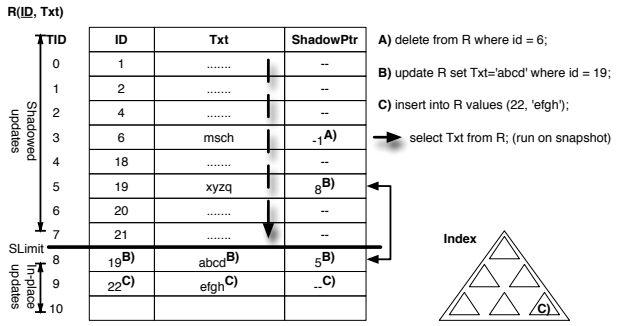


Figure 5: Implementation of the Tuple Shadowing approach.

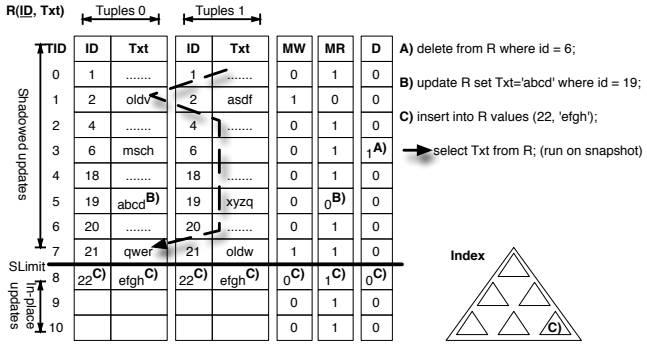


Figure 6: Implementation of the Twin Tuples approach.

to allow merging the two different version of the tuple at a later time and to allow different readers to access the two different versions. Since no index update is performed, all accesses to both the new and the old version of the tuple go through the original index entry, adding indirection. Deletions of tuples – as exemplified by statement A) – are not executed in place but rather accomplished by flagging the tuple in question as deleted. Insertions – like statement C) – are performed at the end of the relation without maintaining a specific order.

All OLTP accesses have to be directed to the most recent version of the tuple. Therefore, these accesses have to consult the pointer saved in the *ShadowPtr* field to check whether or not a more recent version of the data exists. OLAP queries work on the consistent snapshot and thus do not need to check the *ShadowPtr* for a more recent version. Additionally, checking whether or not a tuple has been flagged as deleted is not necessary for snapshot accesses, since deletion flags only apply to the most recent version of all tuples used by OLTP queries. Refreshing the snapshot incurs substantial copy and merge costs.

3.3 Twin Tuples

To mitigate the overhead caused by periodically merging the database as is necessary in Tuple Shadowing, a technique referred to as the Twin Block approach or – when done on a per tuple level – Twin Tuples approach can be employed [1].

In the Twin Tuples approach, two copies of every data item exist as is illustrated in Figure 6. Two bitmaps indicate which version of a tuple is valid for reads and writes by OLTP transactions. Reads are performed on the tuple denoted by the *MR* bit, the tuple that writes are performed on is denoted by the *MW* bit. A consistent snapshot of the data can be accessed by always reading the tuples that are not modified as indicated by the negation of the *MW* bit.

Figure 6 shows that deletions like statement A) are performed by flagging the tuple in question as deleted. Insertions like statement C) are again performed at the end of the storage vector without maintaining a specific order. Updates on the other hand are now performed on one of the two stored tuples. When the first update on a tuple is performed, the *MR* flag is set to the value of the *MW* flag and the update is performed on the tuple denoted by the *MW* flag. Since *MW* is atomically toggled only when the consistent snapshot is being refreshed, for the duration of a snapshot the tuple denoted by $\neg MW$ is never written to and thus stays in a consistent state.

This behavior is exemplified by statement B) updating the tuple with TID 5. Here, the original tuple is read from the tuple denoted by *MR*, an update is performed and the result is written to the twin location denoted by *MW*. Afterwards, *MR* is set to *MW*.

OLAP queries read the tuples that have not been written, that is, tuples in twin location $\neg MW$. Because the value of the *MW* flag can vary for different tuples, scans are performed in a zigzag pattern on the data array, potentially causing diminished scan performance.

3.4 HotCold approach

With hardware page shadowing, an update to a single value on a page causes the entire page to be copied. The HotCold approach is intended to cluster update-intensive tuples to the so called hot section in memory. Updates which would modify a tuple which is not in the hot section are copied to that section and marked as deleted in the cold section. That way, modifications only takes place in the hot part. The technique is a combination of Tuple Shadowing and Hardware Page Shadowing as the update clustering is software controlled whereas shadow copying is done using the VM-Fork mechanism.

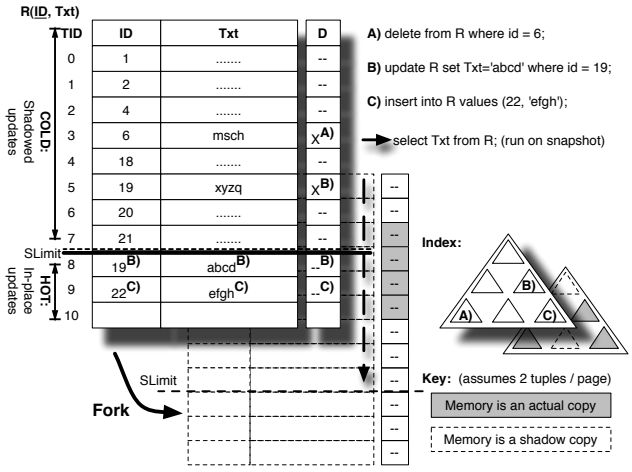


Figure 7: Implementation of the HotCold approach.

In Figure 7, statement B) modifies a tuple with a TID before the current *SLimit*, which makes the tuple a cold

tuple. Instead of modifying the tuple in place, a deletion flag is set and the tuple is copied to a slot inside the hot part of the store. There, modifications are applied in-place.

Statement C) in Figure 7 inserts tuples at the end of the vector equally to the way tuples are inserted in Tuple Shadowing. This – on average – does not cause a page to be copied since the vector is lazily backed with physical memory pages and areas that have not been backed yet are not copied via the hardware page shadowing mechanism. Deletions in the cold part of the store are performed by flagging a tuple as deleted as can be seen with statement A).

3.5 Index structure synchronization

For approaches where index structures are used both for OLTP transactions as well as for OLAP queries, we have to take index synchronization into account. When index structures are shared, updates to the index can conflict with lookups. We examined four approaches to alleviate this problem:

1. Abandon indexes for OLAP queries or creating OLAP indexes on demand.
2. Eagerly copying indexes when a snapshot is created.
3. Employing hardware page shadowing to lazily maintain index snapshots after database snapshot creation.
4. Latching indexes to synchronize conflicting index operations.

Approach 1) is interesting when all OLAP queries rely entirely on table scans with no particular order. Since none of our implementations guarantees any specific order on the data, we assume that having indexes available on the snapshots is oftentimes required and thus do not further investigate this option. Additionally, 1) does not require any specific implementation or synchronization considerations.

Approach 2) generates a separate copy of the indexes for each snapshot by eagerly duplicating the index when a snapshot is created Therefore, no synchronization is necessary as no indexes are shared but reorganization speed decreases. Hardware page shadowing for indexes as done in approach 3) achieves the same result but does not create a complete copy of the indexes. Rather, it copies only the page table of the pages used to store index data and applies the *copy-on-update* mechanism as introduced in Section 3.1. In the last approach, 4), index data between OLTP transactions and OLAP queries is shared making synchronization necessary.

For our Hardware Page Shadowing approach as well as for the HotCold approach, sharing an index structure between OLTP transactions and OLAP queries is implicit with the process cloning/forking. Thus, index latching and shared usage of the index is not applicable with these approaches. Approaches 1), 2) and 3) are applicable.

4. CLASSIFICATION

The following section contains a classification of the different snapshotting techniques examined in this paper.

4.1 Snapshotting method

The techniques discussed in this work can be subdivided by the method they use to achieve a consistent snapshot while still allowing high throughput OLAP transactions on the data. The HotCold approach as well as the plain hardware page shadowing approach use a hardware supported copy on write mechanism to create a snapshot. In contrast to that, tuple shadowing as well as the twin object approach use software mechanisms to keep a consistent snapshot of the data intact while modifications are stored separately. This is also displayed in Figure 8 where all techniques are classified by whether snapshot maintenance is done in software, in hardware or both:

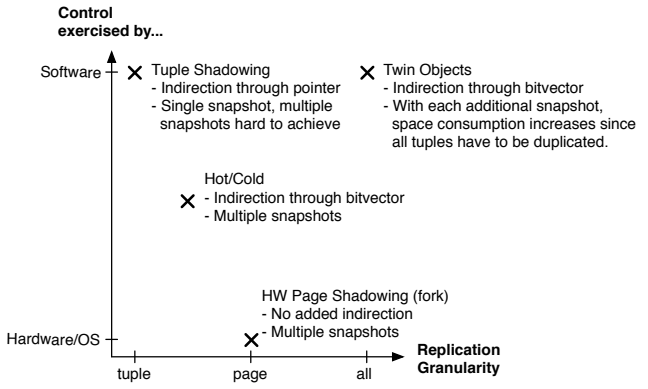


Figure 8: Techniques classified by granularity and control mechanism.

The snapshotting mechanism has a direct impact on the amount of reorganization required when the snapshot needs to be refreshed. The hardware supported page shadowing approach requires no reorganization whatsoever, only the OLTP process needs to be quiesced and the `fork` system call needs to be executed. Since the HotCold approach relies on the same mechanism to generate a consistent snapshot, no reorganization is required either but an optional reorganization can be performed. This saves memory by actually removing tuples that have been flagged as deleted and it also increases scan performance as a deletion flag checks become unnecessary during scans.

With software based snapshotting approaches, reorganization is mandatory on snapshot refresh. First, tuples flagged as deleted need to be actually removed or at least marked as unused so that they can be overwritten with new tuples at a later point. In case of tuple shadowing, updates saved in shadow copies have to be written back to the original version to prevent a long chain of versions from forming which would linearly increase the level of indirection when accessing tuples. Whereas the approaches based on hardware page shadowing only required quiescing the OLTP process, software based snapshotting techniques usually require the entire database to quiesced incurring

To conclude, approaches maintaining a snapshot using software mechanisms require a reorganization phase to refresh that snapshot whereas approaches relying on hardware page shadowing need no reorganization or at most an optional reorganization phase.

4.2 Indirection

With hardware page shadowing, all indirection required is handled by the operating system's virtual memory mechanisms. Since virtual memory is used by all approaches since direct allocation of physical memory is neither useful nor technically simple, no additional level of indirection is added by hardware supported page shadowing.

Software based approaches introduce a level of indirection: Tuple shadowing keeps a pointer to updated shadow copies

of each tuple forcing OLTP queries to check whether a newer version exists or not. The twin object approach requires a bitmap to be checked on each read access and thereby introduces indirection on tuple access.

4.3 Memory overhead and granularity

As the name suggests, hardware supported page shadowing uses a page as its smallest granularity level causing an entire page to be copied on modification. This is also displayed in Figure 8 where all techniques are classified by the granularity in which memory consumption grows due to modification.

Since in hardware page shadowing all pages end up being replicated in a worst case scenario, the memory used for OLTP transaction processing is at most doubled to maintain one consistent snapshot for OLAP processing. Because of the page level granularity, not all tuples need to be modified to cause worst-case memory consumption: If at least one bit is modified on each page, all pages will end up being copied.

Compared to pure hardware page shadowing, the HotCold approach lowers the rate at which memory consumption increases. This is done by clustering updates in a designated part of the memory called the hot area. In a worst case scenario, memory consumption still doubles to maintain a consistent snapshot, but every tuple has to be modified to cause worst case behavior. Thus, the HotCold approach effectively decreases the speed at which memory is consumed by OLTP transactions.

Tuple shadowing as well as twin objects work with a per-tuple granularity. Tuple shadowing copies a tuple on modification thus increasing memory consumption linearly with the number of modified tuples. Twin objects saves two versions of each tuple by default, thus exhibiting worst case memory consumption right from the start – the approach is mainly used to illustrate the varying degrees of overhead introduced by reorganization.

4.4 Concurrency in indexes

A low cost snapshot of the database does not necessarily allow for high performance query execution. One of the reasons is that metadata like indexes are missing. In section 3.5, we introduced four ways of dealing with indexes which we will now revisit for a classification.

4.4.1 Abandoning indexes

The trivial solution of abandoning all index structures consumes no additional memory and at the same time offers no help when accessing data form OLAP queries. Required indexes can be regenerated online incurring a runtime performance overhead during query execution.

4.4.2 Eager index copy

Indexes can be duplicated eagerly when the snapshot is taken. This results in an increase in memory consumption but retains indexes for use in OLAP queries. Since OLTP as well as OLAP queries need to be quiesced for index copies to be generated in a consistent fashion, the additional time spend while refreshing a snapshot decreases OLTP as well as OLAP throughput. At transaction and query runtime, no overhead is incurred.

4.4.3 Index fork

Equivalently to data, indexes can be copied using the hardware snapshotting technique discussed in section 3.1. In a worst case scenario, memory consumed by indexes duplicates over time as index entries are updated. When the snapshot is created, the only delay incurred is the duration of the fork system call which is short compared to eagerly copying the entire index (see Figure 5.1). At runtime, pages which are modified for the first time have to be copied. This is done by the OS/MMU and takes only about 2 microseconds for a 4 kilobyte page [9].

4.4.4 Index synchronization

For techniques where index sharing is possible, namely Tuple Shadowing and Twin Objects, inconsistencies due to concurrent access have to be prevented. This can be done by latching index structures so that writers get exclusive access to an index whereas multiple readers can access it concurrently. In this case, indexes do not have to be duplicated but the latches incur a comparably small memory overhead. Minimally, every index access has to pass at least one latch. Thus the runtime overhead for this approach consists of the time it takes to acquire a latch as well as possible wait-time in case the latch is held by another process.

4.5 Classification summary

Backend	Snapshot mechanism	Indir.	Gran.	Index sharing
Fork	hw	VM only	page	n/a
Tuple	sw	VM + ptr	tuple	yes
Twin	sw	VM + bit	all	yes
HotCold	hw/sw	VM + bit	tuple	n/a

Figure 9: Classification overview between all presented techniques and index synchronization mechanisms.

5. EVALUATION

In this section, all proposed techniques for the hybrid execution of OLTP transactions and read only OLAP queries will be thoroughly evaluated.

5.1 Snapshotting performance

For all techniques, OLTP processing has to be quiesced when the snapshot is refreshed. Since this directly impacts OLTP transaction throughput, we measured the total time it takes before OLTP processing can be restarted. For techniques employing hardware page shadowing, the time required to finish the `fork` system call is measured. For software based approaches, the time required for memory reorganization is measured. When reorganization is optional, the time required for the optional part of the reorganization is given in braces.

Backend	4kb pages	2mb pages
VM-Fork	47ms	13ms
Tuple	500ms	483ms
Twin	94ms	85ms
HotCold	50ms	13ms
	(2829ms)	(2097ms)

Table 1: Reorganization time by backend, optional reorganization runtime given in brackets.

Table 1 shows the time required to refresh a snapshot for the different techniques. Reorganization took place after loading the data of the TPC-C benchmark scaled to 5 warehouses and then running the TPC-C transaction mix until a total of 100,000 transactions (roughly 44,000 neworder transactions) were finished. A snapshot of the database containing the data that was initially loaded was maintained while executing the transactions.

5.2 Raw scan performance

In addition to the tests conducted during the execution of the CH-BenCHmark, we measured scan performance in a microbenchmark setting. First, we evaluated the time it takes to determine which tuples inside the store are valid, that is, time to find all valid TIDs. Second, we evaluated the predicates *min(4b)* and *min(50b)* which determines the lowest value for a 4 byte integer and for a 50 byte string, respectively.

Backend	Valid Tuples	4kb pages		2mb pages	
		Min(4b)	Min(50b)	Min(4b)	Min(50b)
VM-Fork	72ms	188ms	702ms	186ms	701ms
Tuple	72ms	216ms	715ms	212ms	708ms
Twin	74ms	242ms	813ms	250ms	796ms
HotCold	146ms	199ms	769ms	197ms	767ms

Table 2: Scan performance on snapshot after removing 1% and updating 2% of the tuples.

The two queries were run on a snapshot taken after 30 million tuples were loaded into the table being tested. Before running the queries, OLTP transactions changing a total of 2% of the tuples inside the table and deleting another 1% were run.

The time it takes to determine all valid TIDs is given as the 'Valid Tuples' value. It is a baseline for table scan execution speed. 'Valid Tuples' performance is inferior on the HotCold store. This stems from the fact that reorganization of that store is optional and a snapshot can therefore contain tuples which have been marked as deleted. If reorganization was changed to be mandatory in the HotCold approach, checking for deleted tuples would no longer be necessary and the runtime would be in the same ballpark as it is on the other stores.

When comparing the relative difference in speed of execution between the *min(4b)* query – which loads a 4 byte int value per tuple – and the *min(50b)* query – which loads 50 bytes of data per tuple – we can observe that the dominating factor in query execution is loading data. Differences between the VM-Fork and other backends are caused by added indirection in case of the HotCold approach or bigger tuple size because of added metadata (e.g. the *ShadowPtr*) resulting in higher memory pressure in the other approaches.

Both *min*-queries were run on memory backed by 4kb as well as 2mb pages. Large pages reduce the number of TLB misses since lookup information of larger chunks of memory can be resolved using the entries inside the TLB. With table scans, no significant improvement can be observed. We sampled the number of TLB misses that occur during scan operations both with 4kb as well as with 2mb pages[1]. In both scenarios, the number of TLB misses is zero or close to zero suggesting that TLB misses have no impact on scan operations since misses are rare to begin with. It is assumed that the low number of TLB misses is due to predictive address resolution done by the CPU because of the sequential access pattern of scan queries.

[1]Samples were taken with *oprofile* [10] which periodically acccesses CPU performance counters during execution.

5.3 OLTP&OLAP CH-BenCHmark

To be able to measure the performance of a hybrid system running OLTP transactions as well as OLAP queries in parallel, the CH-BenCHmark was developed [4, 2]. The benchmark extends the TPC-C schema so that TPC-C transactions as well as queries semantically equivalent to TPC-H queries can be executed on the same database state.

For the purpose of measuring the memory overhead incurred by different granularities in the tested snapshotting techniques, we extended the transactional part of the benchmark to include a transaction implementing warranty and return cases. This changes the access pattern of the TPC-C on the *orderline* relation so that a small number of older tuples (2% on average) is updated even after delivery.

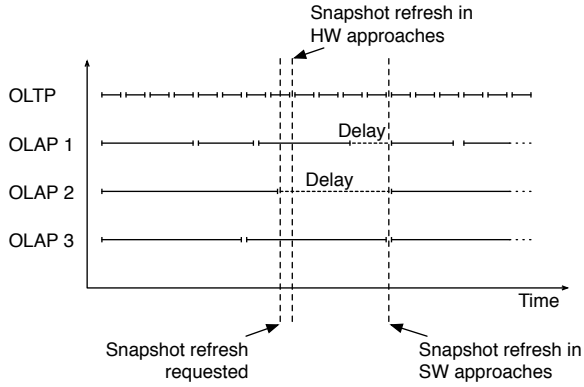


Figure 10: Schematic representation of the CH-BenCHmark used to evaluate all storage backends.

The benchmark was run with one thread executing OLTP transactions and 3 threads concurrently running OLAP queries (specifically, queries 1 and 5 of the TPC-H) on a snapshot. A snapshot refresh was triggered every 200,000 OLTP transactions. A schematic representation of the benchmark is shown in Figure 10. There, the difference between snapshot refresh delays between Hardware Page Shadowing and software controlled snapshotting mechanisms is displayed. When a hardware supported technique is used, only OLTP execution has to be quiesced. With software controlled mechanisms – like Tuple Shadowing – all threads executing OLAP queries have to stopped as well which reduces throughput because queries have to be either aborted or delayed.

When the snapshot renewal is requested and the delaying strategy is employed, no new OLAP queries are admitted in software controlled snapshotting techniques. As soon as all existing OLAP queries have finished, OLTP processing is quiesced and a the snapshot is refreshed. All OLAP threads finished with their query inhibit a delay until the new snapshot is ready, thus reducing OLAP throughput. When a hardware controlled snapshotting mechanism is used, the snapshot can be renewed as soon as all OLTP transactions have been quiesced. Here, the coexistence of multiple snapshots possible in the hardware-controlled mechanisms VM-

Fork and HotCold is beneficial, as the creation of a new snapshot is independent of parallel query execution on old snapshots.

5.3.1 OLTP/OLAP throughput

Table 3 shows the throughput for OLTP transactions as well as OLAP queries. The OLTP transactions correspond to the transactions of the TPC-C. The OLAP queries consist of queries semantically equivalent to queries 1 and 5 of the TPC-H. The two representative OLAP queries are repeatedly executed in an alternating pattern.

Backend	raw	index fork		index copy		index share	
	OLTP	OLTP	OLAP	OLTP	OLAP	OLTP	OLAP
VM-Fork	85k	60k	10.3	59k	9.7	n/a	n/a
Tuple	25k	22k	6.8	19k	5.9	24k	5.9
Twin	33k	29k	7.0	26k	5.9	27k	7.0
HotCold	84k	59k	9.6	59k	9.9	n/a	n/a

Table 3: OLTP and OLAP throughput per second in the CH-BenCHmark.

Looking at OLTP throughput, it can be observed that techniques based on Hardware Page Shadowing yield higher throughput. This has two major reasons: First, Hardware Page Shadowing allows for faster reorganization than software controlled mechanisms as we observed in Section 5.1. Second, there is no indirection as opposed to Tuple Shadowing where a shadow tuple has to be checked, or Twin Tuples where the tuple to be read or written has to be found using a bit flag (c.f. Appendix 4.2).

OLAP query performance is influenced less by the choice of snapshotting mechanism. Compared to a 50% slowdown as seen in OLTP throughput, OLAP queries run about 25% slower when a software controlled snapshotting mechanism is employed. Here, the slowdown is caused by two main factors: Reorganization time and the delay caused by quiescing OLAP queries. All backends have been architected so that OLAP query performance is as high as possible. This is achieved by maintaining tuples included in the snapshot in their original form and position and adding redirection only for new, updated or deleted tuples which can only be seen by OLTP transactions, not OLAP queries.

Throughput for both OLTP transactions as well as OLAP queries varies with different index synchronization mechanisms. For index copy, performance degradation is caused by an increase in reorganization delay of about 1 second per 1000 megabytes index size. When indexes are shared between transactions and queries, reorganization time is unaffected but instead a runtime overhead for acquiring latches is incurred. For index fork – where indexes are shadow copied with vm page granularity – the decrease in OLTP throughput compared to the raw throughput given in Table 3 is the result of both increased fork time as well as runtime overhead. Here, the part of the page table used for index pages needs to be copied as part of the fork causing a small delay in the order of milliseconds per gigabyte index size. Additionally, more significant delays occur whenever a physical page has to be copied causing a significant performance decrease compared to the raw OLTP throughput given. It should be noted that the raw values shown in Table 3 were measured without any index synchronization causing all indexes to be inaccessible for OLAP query processing.

The benchmark was executed on both 4kb and 2mb pages. Techniques involving the `fork` system call experience better fork performance with larger pages. This is due to the fact that the page table – which is copied eagerly on snapshot refresh – is 512 times smaller when using 2mb pages instead of 4kb pages (see 5.1). Apart from performance gains related to smaller delays caused by higher fork performance, a measurable performance gain from bigger pages is experienced by OLTP transactions. There, the memory access pattern is non sequential – as opposed to OLAP table scans. On most architectures (see, for instance, [7]), the size of virtual memory for which address resolution can be performed in hardware using the TLB is significantly larger when using large pages than when small pages are used. Therefore, TLB misses occur less frequently thus increasing transaction throughput. In measurements we conducted for a TPC-C workload, the number of TLB misses was reduced to about 50% when using large pages as opposed to small pages.

Absolute gains in OLTP performance are higher for approaches with a higher inclination to TLB misses. This is the case for both the HotCold and the Tuple Shadowing approach. Since shadow copies can not be easily located close to the original tuple without high memory consumption overhead or diminished OLTP performance, opportunities for TLB misses during OLTP transactions effectively double. Therefore, reducing the number of TLB misses by roughly 50% results in higher absolute savings compared to, for example, Twin Tuples.

OLAP processing does not significantly profit from using large pages. As noted before, any gains from using large pages are either due to shorter reorganization time or less TLB misses. The effects of shorter reorganization time when the `fork` system call is used are insignificant since the entire delay caused by this operation does only account for about 1% of the total runtime of the benchmark. A throughput increase due to a lower TLB miss rate as observed for OLTP transactions can not be observed for queries because the access pattern of OLAP queries is sequential, making prefetching possible. Therefore, the TLB miss rate in OLAP queries can be assumed to be low and thus no significant performance increase can be expected from reducing TLB misses.

5.3.2 Memory consumption

For the same CH-BenCHmark configuration as used in Section 5.3.1, we measured the total memory usage. Figure 11 shows the absolute memory used by our prototypes after executing a given number of OLTP transactions. Memory measurements were taken by monitoring total memory consumption on the test hardware.

It can be observed that the memory curve of all approaches is approximately linear. As the insertion-/update-rate of the CH-BenCHmark is constant, this course was to be expected. The fluctuations in memory consumption result from the parallel execution of OLAP queries which require a certain amount of memory for computations and intermediate results. With no OLAP processing in parallel, the deviation from a linear shape of the curve is no longer visible given the scale used in Figure 11.

The figure includes the four approaches discussed in this paper as well as curves labeled ‘baseline’ and ‘Row w/o OLAP’. ‘Baseline’ is the minimum amount of memory required to save the tuples without applying compression, its value is calculated. ‘Row w/o OLAP’ is the memory required by a row store implementation without any snapshot

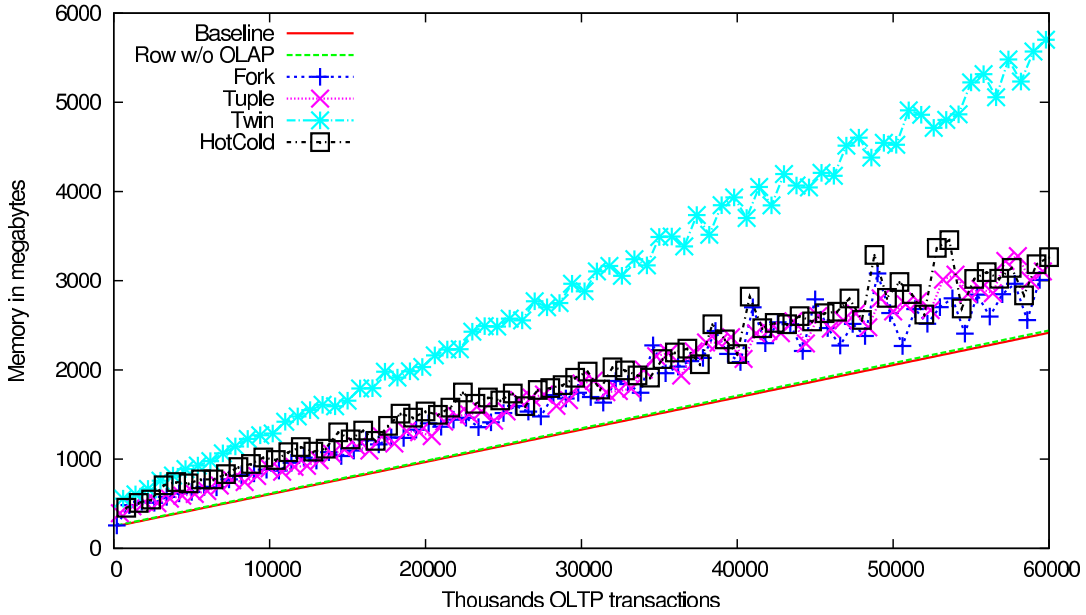


Figure 11: Memory consumed by the different snapshotting techniques over the course of a 6 million OLTP transaction run of the CH-BenCHmark.

mechanism and was measured the same way memory consumption was measured for our snapshot approaches.

The VM-Fork, Tuple Shadowing and HotCold approach each consume roughly equivalent amounts of memory during the benchmark. Compared to baseline memory consumption, the three approaches require roughly 20 to 30% more memory than what is necessary to save the raw data contained in all tuples. The Twin Tuples approach requires about twice as much memory compared to the baseline, which is caused by constantly saving every tuple twice.

The difference in memory consumption between 'Row w/o OLAP' and the approaches researched in this paper can be explained by multiple factors: First, shadow copies consume memory that would not be needed when updating in place. Second, parallel OLAP processing requires memory for intermediate results. Third, more metadata like bitmaps or page table copies need to be kept in memory.

In Figure 11, no clear savings from approaches with finer shadowing granularity can be observed. We believe this is caused by high locality of the TPC-C benchmark as well as small tuple size of those tables which are actually updated.

6. CONCLUSION

Satisfying the emerging requirement for real-time business intelligence demands to execute a mixed OLTP&OLAP workload on the same database system state. In this paper, we analyzed 4 different snapshotting techniques for in-memory DBMS that allow to shield mission-critical OLTP from the longer-running OLAP queries without any additional concurrency control overhead: VM-fork that creates the snapshot by cloning the virtual memory of the database process, Twin Tuples that keeps two copies of each tuple, software-controlled Tuple Shadowing and the HotCold adaptation of the VM-fork. The clear winner in terms of OLTP performance, OLAP query response times and memory consumption is the VM-fork technique which exploits modern multi-core architectures effectively as it allows to create an arbitrary number of time-wise overlapping snapshots with parallel query sessions. The snapshot maintenance is completely delegated to the MMU&OS as they detect and perform the necessary page replications (*copy-on-write*) ultra-efficiently. Thus, the re-emergence of in-memory databases and the progress in hardware supported virtual memory management have led to a promising reincarnation of the shadow paging of the early database days. Unlike the original shadow page snapshots, the hardware controlled VM snapshots are very well suited for processing OLAP queries in a mixed OLTP&OLAP workload.

7. REFERENCES

[1] T. Cao, M. V. Salles, B. Sowell, Y. Yue, J. Gehrke, A. Demers, and W. White. Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications. In *SIGMOD*, 2011.

[2] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompaß, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload ch-benchmark. In *DBTest*, 2011.

[3] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1):48–57, 2010.

[4] F. Funke, A. Kemper, and T. Neumann. Benchmarking Hybrid OLTP&OLAP Database Systems. In *BTW*, 2011.

[5] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Trans. Knowl. Data Eng.*, 4(6):509–516, 1992.

[6] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992,

2008.

[7] Intel. First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem). http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf, 2008.

[8] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[9] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, 2011.

[10] J. Levon. *OProfile Manual*. Victoria University of Manchester, 2004.

[11] R. A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Trans. Database Syst.*, 2(1):91–104, 1977.

[12] T. Neumann. Efficiently compiling efficient query plans for modern hardware. In *VLDB*, 2011.

[13] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD*, pages 1–2, 2009.

[14] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *ICDE*, pages 452–462, 1989.

[15] VoltDB LLC. VoltDB Technical Overview. http://voltdb.com/_pdf/VoltDBTechnicalOverviewWhitePaper.pdf, 2010.

APPENDIX

A. HARD- AND SOFTWARE USED

All tests were conducted on a database server with two Intel Xeon X5570 quad-core CPUs at 2.93GHz. The machine is equipped with 64 gigabytes of main memory and did not swap during any of the tests. All code is written in C++ and compiled with g++ version 4.4 using `-O3` optimization settings. Transactions as well as queries are compiled to machine code using the LLVM 2.8 compiler infrastructure with default optimization passes; see [12] for more information on query compilation.

B. WORDLOAD USED

During the run of the CH-Benchmark, OLAP queries 1 and 5 were execute in an alternating sequence. The queries roughly mimic queries 1 and 5 from the TPC-H benchmark. The schema of the CH-Benchmark which is a combination of the schemas of TPC-C and TPC-H is shown in Figure 12. Further information can be found in the CH-Benchmark paper [2].

B.1 Query 1

```
select ol_number,
       sum(ol_quantity) as sum_qty,
       sum(ol_amount) as sum_amount,
       avg(ol_quantity) as avg_qty,
       avg(ol_amount) as avg_amount,
       count(*) as count_order
from orderline
where ol_delivery_d >
    timestamp '2010-01-01␣00:00:00'
group by ol_number order by     ol_number
```

B.2 Query 5

```
select n_name, sum(ol_amount) as revenue
from customer, order, orderline,
       stock, supplier, nation, region
where
       c_id = o_c_id
       and c_w_id = o_w_id
       and c_d_id = o_d_id
       and ol_o_id = o_id
       and ol_w_id = o_w_id
       and ol_d_id=o_d_id
       and ol_w_id = s_w_id
       and ol_i_id = s_i_id
  -- instead of FK :
       and mod((s_w_id * s_i_id),10000) =
         su_suppkey
       and ascii(substr(c_state,1,1)) =
         su_nationkey
       and su_nationkey = n_nationkey
       and n_regionkey = r_regionkey
       and r_name = 'Europa'
       and o_entry_d >=
         timestamp '2010-01-01␣00:00:00'
group by
      n_name
order by
      revenue desc
```

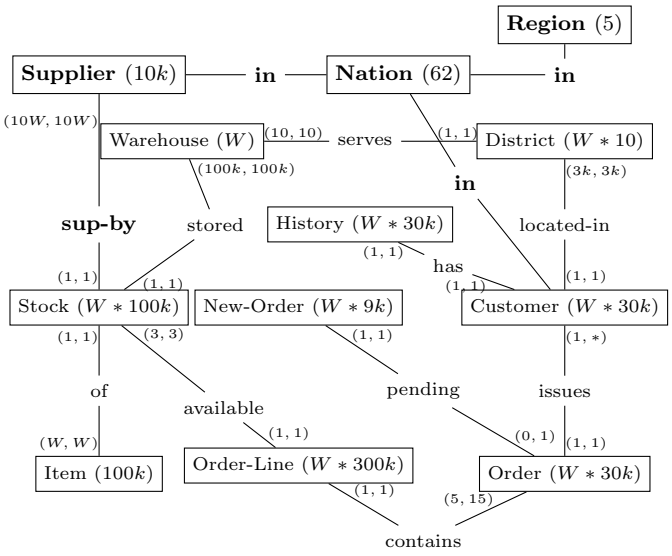


Figure 12: The schema of the CH-Benchmark.

B.3 OLTP Transactions

- New-Order
- Payment
- Delivery
- Order-Status
- Stock-Level