# HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots

Alfons Kemper[1], Thomas Neumann[2]

*Fakultät für Informatik*
*Technische Universität München*
*Boltzmannstraße 3, D-85748 Garching*
[1]kemper@in.tum.de
[2]neumann@in.tum.de

***Abstract*—The two areas of online transaction processing (OLTP) and online analytical processing (OLAP) present different challenges for database architectures. Currently, customers with high rates of mission-critical transactions have split their data into two separate systems, one database for OLTP and one so-called *data warehouse* for OLAP. While allowing for decent transaction rates, this separation has many disadvantages including data freshness issues due to the delay caused by only periodically initiating the Extract Transform Load-data staging and excessive resource consumption due to maintaining two separate information systems. We present an efficient hybrid system, called *HyPer*, that can handle both OLTP and OLAP simultaneously by using hardware-assisted replication mechanisms to maintain consistent snapshots of the transactional data. HyPer is a main-memory database system that guarantees the ACID properties of OLTP transactions and executes OLAP query sessions (multiple queries) on the same, arbitrarily current and consistent snapshot. The utilization of the processor-inherent support for virtual memory management (address translation, caching, copy on update) yields both at the same time: unprecedentedly high transaction rates as high as 100000 per second and very fast OLAP query response times on a single system executing both workloads in parallel. The performance analysis is based on a combined TPC-C and TPC-H benchmark.**

## I. Introduction

Historically, database systems were mainly used for online transaction processing. Typical examples of such transaction processing systems are sales order entry or banking transaction processing. These transactions access and process only small portions of the entire data and, therefore, can be executed quite fast. According to the standardized TPC-C benchmark results the currently most powerful systems can process more than 100.000 such sales transactions per second.

About two decades ago a new usage of database systems evolved: Business Intelligence (BI). BI-applications rely on long running so-called Online Analytical Processing (OLAP) queries that process substantial portions of the data in order to generate reports for business analysts. Typical reports include aggregated sales statistics grouped by geographical regions, or by product categories, or by customer classifications, etc. Initial attempts – such as SAP's EIS project [1] – to execute these queries on the operational OLTP database were dismissed as the OLAP query processing led to resource contentions and severely hurt the mission-critical transaction processing. Therefore, the data staging architecture was devised where the transaction processing is carried out on a dedicated OLTP database system. In addition, a separate Data Warehouse system is installed for business intelligence query processing. Periodically, e.g., during the night, the OLTP database changes are extracted, transformed to the layout of the data warehouse schema, and loaded into the data warehouse. This data staging and its associated ETL (Extract–Transform–Load) obviously incurs the problem of *data staleness* as the ETL process can only be executed periodically.

Recently, strong arguments for so-called *real time business intelligence* were made. Hasso Plattner, the co-founder of SAP, advocates the "data at your fingertips"-goal for enterprise resource planning systems [2]. The currently exercised separation of transaction processing on the OLTP database and BI query processing on the *data warehouse* that is only periodically refreshed violates this goal as business analysts have to base their decisions on stale (outdated) data. Real-time/operational business intelligence demands to execute OLAP queries on the current, up-to-date state of the transactional OLTP data. We propose to enhance the transactional database with highly effective query processing capabilities – thereby shifting (some of) the query processing from the DW to the OLTP system. Therefore, mixed workloads of OLTP transaction processing and OLAP query processing on the same database have to be supported. This is somewhat counter to the recent trend of building dedicated systems for different application scenarios. The integration of these two very different workloads on the same system necessitates drastic performance improvements which can be achieved by main-memory database architectures.

On first view, the dramatic explosion of the (Internet accessible) data volume may contradict this premise of keeping all transactional data main memory-resident. However, a closer examination shows that the business critical transactional database volume has limited size, which favors main memory data management. To corroborate this assumption let us analyze one of the largest commercial enterprises, Amazon, which has a yearly revenue of about 15 billion Euros. Assuming that an individual order line values at about 15 Euros and each order line incurs stored data of about 54 bytes – as specified for the TPC-C-benchmark –, we derive a total data volume of 54 GB per year for the order lines which is the dominating repository in such a sales application. This estimate neither includes the other data (customer and product data) which

978-1-4244-8960-2/11/$26.00 © 2011 IEEE

increases the volume nor the possibility to compress the data to decrease the volume. Nevertheless it is safe to assume that the yearly sales data can be fit into main memory of a large scale server. This was also analyzed by Ousterhout et. al. [3] who proclaim the so-called RAMcloud as a main-memory storage device for the largest Internet software applications. Extrapolating the past developments it is safe to forecast that the main memory capacity of commodity as well as high-end servers is growing faster than the largest business customer's requirements. For example, Intel announced a large multi-core processor with several TB of main memory as part of its so-called Tera Scale initiative [4]. We are currently in the process of ordering a TB server from Dell for a "mere" 60000 Euros.

The transaction rate of such a large scale enterprise with 15 billion Euro revenue can be estimated at about 32 order lines per second. Even though the arrival rate of such business transactions is highly skewed (e.g., Christmas sales peaks) it is fair to assume that the peak load will be below a few thousand order lines per second.

For our HyPer system we adopt a main-memory architecture for transaction processing. We follow the lock-less approach first advocated in [5] whereby all OLTP transactions are executed sequentially – or on private partitions. This architecture obviates the need for costly locking and latching of data objects or index structures as the sole update transaction "owns" the entire database – or its private partition of the database. Obviously, this serial execution approach is only viable for a pure main memory database where there is no need to mask IO operations on behalf of one transaction by interleavingly utilizing the CPUs for other transactions. In a main-memory architecture a typical business transaction (e.g., an order entry or a payment processing) has a duration of only a few up to ten microseconds. Such a system's viability for OLTP processing was previously proven in a research prototype named H-Store [6] conducted by researchers led by Mike Stonebraker at MIT, Yale and Brown University. The H-Store prototype was recently commercialized by a start-up company named VoltDB.

However, the H-Store architecture is limited to OLTP transaction processing only. If we simply allowed complex OLAP-style queries to be injected into the workload queue they would clog the system, as all subsequent OLTP transactions have to wait for the completion of such a long running query. Even if such OLAP queries finish within, say, 30 ms they lock the system for a duration in which around 1000 or more OLTP transactions could have completed.

Nevertheless, our goal was to architect a main-memory database system that can

- process OLTP transactions at rates of tens or hundreds of thousands per second as efficiently as dedicated OLTP main memory systems such as VoltDB or TimesTen, and, at the same time,
- process OLAP queries on up-to-date snapshots of the transactional data as efficiently as dedicated OLAP main memory DBMS such as MonetDB or TREX.

This challenge is sketched in Figure 1. We architected such an efficient hybrid system, called *HyPer*, that can handle both OLTP and OLAP simultaneously by using hardware-assisted replication mechanisms to maintain consistent snapshots of the transactional data. HyPer is a main-memory database system that guarantees the ACID properties of OLTP transactions. In particular, we devised logging and backup archiving schemes for durability, atomicity and fast recovery. In parallel to the OLTP processing, HyPer executes OLAP query sessions (multiple queries) on the same, arbitrarily current and consistent snapshot. These snapshots are created by forking the OLTP process and thereby creating a consistent virtual memory snapshot. This snapshot is kept consistent via the implicit OS/processor-controlled lazy copy-on-write mechanism. The utilization of the processor-inherent support for virtual memory management (address translation, caching, copy on update) accomplishes both in the same system and at the same time: unprecedentedly high transaction rates of millions of transactions per minute as high as any OLTP-optimized database system and ultra-low OLAP query response times as low as the best OLAP-optimized column stores. These numbers were achieved on a commodity desktop server. Even the creation of a fresh, transaction-consistent snapshot can be achieved in subseconds.

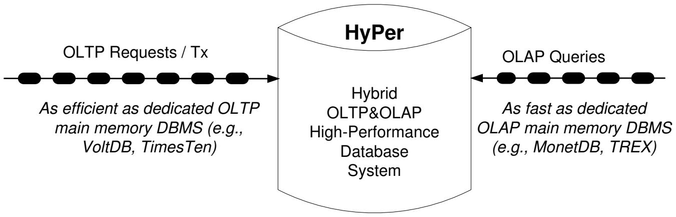
HyPer
OLTP Requests / Tx
OLAP Queries
As efficient as dedicated OLTP main memory DBMS (e.g., VoltDB, TimesTen)
Hybrid OLTP&OLAP High-Performance Database System
As fast as dedicated OLAP main memory DBMS (e.g., MonetDB, TREX)

Fig. 1. Hybrid OLTP&OLAP Database Architecture

## II. Related Work/Systems

HyPer is a new RISC-style database systems [7] like RDF-3X [8] (albeit for a very different purpose). Both systems are developed from scratch. Thereby, historically motivated ballast of traditional database systems is omitted and new hardware and OS-functionality can be leveraged.

The development of main memory database systems (or in-memory DBMS) originally started for the OLTP world. TimesTen [9] was among the first such systems and was recently acquired by Oracle and primarily serves as a "front" cache for the Oracle mainstream database system. P*TIME / Transact in Memory [10] was acquired by SAP in 2005. SolidDB of Solid Information Technology is a main memory DB developed in Helsinki. In the meantime IBM took over this company. For SolidDB the tuple level [11] snapshots were proposed that are kept consistent by tuple shadowing instead of page shadowing. The authors report 30 % transactional throughput increase and a smaller main memory footprint. The page-level shadowing dates back to the early ages of relational database system development [12]. In HyPer we rely on hardware-supported page shadowing that is controlled by the processor's memory management unit (MMU). For disk based database systems shadowing was not really successful because it destroys the page clustering. This hurts the scan performance, e.g., for a full table scan, as the disk's read/write head has to be moved. HyPer is based on virtual memory supported shadow paging where scan performance is not hurt by shadowing. In main memory there is no difference between accessing two consecutive physical memory pages versus accessing two physical pages that are further apart. Furthermore, the snapshots based on VM shadowing do not affect the logical page layout, i.e., potentially non-sequential physical page accesses are hidden by the hardware.

Most recently, the drastic increase of main memory capacity and the demand for real-time/operational business intelligence has led to a revival of main memory database system research and commercial development. The recent main-memory database systems can be separated by their application domain: OLAP verus OLTP. MonetDB is the most influential database research project on column store storage schemes for an in-memory OLAP database. An overview of the system can be found in the summary paper [13] presented on the occasion of receiving the 10 year test of time award of the VLDB conference. TREX [14] is SAP's most prominent database project that relies, like MonetDB, on the column-major storage scheme. It is now known as Business Warehouse Accelerator and serves as the basis for SAP's business intelligence functionality. According to Hasso Plattners key note at SIGMOD 2009 [2] SAP intends to extend it to include OLTP functionality and then make it the basis for hosted applications, e.g., Business by Design. The hybrid system is apparently a combination of TREX and P*TIME and relies on merging the OLTP updates periodically into the column store of the OLAP TREX database [15]. In HyPer this merge is implicit and hardware-supported by creating a new VM snapshot.

Based on an early study for banking transactions [16] the authors of H-Store [17], [6], [5] deserve the credit for analyzing the overhead imposed by various traditional database management features (buffer management, logging, locking, etc.). They proved the feasibility of a main memory database system that processes transactions sequentially without synchronization overhead. VoltDB [18] is the commercialization of H-Store. The published VoltDB performance numbers are largely due to database partitioning across a compute cluster. [19] devised synchronization concepts for allowing inter-partition transactions. Ulusoy and Buchmann [20] investigated main memory database partitioning for optimized concurrency control for real time applications. The automatic derivation of partitioning schemes is an old research issue of distributed database design and receives renewed interest [21].

HyPer's partitioning technique (cf. Section III-D) is primarily used for intra-node parallelism and is particularly beneficial for multi-tenancy database applications [22].

Crescando is a research project at ETH Zürich [23] that processes queries in a batch by periodically scanning all the data in a similar fashion as executing continuous queries over streaming data. At EPFL Lausanne several projects around the database system Shore have the goal to optimize the locking [24] and logging [25] performance on modern multi-core processors. Blink and its commercial product IBM Smart Analytics Optimizer (ISAO) [26], [27] are recent developments at IBM to augment an OLTP database system with an in-memory database for OLAP queries. Their original design was based on materializing all the joins and use compression to reduce the size of the resulting in-memory data.

## III. System Architecture

The HyPer architecture was devised such that OLTP transactions and OLAP queries can be performed on the same main memory resident database – without interfering with each other. In contrast to old-style disk-based storage servers we omitted any database-specific buffer management and page structuring. The data resides in quite simple, main-memory optimized data structures within the virtual memory. Thus, we can exploit the OS/CPU-implemented address translation at "full speed" without any additional indirection. We currently experiment with the two predominant relational database storage schemes: In the *row store* approach we maintain relations as arrays of entire records and in the *column store* approach the relations are vertically partitioned into vectors of attribute values. Currently, the HyPer prototype is globally configured to operate as a column or a row store – but in future work the table layout will be adjustable according to the access patterns.

Even though the virtual memory can (significantly) outgrow the physical main memory we limit the database to the size of the physical main memory in order to avoid OS-controlled swapping of virtual memory pages.

### A. OLTP Processing

Since all data is main-memory resident there will never be a halt to await IO. Therefore, we can rely on a single-threading approach first advocated in [5] whereby all OLTP transactions are executed sequentially. This architecture obviates the need for costly locking and latching of data objects as the sole update transaction "owns" the entire database. Obviously, this serial execution approach is only viable for a pure main memory database where there is no need to mask IO operations on behalf of one transaction by interleavingly utilizing the CPUs for other transactions. In a main-memory architecture a typical business transaction (e.g., an order entry or a payment processing) has a duration of only around ten $\mu$s. This translates to throughputs in the order of tens of thousands per second, much more than even large scale business applications require – as analyzed in the Introduction.

The serial execution of OLTP transactions is exemplified in Figure 2 by the queue on the left-hand side in which the transactions are serialized to await execution. The transactions are implemented as stored procedures in a high-level scripting language. This language provides the functionality to look-up database entries by search key, iterate through sets of objects, insert, update and delete data records, etc. The high-level scripting code is then compiled by the HyPer system into low-level code that directly manipulates the in-memory data structures.

Obviously, the OLTP transactions have to guarantee short response times in order to avoid long waiting times for

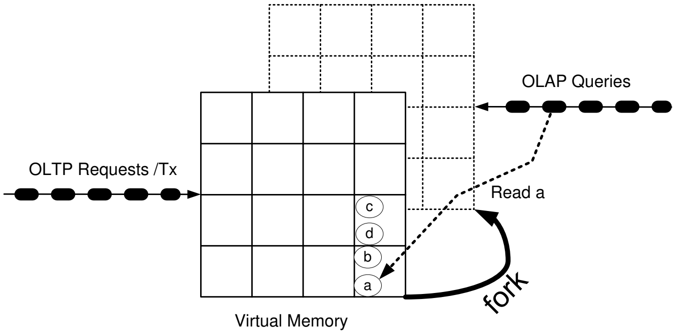

Fig. 2. Forking a New Snapshot

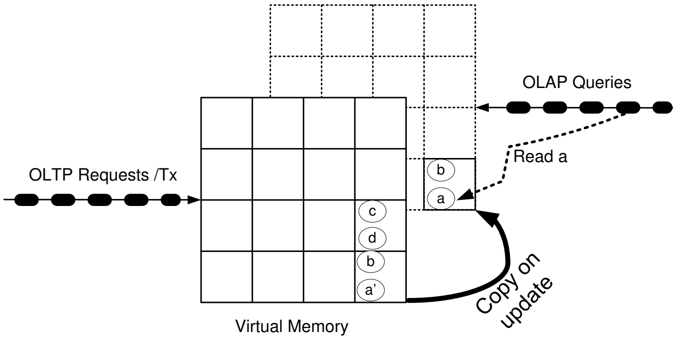

Fig. 3. Copy on Update to Preserve Consistent Snapshot

subsequent transactions in the queue. This prohibits any kind of interactive transactions, e.g., requesting user input or synchronously invoking a credit card check of an external agency. This, however, does not constitute a real limitation as our experience with high-performance business applications, such as SAP R/3 [28], [29] reveals that these kinds of interactions occur outside the database context in the application servers.[1]

### B. OLAP Snapshot Management

If we simply allowed complex OLAP-style queries to be injected into the OLTP workload queue they would clog the system, as all subsequent OLTP transactions have to wait for the completion of such a long running query. Even if such OLAP queries finish within, say, 30 ms they lock the system for a duration in which possibly thousands of OLTP transactions could have completed. To achieve our goal of architecting a main-memory database system that

- processes OLTP transactions at rates of tens of thousands per second, and, at the same time,
- processes OLAP queries on up-to-date snapshots of the transactional data

we exploit the operating systems functionality to create virtual memory snapshots for new, duplicated processes. In Unix, for example, this is done by creating a child process of the OLTP process via the `fork()` system call. To guarantee transactional consistency, the `fork()` should only be executed in between two (serial) transactions, never in the middle of one transaction. In section IV-F we will relax this constraint by utilizing the undo log to convert an action consistent snapshot (created in the middle of a transaction) into a transaction consistent one.

The forked child process obtains an exact copy of the parent processes address space, as exemplified in Figure 2 by the overlayed page frame panel. This virtual memory snapshot that is created by the `fork()`-operation will be used for executing a session of OLAP queries – as indicated on the right hand side of Figure 2.

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 3. Initially, parent process (OLTP) and child process (OLAP) share the same physical memory segments by translating either virtual addresses (e.g., to 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 initiate the replication of the virtual memory page on which $a$ resides. 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 the OLAP query session. 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 (cf. Figure 4).

Another intuitive way to view the functionality is as follows: The OLTP process operates on the entire database, part of which is shared with the OLAP module. All OLTP changes are applied to a separate copy (area), the Delta – consisting of copied (shadowed) database pages. Thus, the OLTP process creates its working set of updated pages on demand. This is somewhat analogous to swapping pages into a buffer pool – however, the copy on demand of updated pages is three to four orders of magnitude faster as it takes only 2 $\mu$s to copy a main memory page instead of 10 ms to handle a page fault in the buffer pool. Every "now and then" the Delta is merged with the OLAP database by forking a new process for an up-to-date OLAP session. Thereby, the Delta is conceptually re-integrated into the (*main snapshot*) database. Unlike any software solution for merging a Delta back into the main database, our hardware-supported virtual memory merge (fork) can be achieved very efficiently in subseconds.

The replication (into the Delta) is carried out at the granularity of entire pages, which usually have a default size of 4 KB. In our example, the state change of $a$ to $a'$ induces not only the replication of $a$ but also of all other data items on this page, such as $b$, even though they have not changed. This is the price we opt to pay in exchange for relying on the very effective and fast virtual memory management by the OS and the processor, such as ultra-efficient VM address transformation

[1] Nevertheless, we are currently devising an optimistic lock-less concurrency scheme for long-running transactions being executed in our system.

via TLB caching and copy-on-write enforcement. Also, it should be noted that the replicated pages only persist until the OLAP session terminates – usually within seconds or minutes. Traditional shadowing concepts in database systems are based on pure software mechanisms that maintain shadow copies at the page level [30] or shadow individual objects [11].

Our snapshots incur storage overhead proportional to the number of updated pages by the parent process (i.e., the OLTP request executing process). It replicates the Delta (corresponding to the changed pages) between the memory state of the OLTP process at the time when the `fork()` created the snapshot and the current memory state of the OLTP process. The OLAP processes never change the shared pages – which would of course be unproblematic because of the copy-on-update mechanism. However, to increase performance they should allocate their temporary data structures in non-shared main memory areas. If the main memory capacity is scarce, the OLAP query engine can employ secondary storage devices (e.g. disks), thereby trading main memory capacity for longer execution time. Sorting a relation by creating disk-based runs is one prominent example. All OLAP queries, denoted by the ovals, in the "OLAP Queries" queue access the same consistent snapshot state of the database. We call such a group of queries a *Query Session* to denote that a business analyst could use such a session for a detailed analysis of the data by iteratively querying the same state to, e.g., drill down to more details or roll up for a better overview.

### C. Multiple OLAP Sessions

So far we have sketched a database architecture utilizing two processes, one for OLTP and another one for OLAP. As the OLAP queries are *read-only* they could easily be executed in parallel in multiple threads that share the same address space. Still, we can avoid any synchronization (locking and latching) overhead as the OLAP queries do not share any mutable data structures. Modern multi-core computers which typically have more than ten cores can certainly yield a substantial speed up via this inter-query parallelization.

Another possibility to make good use of the multi-core servers is to create multiple snapshots. The HyPer architecture allows for arbitrarily current snapshots. This can simply be achieved by periodically (or on demand) `fork()`-ing a new snapshot and thus starting a new OLAP query session process. This is exemplified in Figure 4. Here we sketch the one and only OLTP process'es current database state (the front panel) and three active query session processes' snapshots – the oldest being the one in the background. The successive state changes are highlighted by the four different states of data item $a$ (the oldest state), $a'$, $a''$, and $a'''$ (the youngest transaction consistent state). Obviously, most data items do not change in between different snapshots as we expect to create snapshots for most up-to-date querying at intervals of a few seconds – rather than minutes or hours as is the case in current separated data warehouse solutions with ETL data staging. The number of active snapshots is, in principle, not limited, as each "lives" in its own process. By adjusting the priority we can make sure that the mission critical OLTP process is always allocated a core – even if the OLAP processes are numerous and/or utilize multi-threading and thus exceed the number of cores.

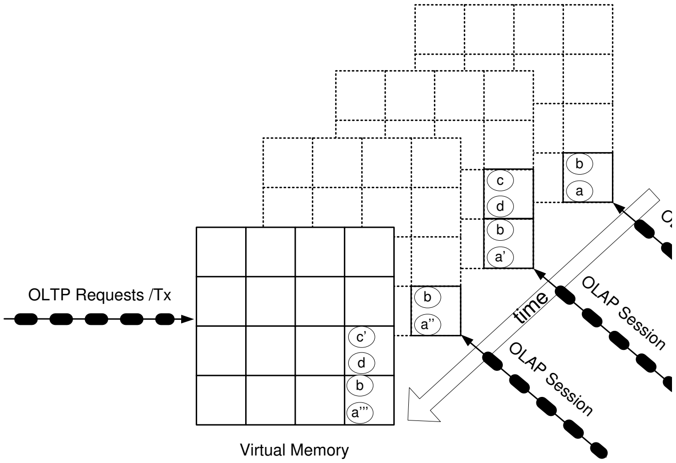

Fig. 4. Multiple OLAP Sessions at Different Points in Time

A snapshot will be deleted after the last query of a session is finished. This is done by simply terminating the process that was executing the query session. It is not necessary to delete snapshots in the same order as they were created. Some snapshots may persist for a longer duration, e.g., for detailed stocktaking purposes. However, the memory overhead of a snapshot is proportional to the number of transactions being executed from creation of this snapshot to the time of the next younger snapshot (if it exists or to the current time). The figure exemplifies this on the data item $c$ which is physically replicated for the "middle age" snapshot and thus shared and accessible by the oldest snapshot. Somewhat against our intuition, it is still possible to terminate the middle-aged snapshot before the oldest snapshot as the page on which $c$ resides will be automatically detected by the OS/processor as being shared with the oldest snapshot via a reference counter associated with the physical page. Thus it survives the termination of the middle-aged snapshot – unlike the page on which $a'$ resides which is freed upon termination of the middle-aged snapshot process. The youngest snapshot accesses the state $c'$ that is contained in the current OLTP process'es address space.

### D. Multi-Threaded OLTP Processing

We already outlined that the OLAP process may be configured as multiple threads to better utilize the multiple cores of modern computers. This is also possible for the OLTP process, as we will describe here. One simple extension is to admit multiple read-only OLTP transactions in parallel. As soon as a read/write-transaction is at the front of the OLTP workload queue the system is quiesced and transferred back into sequential mode until no more update-transactions are at the front of the queue. In realistic applications we expect many more read-only transactions than update transactions – therefore we can expect to obtain some level of parallelism, which could even be increased by (carefully) rearranging the OLTP workload queue.

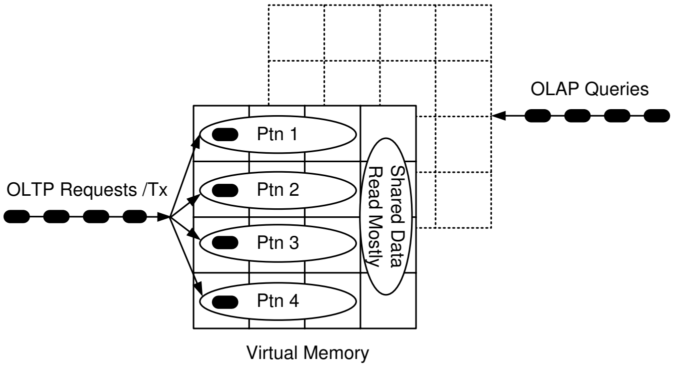

Fig. 5. Multi-Threaded OLTP Processing on Partitioned Data

There are many application scenarios where it is natural to partition the data. One very important application class for this is multi-tenancy – as described in [22]. The different database users (called tenants) work on the same or similar database schemas but do not share their transactional data. Rather, they maintain their private partitions of the data. Only some read-mostly data (e.g., product catalogs, geographical information, business information catalogs like Dun & Bradstreet) is shared among the different tenants.

Interestingly, the TPC-C benchmark exhibits a similar partitioning as most of the data can be partitioned horizontally by the Warehouse, to which it belongs. The only exception is the *Items* table, which corresponds to our read-mostly, shared data partition.

In such a partitioned application scenario HyPer's OLTP process can be configured as multiple threads – to increase performance even further via parallelism. This is sketched out in Figure 5. As long as the transactions access and update only their private partition and access (not update) the shared data we can run multiple such transactions in parallel – one per partition. This is shown in the figure where each oval (representing a transaction) inside the panel corresponds to one such partition-constrained transaction executed by a separate thread.

However, transactions reading across partitions or updating the shared data partition require synchronization. For the VoltDB partitioned database two synchronization methods were analyzed in [21]: a lock-based approach and an optimistic method that may necessitate cascaded roll-backs.

In our current HyPer-prototype cross-partition transactions request exclusive access to the system – just as in our initial purely sequential approach. This is sufficiently efficient in a central system where all partitions reside on one node. However, if the nodes are distributed across a compute cluster, which necessitates a two-phase commit protocol for multi-partition transactions, more advanced synchronization approaches are beneficial. The synchronization aspects are further detailed in Section IV-C.

OLAP snapshots can be forked as before – except that we have to quiesce all threads before this can be done in a transaction consistent manner. Again, we refer to Section IV-F for a relaxation of this requirement by transforming action consistent snapshots into transaction consistent ones via the undo log. The OLAP queries can be formulated across all partitions and the shared data, which is even needed in multi-tenancy applications for administrative purposes.

The partitioning of the database can be further exploited for a distributed system that allocates the private partitions to different nodes in a compute cluster. The read-mostly, shared partition can be replicated across all nodes. Then, partition-constrained transactions can be transferred to the corresponding node and run in parallel without any synchronization overhead. Synchronization is needed for partition-crossing transactions and for the synchronized snapshot creation across all nodes.

## IV. Transaction Semantics and Recovery

Our OLTP/OLAP transaction model corresponds most closely to the multiversion mixed synchronization method, as described by Bernstein, Hadzilacos and Goodman [31] (Section 5.5). In this model, *updaters* (in our terminology OLTP transactions including the read-only OLTP transactions) are fully serializable and read-only *queries* (our OLAP queries) access the database in a "frozen" transaction consistent state that existed at a point in time before the query was started.

Recently, such relaxed synchronization methods have regained attention as full serializability was, in the past, considered too costly for scalable systems. HyPer achieves both: utmost scalability via OLAP snapshots and full serializability for OLTP processing. A variation of the multiversion synchronization is called snapshot isolation and was first described in [32]. It currently gains renewed interest in the database research community – see, e.g., [33], [34]. Herein, the snapshot synchronization is not constrained to read-only queries but also to the read requests in update transactions.

### A. Snapshot Isolation of OLAP Query Sessions

In snapshot isolation a transaction/query continuously sees the transaction consistent database state as it existed at a point in time (just) before the transaction started. There are different possibilities to implement such a snapshot – while database modifications are running in parallel:

*Roll-Back:* This method, as used in Oracle, updates the database objects in place. If an older query requires an older version of a data item it is created by undoing all updates on this object. Thus, an older copy of the object is created in a so-called roll-back segment by reversely applying all undo log records up to the required point in time.

*Versioning:* All object updates create a new timestamped version of the object. Thus, a read on behalf of a query retrieves the youngest version (largest timestamp) whose timestamp is smaller than the starting time of the query. The versioned objects are either maintained durably (which allows time travelling queries) or temporarily until no more active query needs to access them.

*Shadowing [30]:* Originally, shadowing was invented to obviate the need for undo logging as all changes were written to shadows first and then installed in the database at transaction commit time. However, the shadowing concept can also be applied to maintaining snapshots.

*Virtual Memory Snapshots:* Our snapshot mechanism explicitly creates a snapshot for a series of queries, called a query session. In this respect, all queries of a Query Session are bundled to one transaction that can rely on the transaction consistent state preserved via the `fork()`-process.

### B. Transaction Consistent Archiving

We can also exploit the VM snapshots for creating backup archives of the entire database on non-volatile storage. This process is sketched on the lower right hand side of Figure 6. Typically, the archive is written via a high-bandwidth network of 1 to 10 Gb/s to a dedicated storage server within the same compute center. It is beneficial to use an rDMA interface (e.g., Myrinet or Infiniband) in order to unburden the server's CPU from the data transmission task. To maintain this transfer speed the storage server has to employ several (more than 10) disks for a corresponding aggregated bandwidth.

### C. OLTP Transaction Synchronization

In the single-threaded mode the OLTP transactions do not need any synchronization mechanisms as they own the entire database.

In the multi-threaded mode (cf. Section III-D) we distinguish two types of transactions:

- **partition-constrained transactions** can read and update the data in their own partition as well as read the data in the shared partition. However, the updates are limited to their own partition.
- **partition-crossing transactions** are those that, in addition, update the shared data or access (read or update) data in another partition.

Partition crossing transactions should be rare as updates to shared data seldom occur and the partitioning is derived such that transactions usually operate only on their own data. The classification of the stored procedure transactions in the OLTP workload is done automatically based on analyzing their implementation code and their invocation parameters. If, during execution it turns out that a transaction was erroneously classified as "partition constrained" it is rolled back and reinserted into the OLTP workload queue as "partition crossing."

The HyPer system admits at most one partition constrained transaction per partition in parallel. Therefore, there is no need for any kind of locking or latching as the partitions have non-overlapping data structures and the shared data is accesses read-only.

A partition crossing transaction, however, has to be admitted in exclusive mode. In essence, it has to preclaim an exclusive lock (or, in POSIX terminology, it has to pass a barrier before being admitted) on the entire database before it is admitted. Thus, the execution of partition crossing transactions is relatively costly as they have to wait until all other transactions are terminated and for their duration no other transactions are admitted. Once admitted to the system, the transaction runs at full speed as the exclusive admittance of partition crossing transactions again obviates any kind of locking or latching synchronisation of the shared data partition or the private data partitions.

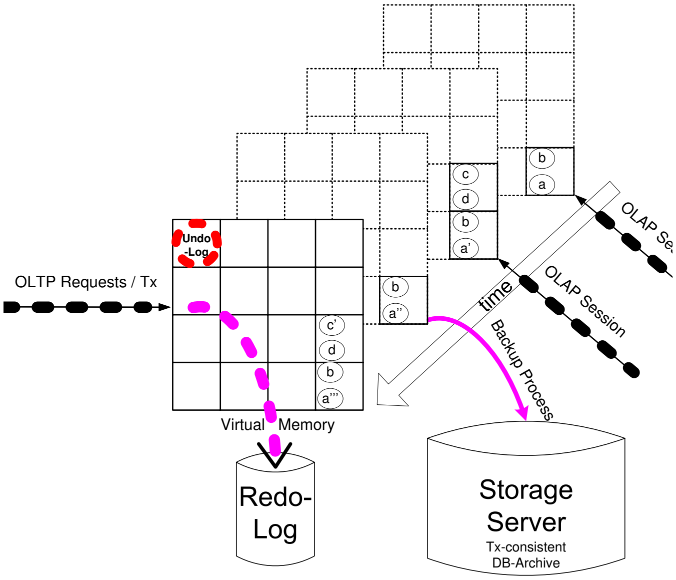

Fig. 6. Durable Redo and Volatile Undo Logging

### D. Durability

The *durability* of transactions requires that all effects of committed transactions have to be restored after a failure. To achieve this we employ classical redo logging in HyPer. This is highlighted by the gray/pink ovals emanating from the serial transaction stream leading to the non-volatile Redo-Log storage device in Figure 6. We employ *logical* redo logging [35] by logging the parameters of the stored procedures that represent the transactions. In traditional database systems logical logging is problematic because after a system crash the database may be in an action-inconsistent state. This cannot happen in HyPer as we restart from a transaction consistent archive (cf. Figure 6). It is only important to write these logical log records in the order in which they were executed in order to be able to correctly recover the database. In the single threaded OLTP configuration this is easily achieved. For the multi-threaded system only the log records of the partition crossing transactions have to be totally ordered relative to all transactions while the partition constrained transactions' log records may be written in parallel and thus only sequentialized per partition.

*High Availability and OLAP Load Balancing via Secondary Server:* The redo log stream can also be utilized to maintain a secondary server. This secondary HyPer server merely executes the same transactions as the primary server. In case of a primary server failure the transaction processing is switched over to the secondary server. However, we do not propose to abandon the writing of redo log records to stable storage and to only rely on the secondary server for fault tolerance. A software error may – in the worst case – lead to a "synchronous" crash of primary and secondary servers.

The secondary server is typically under less load as it needs not execute any read-only OLTP transactions and, therefore,

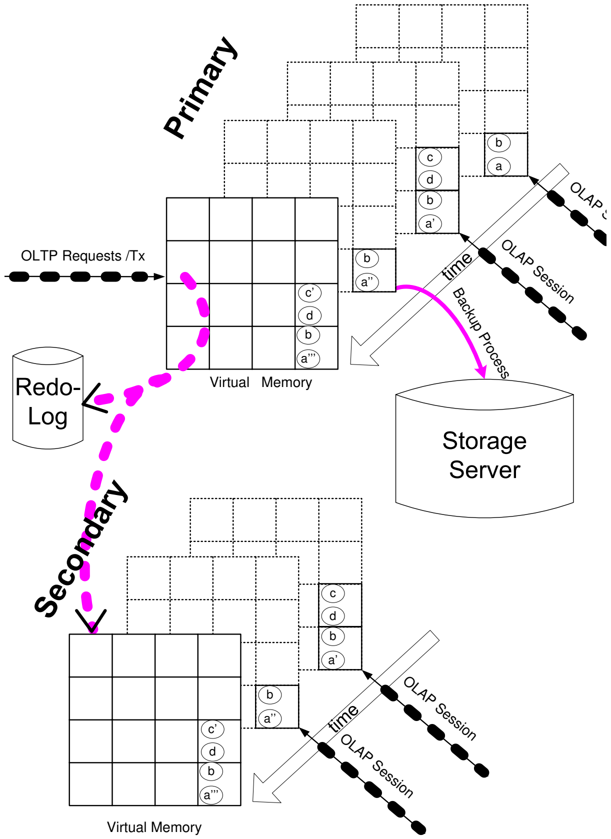

Fig. 7. Secondary Server: Stand-By for OLTP and Active for OLAP

has less OLTP load than the primary server. This can be exploited by delegating some (or all) of the OLAP querying sessions to the secondary server. Instead of – or in addition to – forking an OLAP session's process on the primary server we could just as well use the secondary server.

The usage of a secondary server that acts as a stand-by for OLTP processing and as an active OLAP processor is illustrated in Figure 7. Not shown in the figure is the possibility to use the secondary server instead of the primary server for writing a consistent snapshot to a storage server's archive. Thereby, the backup process is delegated from the primary to the less-loaded secondary server.

*Optimization of the Logging:* The write ahead logging (WAL) principle may turn out to become a performance bottleneck as it requires to flush log records before committing a transaction. This is particularly costly in a single-threaded execution as the transaction – and all succeeding ones – have to wait.

Two commonly employed strategies that were already described by DeWitt et. al. [36] and extended in the recent paper about the so-called Aether system [25] are possible: **Group commit** or **asynchronous commit**.

Group commit is, for example, configurable in DB2 or MS SQL Server. A final commit of a transaction is not executed right after the end of a transaction. Rather, log records of several transactions are accumulated and flushed in a batched mode. Thus, the acknowledgment of a commit is delayed. While waiting for the batch of transactions to complete and their log records being flushed all their locks are already freed. This is called early log release (ELR) and does not jeopardize the serializability correctness. In our non-locking system this translates to admitting the next transaction(s) for the corresponding partition – viewing admission as granting an exclusive lock for the entire partition. Once the log buffer is flushed for the group of transactions, their commit is acknowledged to the client.

Another, less safe, method can be configured in Oracle and PostgreSQL. It relaxes the WAL principle by avoiding to wait for the flushing of the log records. As soon as the log records are written into the volatile log buffer the transaction is committed. This is called *asynchronous commit*. In the case of a failure some of these log records may be lost and thus the recovery process will miss those committed transactions during restart.

### E. Atomicity

The atomicity of transactions requires being able to eliminate any effects of a failed transaction from the database. We only have to consider explicitly aborted transactions, called the R1-recovery. The so-called R3-recovery that demands that updates of "loser"-transactions (those that were active at the time of the crash) are undone in the restored database is not needed in HyPer, as the database is in volatile memory only and the logical redo logs are written only at the time when the successful commit of the transaction is guaranteed. Furthermore, the archive copy of the database that serves as the starting point for the recovery is transaction consistent and, therefore, does not contain any operations that need to be undone during recovery (cf. Figure 6). As a consequence, undo logging is only needed for the active transaction (in multi-threaded mode for all active transactions) and can be maintained in volatile memory only. This is highlighted in Figure 6 by the ring buffer in the top left side of the page frame panel. During transaction processing the before images of any updated data objects are logged into this buffer. The size of the ring buffer is quite small as it is bounded by the number of updates per transaction (times the number of active transactions in multi-threaded operation).

### F. Cleaning Action Consistent Snapshots

Undo-logging can also be used to create a transaction consistent snapshot out of an action-consistent VM snapshot that was created while some transactions were still active. This is particularly beneficial in a multi-threaded OLTP system as it avoids completely quiescing transaction processing. After forking the OLAP process including its associated VM snapshot the undo log records are applied to the snapshot state – in reverse chronological order. As the undo log buffer reflects all effects of active transactions (at the time of the fork) – and only those – the resulting snapshot is transaction-consistent and reflects the state of the database before initiation of the transactions that were still active at the time of the fork.

### *G. Recovery after a System Failure*

The recovery process is based on the durable storage of the database archive and the redo log – cf. Figure 6. During recovery we can start out with the youngest fully written archive, which is restored into main memory. Then the redo log is applied in chronological order – starting with the first redo log entry after the fork for the snapshot of the archive. As the archive can be restored at a bandwidth of up to 10 Gb/s (limited by the network's bandwidth from the storage server) and the redo log can be applied at transaction rates of 100,000 per second the fail-over time for a typical large enterprise (e.g., 100 GB database and thousands of tps) is in the order of one to a few minutes only – if backup archives are written on an hourly basis. If this fail-over time cannot be tolerated it is also possible to rely on replicated HyPer-servers – as sketched in Figure 7. In the case of a failure a simple switch-over restores the OLTP system very quickly.

## V. Evaluation

We base our performance evaluation of the HyPer prototype on a benchmark we call TPC-CH to denote that it is a "merge" of the two standardized TPC benchmarks (www.tpc.org): The TPC-C benchmark was designed to evaluate OLTP database system performance and the TPC-H benchmark for analyzing OLAP query performance. Both benchmarks "simulate" a sales order processing (order entry, payment, delivery) system of a merchandising company. The benchmark constitutes the core functionality of such a commercial merchandiser like Amazon.

### *A. The TPC-CH-Benchmark*

The database schema of the TPC-CH benchmark is shown in Figure 8 as an Entity-Relationship-Diagram with cardinality indications of the entities and the (min,max)-notation to specify the cardinalities of the relationships. The cardinalities correspond to the initial state of the database when the TPC-C benchmark is started and increase (in particular, in number of Orders and Order-Lines) during the benchmark run. The initial database state can be scaled by increasing the number of Warehouses – thereby also increasing the number of Customers, Orders and Order-Lines, as each Customer has already submitted one Order with 10 Order-Line, on average. The original TPC-C schema, that we kept entirely unchanged, consists of the 9 relations in non-bold type face.

In addition, we included three relations (highlighted in bold type face) from the TPC-H benchmark in order to be able to formulate all 22 queries of this benchmark in a meaningful way. These relations are:

- *Supplier*: There are 10000 Suppliers that are referenced via a foreign key of the *Stock* relation. Thus, there is a fixed, randomly selected Supplier per Item/Warehouse combination.
- *Nation and Region*: These relations model the geographic location of Suppliers and Customers. There are 62 Nations and 5 Regions.

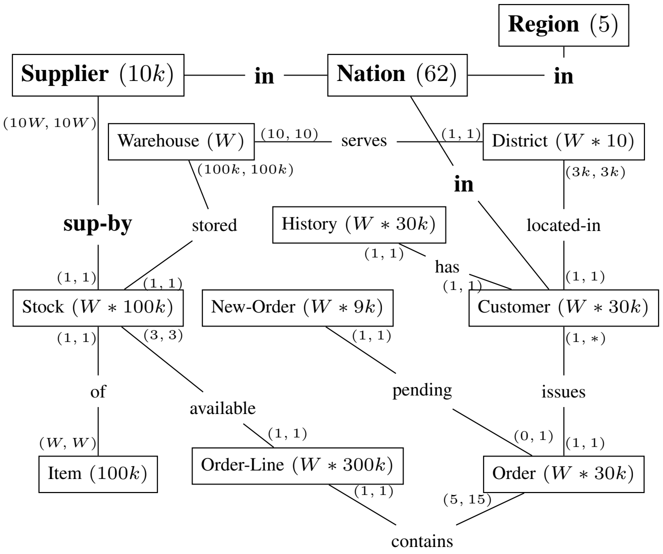

Fig. 8. Entity-Relationship-Diagram of the TPC-C&H Database

The TPC-C OLTP transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. All these transaction, including the read-only transaction *Order-Status* and *Stock-Level* were executed in serializable semantics via HyPer's OLTP workload queue.

In order to compare our results with the dedicated pure OLTP-system VoltDB we used their setup, which includes some modification of the benchmark. We cite their benchmark description [37]:

> *The VoltDB benchmark differs from the official TPC-C benchmark in two significant ways. Operationally, the VoltDB benchmark does not include any wait times, which we feel are no longer relevant. It also does not include fulfillment of orders submitted to one warehouse, with items from another warehouse (approximately 10% of the new order transactions in the official benchmark). Each benchmark was run with 12 warehouses (partitions) per node.*

The latter modification is only relevant for HyPer's multi-threaded OLTP processing on partitions which benefits from the exclusion of partition-crossing transactions. For the single-threaded process this simplification is irrelevant, i.e., there is no performance difference.

The transaction mix of the benchmark is such that the three update transactions (New-Order, Payment, and Delivery) reflect typical business procedures. The system maintains a balanced database state, i.e., every order is eventually paid and delivered. As the Delivery transaction processes ten orders in a batch it is scheduled only 1/10-th as frequently as the other two.

The performance of a system is usually specified in number of New-Order transactions that are processed – while, of course, all the other transactions have to be processed. To compare our results to other systems we will also report the aggregate count of all five transactions per second (tps).

### *B. OLAP Queries*

For the comprehensive OLTP&OLAP Benchmark we adapted the 22 queries of the TPC-H benchmark for the TPC-

|  | **HyPer** configurations |  |  |  |  |  | **MonetDB** | **VoltDB** |
|---|---|---|---|---|---|---|---|---|
|  | one query session (stream)<br>single threaded OLTP |  | 8 query sessions (streams)<br>single threaded OLTP |  | 3 query sessions (streams)<br>5 OLTP threads |  | no OLTP<br>1 query stream | no OLAP<br>only OLTP |
| Query No. | OLTP throughput | Query resp. times (ms) | OLTP throughput | Query resp. times (ms) | OLTP throughput | Query resp. times (ms) | Query resp. times (ms) | results from [18] |
| Q1 | new order: 56961 tps; total: **126576** tps | 67 | new order: 29359 tps; total: **65269** tps | 71 | new order: 171384 tps; total: **380868** tps | 71 | 63 | **55000** tps on single node; **300000** tps on 6 nodes |
| Q2 |  | 163 |  | 233 |  | 212 | 210 |  |
| Q3 |  | 66 |  | 78 |  | 73 | 75 |  |
| Q4 |  | 194 |  | 257 |  | 226 | 6003 |  |
| Q5 |  | 1276 |  | 1768 |  | 1564 | 5930 |  |
| Q6 |  | 9 |  | 19 |  | 17 | 123 |  |
| Q7 |  | 1151 |  | 1611 |  | 1466 | 1713 |  |
| Q8 |  | 399 |  | 680 |  | 593 | 172 |  |
| Q9 |  | 206 |  | 269 |  | 249 | 208 |  |
| Q10 |  | 1871 |  | 2490 |  | 2260 | 6209 |  |
| Q11 |  | 33 |  | 38 |  | 35 | 35 |  |
| Q12 |  | 156 |  | 195 |  | 170 | 192 |  |
| Q13 |  | 185 |  | 272 |  | 229 | 284 |  |
| Q14 |  | 122 |  | 210 |  | 156 | 722 |  |
| Q15 |  | 528 |  | 1002 |  | 792 | 533 |  |
| Q16 |  | 1353 |  | 1584 |  | 1500 | 3562 |  |
| Q17 |  | 159 |  | 171 |  | 168 | 342 |  |
| Q18 |  | 108 |  | 133 |  | 119 | 2505 |  |
| Q19 |  | 103 |  | 219 |  | 183 | 1698 |  |
| Q20 |  | 114 |  | 230 |  | 197 | 750 |  |
| Q21 |  | 46 |  | 50 |  | 50 | 329 |  |
| Q22 |  | 7 |  | 9 |  | 9 | 141 |  |

Fig. 9. Performance Comparison: HyPer OLTP&OLAP, MonetDB only OLAP, VoltDB only OLTP

CH schema of Figure 8. In the re-formulation we made sure that the queries retained their semantics (from a business point of view) and their syntactical structure. The OLAP queries do not benefit from database partitioning as they all require scanning the data across all partition boundaries. For example, Query Q5 of the TPC-H benchmark lists the revenue achieved through local suppliers and is re-formulated on our TPC-CH schema as follows:

```
select n_name, sum(ol_amount) as revenue
from Nation join Customer on ... join Order on ...
          join Order-Line on ... join Stock on ...
          join Supplier on ... join Region on ...
where su_nationkey=n_nationkey /* Cu and Su in the */
      and r_name='Europe'      /* same N of this R */
      and o_entry_d>= ...
group by n_name
order by revenue desc;
```

### C. Performance of Different HyPer Configurations

All benchmarks were carried out on a TPC-C-setup with 12 Warehouses. Thus, the initial database contained 360,000 Customers with 3.6 million order lines – totalling about 1 GB of net data. For reproducability reasons all query sessions were started (fork-ed) at the beginning of the benchmark (i.e., in the initial 12 Warehouse state) and the 22 queries were run in – altered, to exclude caching effects – sequence five times within each query session. Thus, each OLAP session/process was executing 110 queries sequentially. We report the median of each query's response times. These query sessions were executed in parallel to a single- or multi-threaded OLTP process – see Figure 9.

HyPer can be configured as a row store or as a column store. For OLTP we did not experience a significant performance difference; however, the OLAP query processing was significantly sped up by a column-wise storage scheme. Therefore, we only report the OLTP and OLAP performance of column store configurations.

The HyPer benchmark as well as the MonetDB query benchmark were run on a commodity server, with the following specifications:

- Dual Intel X5570 Quad-Core-CPU, 8MB Cache
- 64GB RAM
- 16 300GB SAS-HD (not used in benchmarks)
- Linux operating system RHEL 5.4
- Price: 13,886 Euros (discounted price for universities)

The OLTP performance of VoltDB we list for comparison was not measured on our hardware but extracted from the product overview brochure [18] and discussions on their web site [37]. The VoltDB benchmark was carried out on similar hardware (dual-quad Xeon CPU Dell R610 servers). The major difference was that the HyPer benchmark was run on a single server whereas VoltDB was scaled out to 6 nodes. In addition, the HyPer benchmark was carried out with redo logging to another storage server while VoltDB was run without any logging or replication.

HyPer's throughput results obtained on a single commodity server correspond to the published throughput results of VoltDB [18] on a 6-node cluster. As the VoltDB publications point out [18], these throughput numbers correspond to the very best published TPC-C results for high-scaled disk-based database configurations. The HyPer OLTP throughput numbers were even achieved while one, eight, or three parallel OLAP processes were continuously executing the OLAP queries in parallel to the OLTP workload (cf. Figure 9 from left to right).

The VoltDB system cannot support the parallel session(s) of OLAP queries. The performance results reveal that the left-most HyPer configuration under-untilizes the 8-core server while the middle configuration with 9 processes (1 OLTP, 8 OLAP) overloads the 8-core server. The lesson learned from this configuration is that the mission-critical OLTP process should be prioritized – which we did not in the experiment. The right-most configuation of 5 OLTP threads and 3 OLAP processes fully utilizes the server.

The query response times of HyPer in comparison with MonetDB reveal that the two query execution engine essentially have the same performance. For those outlier queries where the response times vastly vary we simply failed to ”tweak” MonetDB (e.g., by hints or query rewrites or query unnesting) to execute the same logical plan as HyPer. Furthermore, the out-of-the-box MonetDB installation we used does not appear to employ the advanced ”cracking” technique that horizontally partitions the columns on demand to optimize similar queries executed in sequence. MonetDB was run as a dedicated OLAP engine as we could not effectively execute the OLTP workload on MonetDB – the lack of indexes prevents any reasonable throughput on the TPC-C benchmark.

### D. Memory Consumption

In these experiments we monitored the memory consumption to assess the overhead imposed by the copy-on-write mechanism that maintains the consistency of the forked OLAP sessions. To isolate the effect of snapshot maintenance from the transient query execution's memory consumption, the OLAP processes remained idle while the OLTP process was executing and maintaining the snapshot via the implicit copy-on-write. The lower curve (A) of Figure 10 shows the memory footprint of the pure OLTP system without any OLAP snapshot. The memory footprint increases proportional to the volume of newly generated transactional data. The steps in this curve are due to resizing the data structures due to reaching the capacity of pre-allocated column vectors. The upper curve (B) shows the memory consumption of the system in which we forked an OLAP snapshot/process at the beginning of the OLTP transaction processing. We see that initially the OLTP process builds up its working set of replicated pages. The size of this working set – once created – does not increase much during the continuous benchmark run as the updates concern mostly newly generated data – therefore the curves A and B run largely parallel. The "zig-zag"-curve (C) shows the memory footprint of the system consisting of an OLTP process and an OLAP process that is initially forked and then at intervals of 500,000 transactions refreshed, i.e., terminated and reforked. The memory consumption of this configuration oscillates between the pure OLTP system and the configuration with one "long duration" OLAP snapshot. The spikes (above the other OLTP&OLAP configuration, B) are due to artifacts of the storage allocation for increased vector sizes and process forking overhead (the memory footprint was measured at the OS level in number of physically allocated pages).

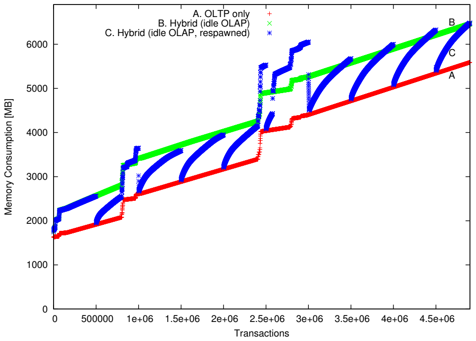

Fig. 10. Memory Consumption: (A) pure OLTP, (B) OLTP&one stable OLAP, (C) OLTP&continuously refreshed OLAP

### E. Scaling to Very Large Main Memory Sizes

Technological advances will soon allow main memory sizes of several TB capacity. For a default page size of 4 KB, a TB database has to manage a page table with 250 mil entries, summing up to 4 GB in size. For such ultra-large scale main memory databases the fork-execution can be optimized in several ways:

1) Use lazy page table copying as devised by [38]. Only the top levels of the hierarchically structured page table are eagerly copied whereas the lowest level with the so-called pte-entries is copied on demand.
2) Fork only the secondary server and while forking, buffer the incoming log records.
3) Increase the page size of segments of data objects that are most likely to be immutable.

Current operating systems and processors can accommodate different page sizes: For example, 4 KB as a default size and 2 MB for large segments. We propose to partition the data into two partitions: a so-called cold and a warm partition which are maintained in self-organized fashion. An update of a cold tuple will initiate the exchange of this tuple with an aged ("cooled down") tuple from the hot partition. The cold partition is stored on large 2MB-pages and the hot partition which incurs the replication costs due to snapshot maintenance is stored on default small 4KB-pages. The subsequent table demonstrates the costs of forking a main memory database of various sizes under the two different page sizes:

|  | small pages (4 KB) |  | large pages (2 MB) |  |
|---|---|---|---|---|
| DB size<br>in MB | fork<br>duration | . . . per<br>1 MB DB | fork<br>duration | . . . per<br>1 MB DB |
| 409.6 | 7ms | 17$\mu$s | 0.087ms | 0.21$\mu$s |
| 819.2 | 14ms | 17$\mu$s | 0.119ms | 0.15$\mu$s |
| 1638.4 | 28ms | 17$\mu$s | 0.165ms | 0.10$\mu$s |
| 4096 | 34ms | 8$\mu$s | 0.300ms | 0.07$\mu$s |
| 8192 | 69ms | 14$\mu$s | 0.529ms | 0.06$\mu$s |
| 16384 | 136ms | 8$\mu$s | 0.958ms | 0.06$\mu$s |
| 32768 | 271ms | 8$\mu$s | 1.863ms | 0.06$\mu$s |
| 40960 | 344ms | 8$\mu$s | 2.702ms | 0.06$\mu$s |

## VI. Summary

Our HyPer architecture is based on virtual memory supported snapshots on transactional data for multiple query sessions. Thereby, the two workloads – OLTP transactions and OLAP queries – are executed on the same data without interfering with each other. The snapshot maintenance and the high processing performance in terms of OLTP throughput and OLAP query response times is achieved via hardware supported copy on demand (= write) to preserve snapshot consistency. The detection of shared pages that need replication is done efficiently by the OS with Memory Management Unit (MMU) assistance. The concurrent transactional workload and the BI query processing use multi core architectures effectively without concurrency interference – as they are separated via the VM snapshot.

In this way, HyPer achieves the query performance of OLAP-centric systems such as SAP's TREX and MonetDB and, in parallel on the same system, retains the high transaction throughput of OLTP-centric systems, such as Oracles's TimesTen, SAP's P*Time, or VoltDB's H-Store. As the OLAP snapshot can be as current as desired by forking a new OLAP session we are convinced that HyPer's virtual memory snapshot approach is a promising architecture for *real-time business intelligence* systems.

While the current HyPer prototype is a single server scale-up system, the VM snapshotting mechanism is orthogonal to a distributed architecture that scales out across a compute cluster – as we will demonstrate in the future. The snapshot mechanism could also be used in a data warehouse configuration where the transaction workload queues corresponds to a continuous refresh stream emanating from one or several OLTP systems. Then, the "data-owning" process corresponds to the installer of these updates while the OLAP queries can be executed in parallel against consistent snapshots.

## Acknowledgment

We thank Florian Funke and Michael Seibold for helping with the performance evaluation. We acknowledge the many colleagues with whom we discussed HyPer's virtual memory snapshot architecture.

## References

[1] J. Doppelhammer, T. Höppler, A. Kemper, and D. Kossmann, "Database performance in the real world - TPC-D and SAP R/3," in *SIGMOD*, 1997.

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

[3] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for RAMClouds: scalable high-performance storage entirely in DRAM," *Operating Systems Review*, vol. 43, no. 4, 2009.

[4] Intel, "Tera-scale computing research program," 2010, http://techresearch.intel.com/articles/Tera-Scale/1421.htm.

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

[6] 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*, vol. 1, no. 2, 2008.

[7] S. Chaudhuri and G. Weikum, "Rethinking database system architecture: Towards a self-tuning risc-style database system," in *VLDB*, 2000.

[8] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *VLDB J.*, vol. 19, no. 1, 2010.

[9] Oracle, *Extreme Performance Using Oracle TimesTen In-Memory Database*, http://www.oracle.com/technology/products/timesten/pdf/wp/wp_timesten_tech.pdf, July 2009.

[10] S. K. Cha and C. Song, "P*TIME: Highly scalable OLTP DBMS for managing update-intensive stream workload," in *VLDB*, 2004.

[11] A.-P. Liedes and A. Wolski, "Siren: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases," in *ICDE*, 2006.

[12] R. A. Lorie, "Physical integrity in a large segmented database," *TODS*, vol. 2, no. 1, 1977.

[13] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture evolution: Mammals flourished long before dinosaurs became extinct," *PVLDB*, vol. 2, no. 2, 2009.

[14] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *SIGMOD*, 2009.

[15] J. Krüger, M. Grund, C. Tinnefeld, H. Plattner, A. Zeier, and F. Faerber, "Optimizing write performance for read optimized databases," in *DASFAA*, 2010.

[16] A. Whitney, D. Shasha, and S. Apter, "High volume transaction processing without concurrency control, two phase commit, SQL or C," Intl. Workshop on High Performance Transaction Systems, 1997.

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

[18] VoltDB, "Overview," http://www.voltdb.com/_pdf/VoltDBOverview.pdf, March 2010.

[19] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," in *VLDB*, 2010.

[20] Ö. Ulusoy and A. P. Buchmann, "A real-time concurrency control protocol for main-memory database systems," *Inf. Syst.*, vol. 23, no. 2, 1998.

[21] E. P. C. Jones, D. J. Abadi, and S. Madden, "Low overhead concurrency control for partitioned main memory databases," in *SIGMOD*, 2010.

[22] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold, "A comparison of flexible schemas for software as a service," in *SIGMOD*, 2009.

[23] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann, "Predictable performance for unpredictable workloads," *PVLDB*, vol. 2, no. 1, 2009.

[24] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki, "Data-oriented transaction execution," in *VLDB*, 2010.

[25] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki, "Aether: A scalable approach to logging," in *VLDB*, 2010.

[26] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle, "Constant-time query processing," in *ICDE*, 2008.

[27] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman, "Main-memory scan sharing for multi-core cpus," *PVLDB*, vol. 1, no. 1, 2008.

[28] A. Kemper, D. Kossmann, and F. Matthes, "SAP R/3: A database application system (tutorial)," in *SIGMOD*, 1998.

[29] S. Finkelstein, D. Jacobs, and R. Brendle, "Principles for inconsistency," in *CIDR*, 2009.

[30] S. Bailey, "Us patent 7389308b2: Shadow paging," 17. Juni 2008, filed: 30. Mai 2004, granted to Microsoft.

[31] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[32] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "A critique of ANSI SQL isolation levels," in *SIGMOD*, 1995.

[33] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," *TODS*, vol. 34, no. 4, 2009.

[34] T. Neumann and G. Weikum, "x-RDF-3X: fast querying, high update rates, and consistency for RDF databases," in *VLDB*, 2010.

[35] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[36] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," in *SIGMOD*, 1984.

[37] VoltDB, *VoltDB TPC-C-like Benchmark Comparison-Benchmark Description*, https://community.voltdb.com/node/134, May 2010.

[38] D. McCracken, "Sharing page tables in the Linux kernel," in *Proceedings of the Linux Symposium*. Ottawa, CA: IBM Linux Technology Center, July 23rd, 2003, http://www.kernel.org/doc/ols/2003/ols2003-pages-315-320.pdf.