HyPer: HYbrid OLTP&OLAP High PERformance Database System

Alfons Kemper, Thomas Neumann

TUM-I1010
Mai 10

HyPer: HYbrid OLTP&OLAP High PERformance Database System*

Alfons Kemper and Thomas Neumann
Lehrstuhl für Informatik III: Datenbanksysteme
Fakultät für Informatik
Technische Universität München

May 19, 2010

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 transalation, caching, copy on update) yields both at the same time: unprecedented high transaction rates as high as several 100000 per second and ultra-low OLAP query response times of as low as 10 ms – all on a commodity desktop server. Even the creation of a fresh, transaction-consistent snapshot can be achieved in 10 ms.

1 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 highest-scaled systems can process more than 100.000 such sales transactions per second.

*Patent pending – applied for by Technische Universität München / Bayerische Patentallianz.

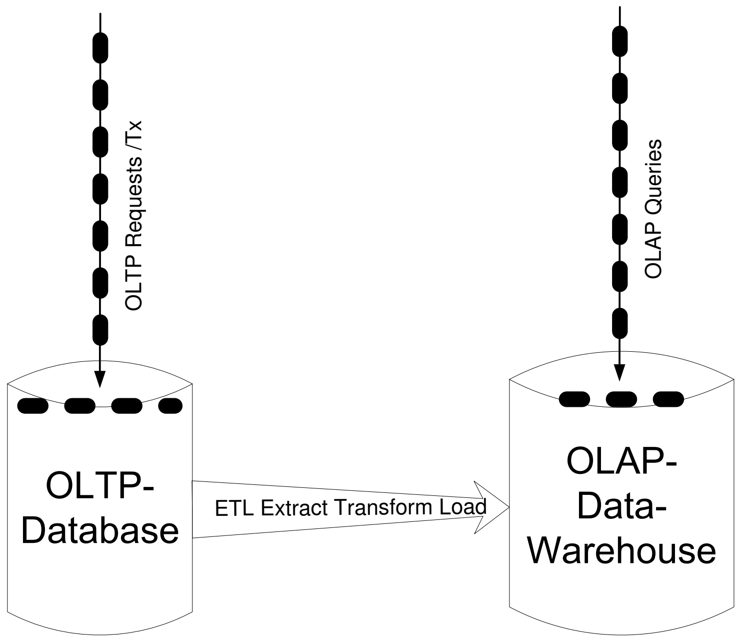

Figure 1: State of the Art: Separate OLTP-DB and OLAP-DW

About two decades ago a new usage of database systems evolved: Business Intelligence (BI). The 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 the aggregated sales statistics grouped by geographical regions, or by product categories, or by customer classifications, etc. Initial attempts – such as SAP's EIS project – 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 exemplified in Figure 1 was devised. Here, the transaction processing is carried out on a dedicated OLTP database system. In addition, a separate Data Warehouse system is installed for the 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 process exhibits several inherent drawbacks:

stale data As the ETL process can only be executed periodically, the data warehouse state does not reflect the latest business transactions. Therefore, business analysts have to base their decisions on stale (outdated) data.

redundancy Obviously, the usage of two systems incurs the cost of maintaining two redundant copies of the data. On the positive side, the redundancy allows to model the data in an application specific way: in normalized tables for OLTP-processing and as a star-schema for OLAP queries.

high investments Maintaining two separate systems incurs an economical penalty as investments for two systems (hardware, software, etc) and maintenance costs for two systems and the complex ETL process have to be taken into account.

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

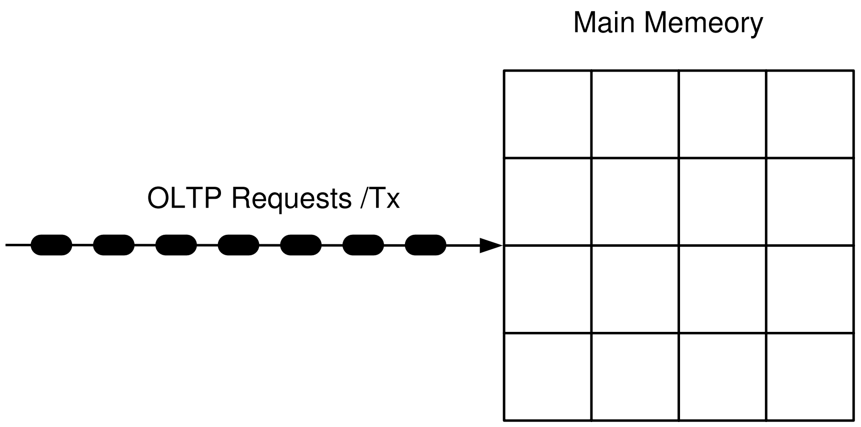

Figure 2: Main Memory OLTP Database Architecture

enterprise resource planning systems [25]. To achieve this goal several architectural changes of current database architectures have to be undertaken. First, the separation of OLTP database and OLAP data warehouse system has to be abandoned. The integration of these two very different workloads on the same system necessitates further 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. We sketch the order processing data volume of this largest online market place 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 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. [24] who proclaim the so-called RAM-cloud as a main-memory storage device for the largest Internet software applications. Furthermore, 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 a TB of main memory as part of its so-called Tera Scale initiative (`http://techresearch.intel.com/articles/Tera-Scale/1826.htm`). 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 as laid out in Figure 2. We follow the single-threading approach first advocated in [15] whereby all OLTP transactions are executed sequentially.[1]. This architecture obviates the

[1]The single threading execution is relaxed to allow a single-threaded execution per partition in Sec-

need for costly locking and latching of data objects or index structures as the only one update transaction "owns" the entire database – or the 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 [17] 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

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

This challenge is sketched in Figure 3. 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 and fast recovery. In parallel to the OLTP processing, HyPer 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) accomplishes both in the same system and at the same time: unprecedented high transaction rates as high as several 100000 per second and ultra-low OLAP query response times of as low as 10 to 30 ms – all on a commodity desktop server for less than 4000 Euro. Even the creation of a fresh, transaction-consistent snapshot can be achieved in 10 ms.

2 Related Work/Systems

- HyPer is a new RISC-style database systems like RDF-3X [22] (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.
- HyPer's partitioning technique is particularly beneficial for multi-tenancy database applications [1, 2].

tion 3.4

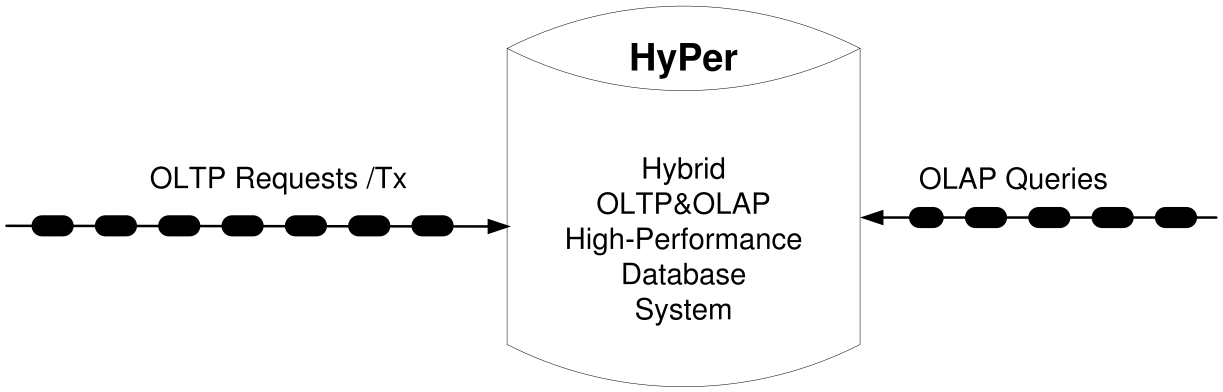

Figure 3: Hybrid OLTP&OLAP Database Architecture

- SolidDB of Solid Information Technology is a main memory DB developed in Helsinki. In the mean time IBM took over this company. Within this project the tuple level [21] 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.
- Shadow Paging [3]
  - Shadow paging has been around in the database literature [14] for decades – starting in the Eighties.
  - For disk based database systems shadowing was not really successful due to its destoying 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 a virtual memory supported shadow paging where scan performance is not hurt by the 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.
- The authors of H-Store [17, 15] deserve the credit for analyzing the overhead imposed by various traditional database management features (buffer management, logging, locking, etc.) and proving the feasibility of a main memory database systems that processes transactions sequentially without synchronization overhead.
- VoltDB [26, 16]: The commercialization of H-Store was first presented at the HPTS workshop end of 2009.
- TimesTen[23]: This is Oracle's main memory database system for OLTP processing. It is typically used as a "front" cache for the Oracle mainstream database system.
- MonetDB: This is the most influential database research project on column store storage schemes for in memory data. An overview of the system can be found in the summary paper [7] resented on the occasion of receiving the 10 year test of time award of the VLDB conference.

- SAP's Main Memory Databases
  - OLTP: P*TIME / Transact in Memory [10]
    - founded by Sang Kyun Cha, now a professor at Seoul University
    - acquired by SAP in 2005
  - OLAP: TREX [6]
    - This is SAP's most prominent database project
    - underlying SAP's business intelligence functionality
    - According to Hasso Plattners key note at SIGMOD 2009 [25] 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
- HP Neoview: Neoview is HP's parallel database system, that is augmented for real time business intelligence. [20] reports on our joint project on managing mixed OLTP&OLAP workloads in Neoview.
- RAMcloud [24] is a research and development project at Stanford University which receives a lot of (academic and industrial) attention. Their goal is to develop a large scale cluster of machines with cooperative main memory storage that can service any object access within 10 microseconds.

3 System Architecture

The HyPer architecture was devised such that OLTP transactions and OLAP queries can be performed on the same main memory resident database. 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.

3.1 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 [15] whereby all OLTP transactions are executed sequentially. This architecture obviates the need for costly locking and latching of data objects as the only one update transaction "owns" the entire database. Obviously, this serial execution approach is only viable for a pure main memory

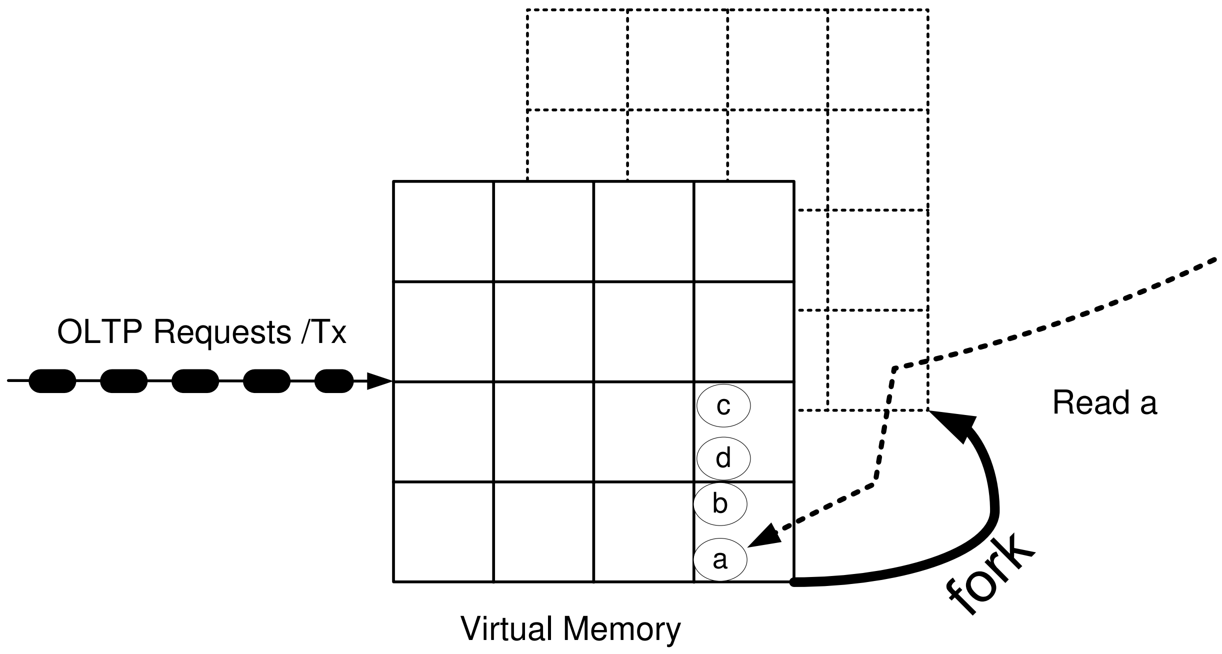

Figure 4: Forking a new Snapshot

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 microseconds. This translates to throughputs in the order of tens of thousands per second, much more than even large scale business applications require – as we analyzed in the Introduction.

The serial execution of OLTP transactions is exemplified in Figure 4 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 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 [19, 12] reveals that these kinds of interactions occur outside the database context in the application servers, anyway.

3.2 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

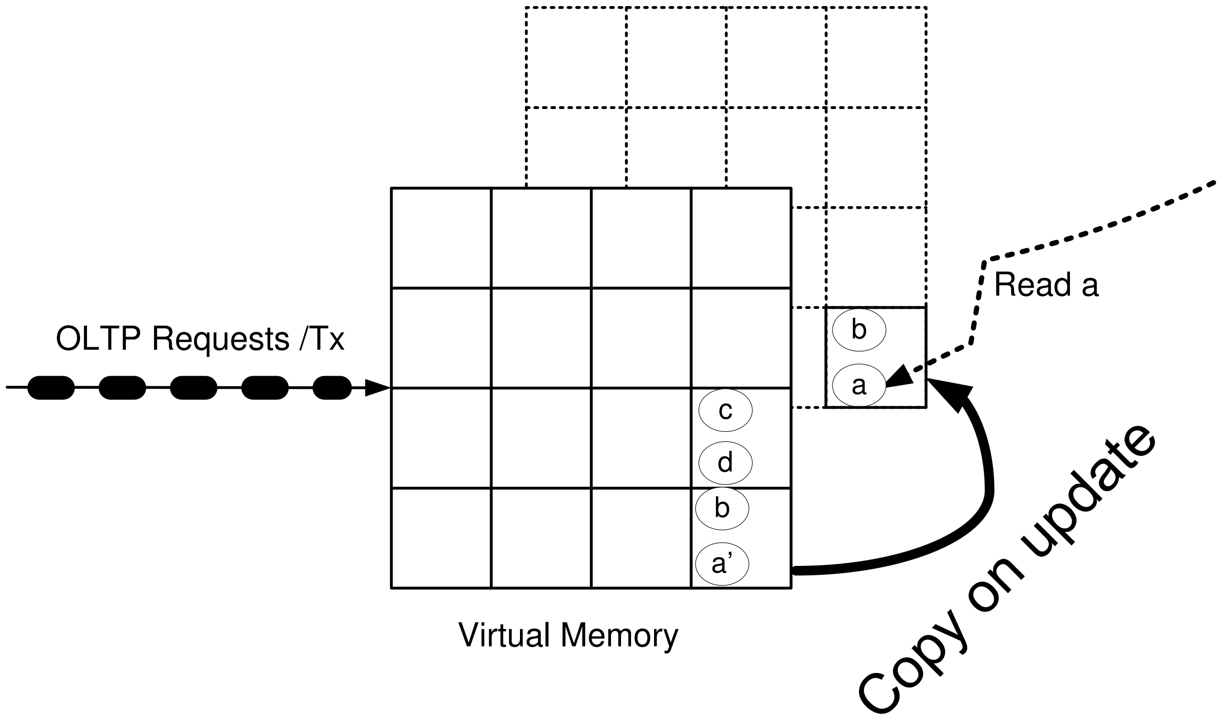

Figure 5: Copy on Update to Preserve Consistent Snapshot

could have completed. To achieve our goal to architect 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 4.6 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 4 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 in Figure 6.

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 in Figure 6. 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-

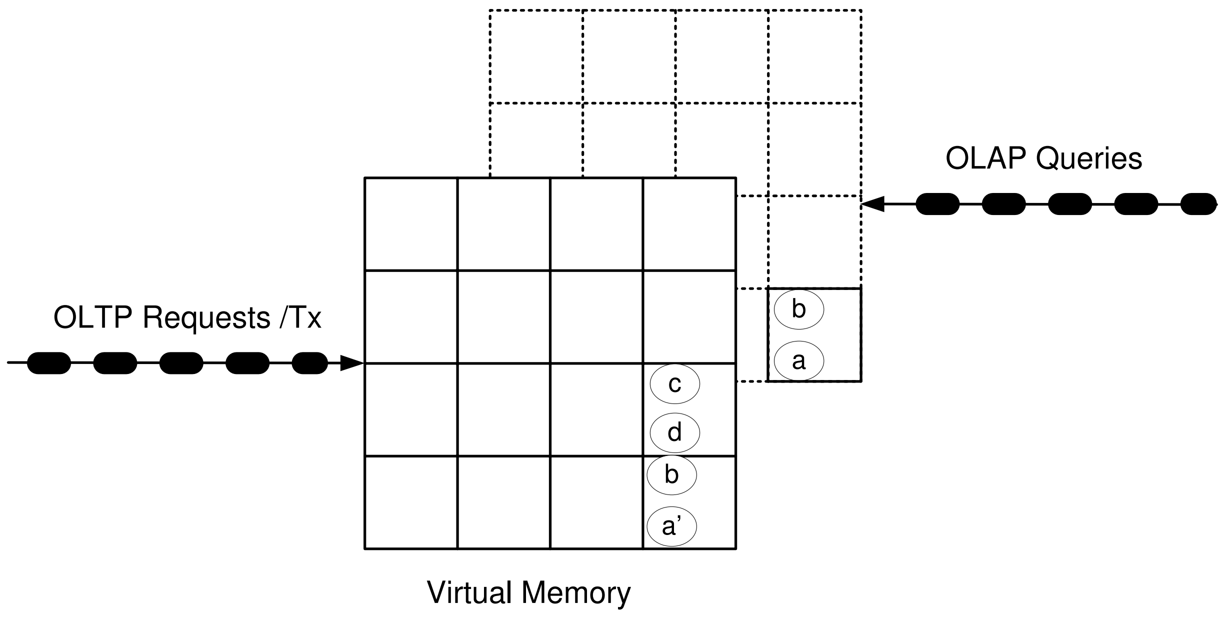

Figure 6: Using the Snapshot for Querying

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 8).

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) *main* database pages. Every "now and then" the Delta is merged with the OLAP database by fork-ing a new process for an up-to-date OLAP session. Thereby, the Delta is conceptually re-integrated into the *main* database. This view is illustrated in Figure 7. Unlike any software solution for merging the two parts, main and Delta, our hardware-supported virtual memory merge (fork) can be achieved very efficiently in about 10 ms (almost independent of the size of the two parts).

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 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 [3] or shadow individual objects as proposed in [21].

Our snapshots incur storage overhead proportional to the number of updated pages by the parent (i.e., the OLTP request executing) process. It replicates the delta (corre-

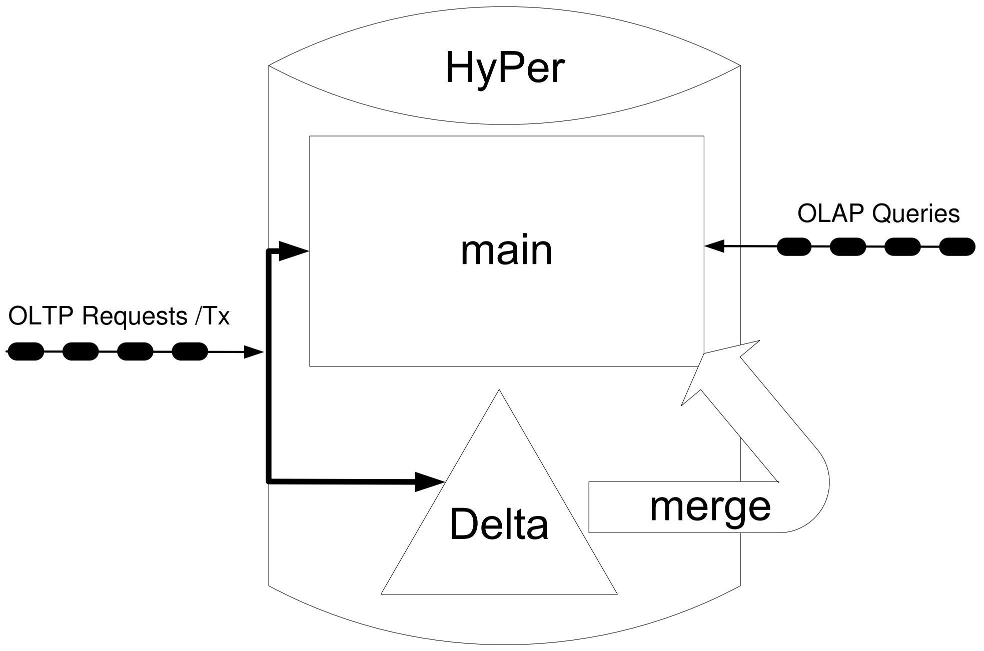

Figure 7: HyPer-Snapshot Creation Viewed as a Delta Merge

sponding 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 (almost) 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.

3.3 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 8. Here we sketch the one and only OLTP processes current database state (the front panel) and three

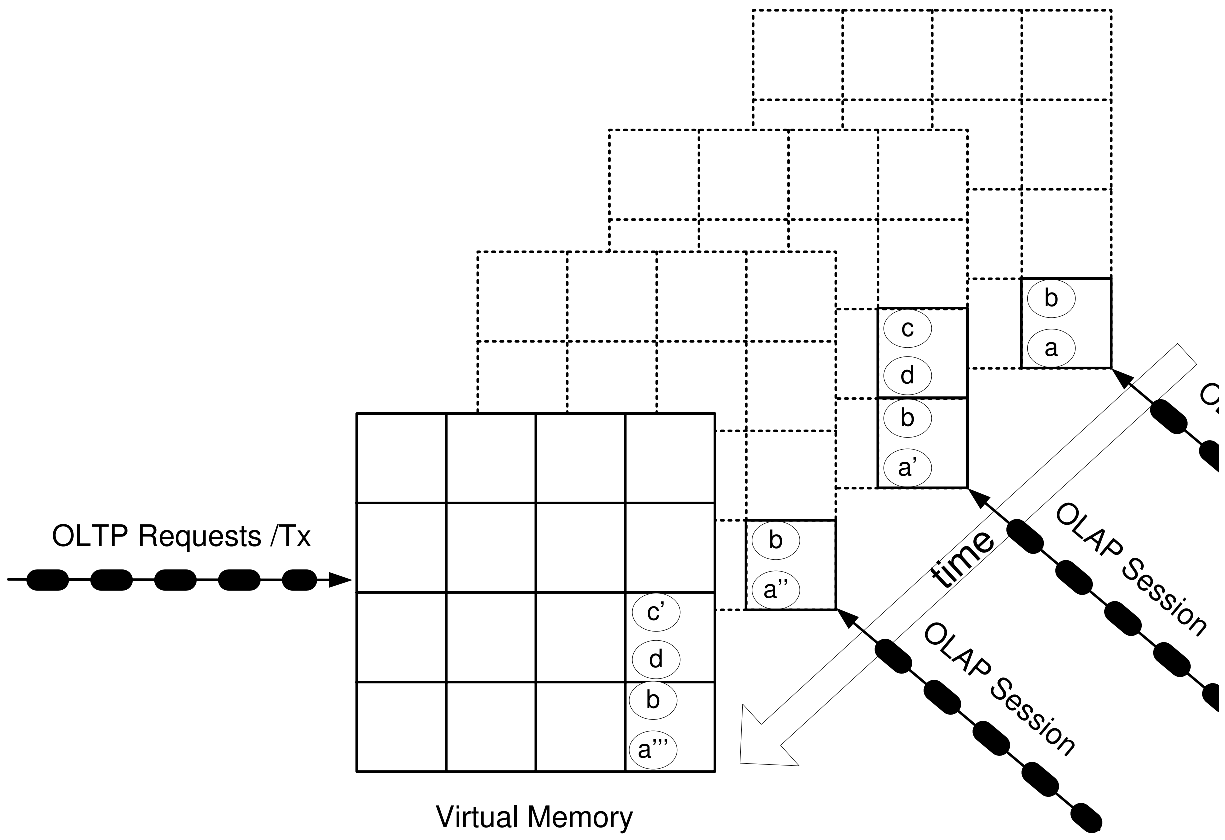

Figure 8: Multiple OLAP Sessions at Different Points in Time

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.

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 actual 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 processes address space.

3.4 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 also possible for the OLTP process, as we will describe here.

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 [1, 2]. 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 in Figure 9. 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. The figure sketches this as 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 exclusive access to the system – just as in our initial purely sequential approach. This way updating the shared data is quite expensive – however, since it is read-mostly updates are rare events on this data. The synchronization aspects are further detailed in Section 4.3.

The 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 4.6 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.

4 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 [5] (Section 5.5.). In this model *updaters* (in our terminology OLTP transactions) are fully serializable

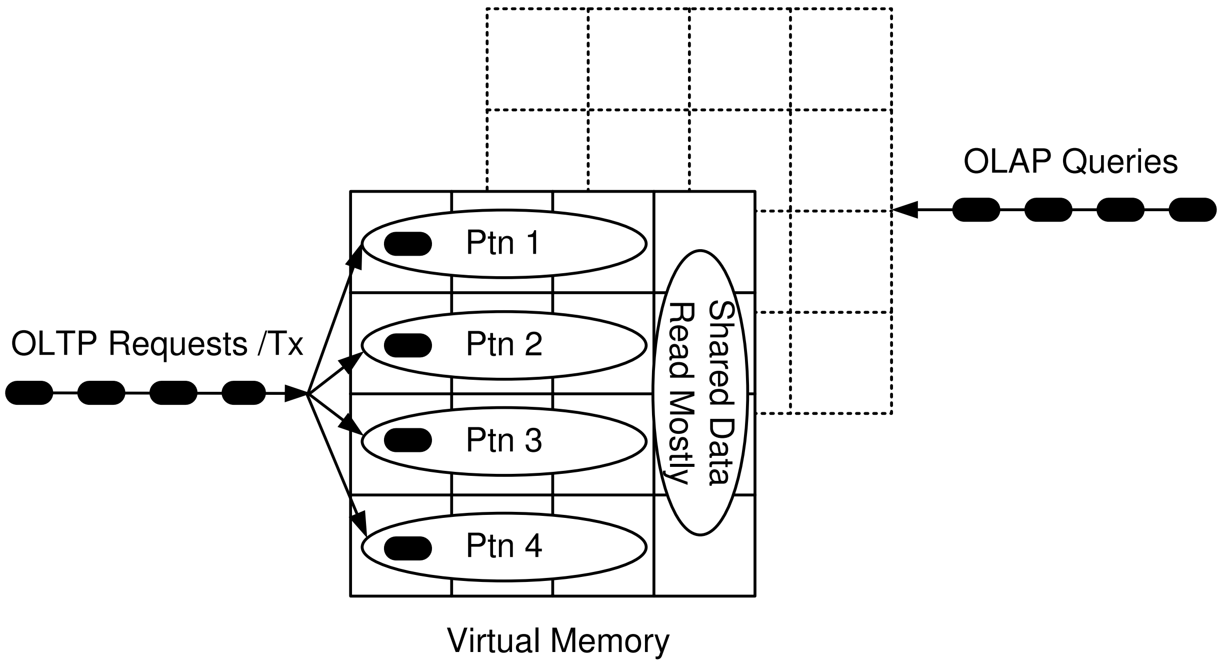

Figure 9: Multi-Threaded OLTP Processing on Partitioned Data

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 and full serializability. A variation of the multiversion synchronization is called snapshot isolation and was first described in [4]. It currently gains renewed interest in the database research community – see, e.g., [8]. Herein, the snapshot synchronization is not constrained to read-only queries but also to the read requests in update transactions. Snapshot isolation is, for example employed in Sleepy Cat (an open source DBMS acquired by Oracle). Also Oracle's flagship commercial database systems employs a variant of snapshot isolation. Oracle's commercial success can partly be attributed to this relaxed concurrency control scheme resembling snapshot isolation because of its superior performance in comparison to strict two-phase locking schemes.

4.1 Snapshot Isolation of OLAP Query Sessions

In snapshot isolation a transaction 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 [3]: 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.

4.2 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 in Figure 10. 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. To maintain this transfer speed the storage server has to employ several (around 10) disks for a corresponding aggregated bandwidth.

4.3 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 3.4) 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.

Transactions of the latter class of partition crossing transactions should be very rare as updates to shared data seldom occur and the partitioning is derived such that transactions usually operates 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.

The HyPer system admits at most one partition constrained transactions 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 transactions, 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

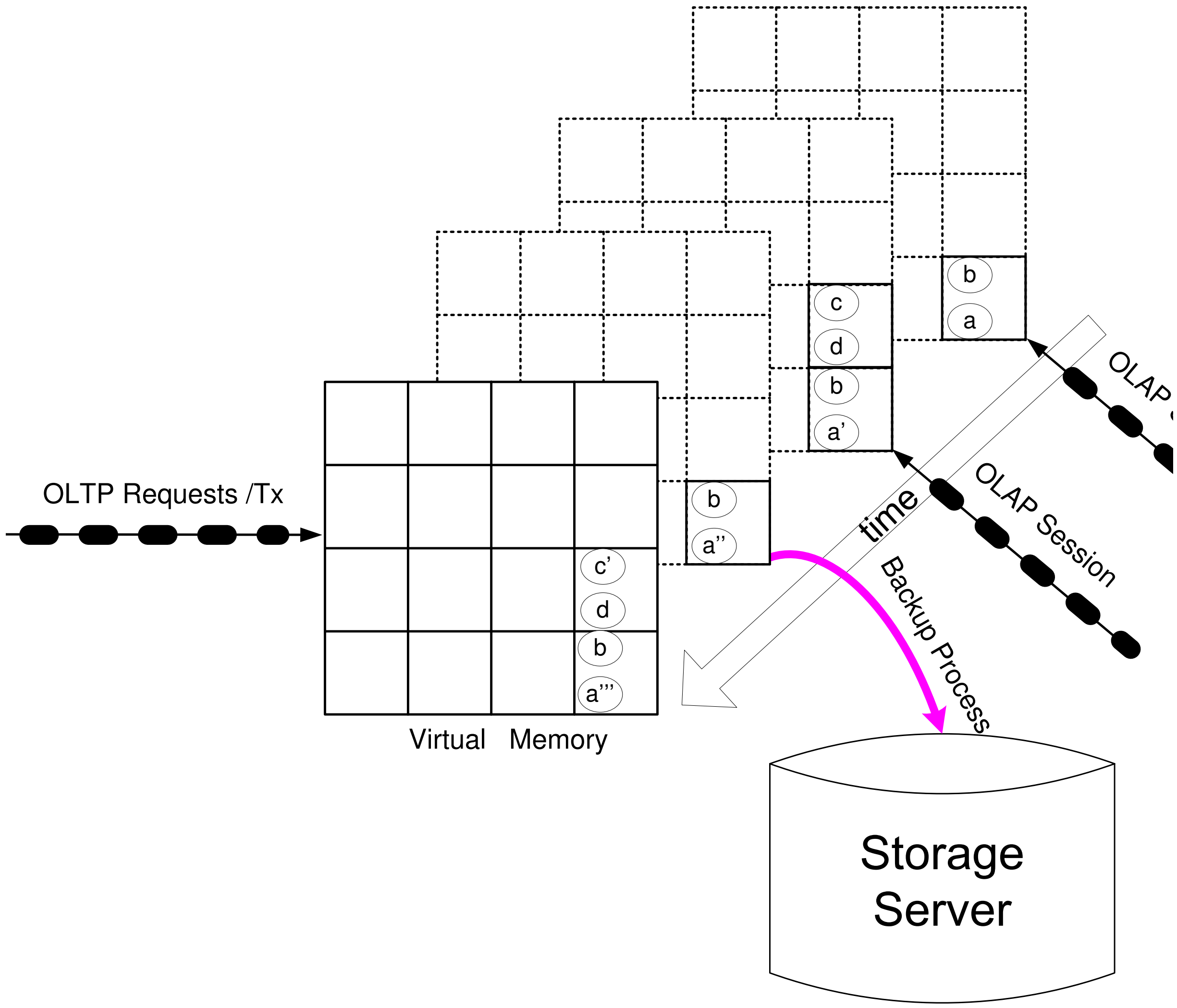

Figure 10: Consistent Snapshot Backup Archive

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.

4.4 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 ovals emanating from the serial transaction stream leading to the non-volatile Redo-Log storage device in Figure 11. We employ *logical* redo logging [13] 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 10). 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 w.r.t. 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 excutes the same transactions as the primary server. In case of a primary server failure the transaction processing can be 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 less loaded as it needs not execute any read-only OLTP transactions and, therefore, 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 12. 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. [11] and in the textbook by Härder and Rahm [14] are possible

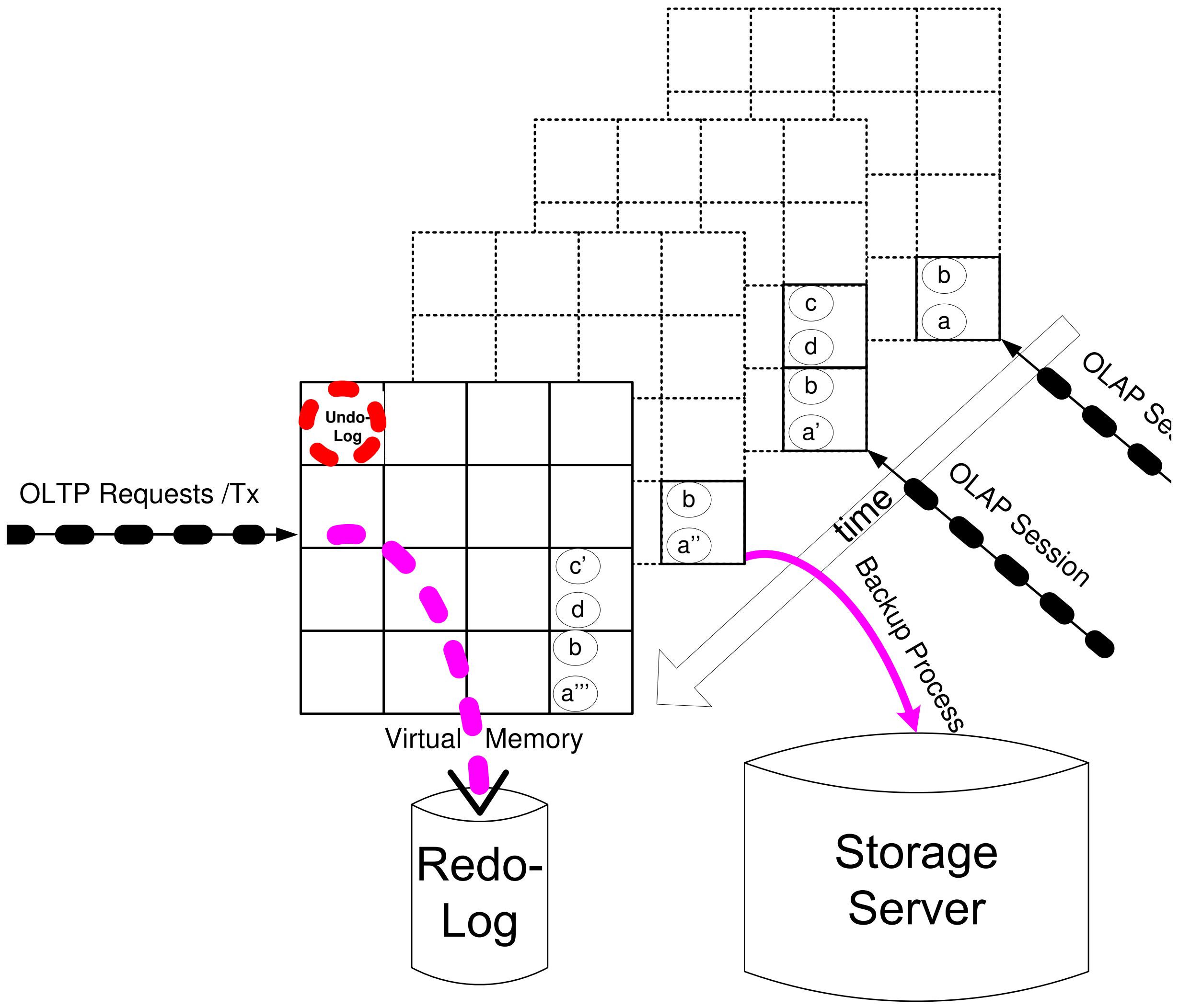

Figure 11: Durable Redo and Volatile Undo Logging

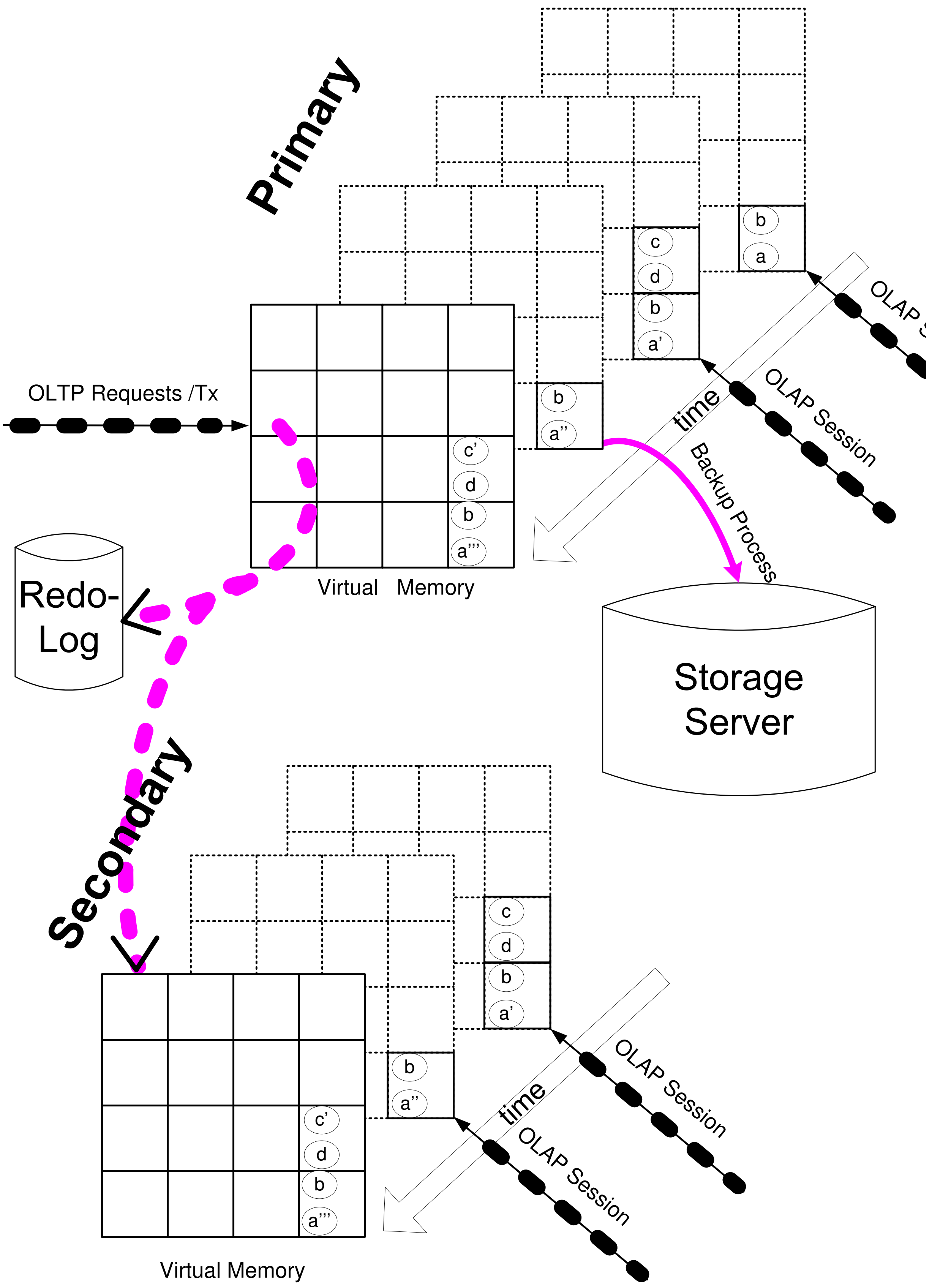

Figure 12: Secondary Server: Stand-By for OLTP and Active for OLAP

- Group commit or
- Asynchronous commit.

Group commit is, for example, configurable in DB2. 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 acknowledgement 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. Once the log buffer is flushed for the group of transactions, their commit is acknowledged to the client.

Another, less safe, method can be used 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.

4.5 Atomicity

The atomicity of transactions requires to be able to eliminate any effects of a failed transaction from the database. We only have to consider explicitly aborted transactions, called the R1-recovery [18]. 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 10). 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 11 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).

4.6 Cleaning Action Consistent Snapshots

The 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 the 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 – including all transactions that were completed up to this point in time.

This transformation of an action consistent snapshot into a transaction consistent one is applied to create an OLAP session's snapshot as well as for preparing a snapshot for archiving the database (cf. Figure 10).

4.7 Recovery after a System Failure

During recovery we can start out with the youngest fully written archive, which is restored in 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, one in active mode and the other one performing the same transactions, e.g., via redo log "sniffing" – as sketched in Figure 12. In the case of a failure a simple switch-over restores the system.

The recovery process is sketched in Figure 13.

5 Evaluation

We base our performance evaluation of the HyPer prototype on the standardized TPC-C benchmark. This benchmark "simulates" 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 or Würth or Otto Versand.

5.1 The TPC-C-Benchmark

The TPC-C benchmark is a standardized benchmark for OLTP applications. The benchmark specification and the supervision of the benchmark execution is managed by the Transaction Processing Council organization (`www.tpc.org`).

The database schema of the TPC-C benchmark is shown in Figure 14 as an Entity-Relationship-Diagram with cardinality indications of the entities and the (min,max)-notation to specify the cardinalities of the relationships. The schema consists of 9 relations:

- *Warehouse*: There are $W \geq 1$ warehouses, each represented by a separate tuple. Our experiments were carried out on a database with 5 warehouses.
- *District*: There are 10 districts per warehouse, whose customers are primarily served by the corresponding warehouse.
- *Customer*: Per district there are 3000 customers, i.e., 30000 per Warehouse and $W * 30000$ in total.
- *Order*: In the initial state every customer has already submitted one order. The number of orders is obviously increasing during the benchmark run and pending orders are continuously processed by the Delivery transaction.

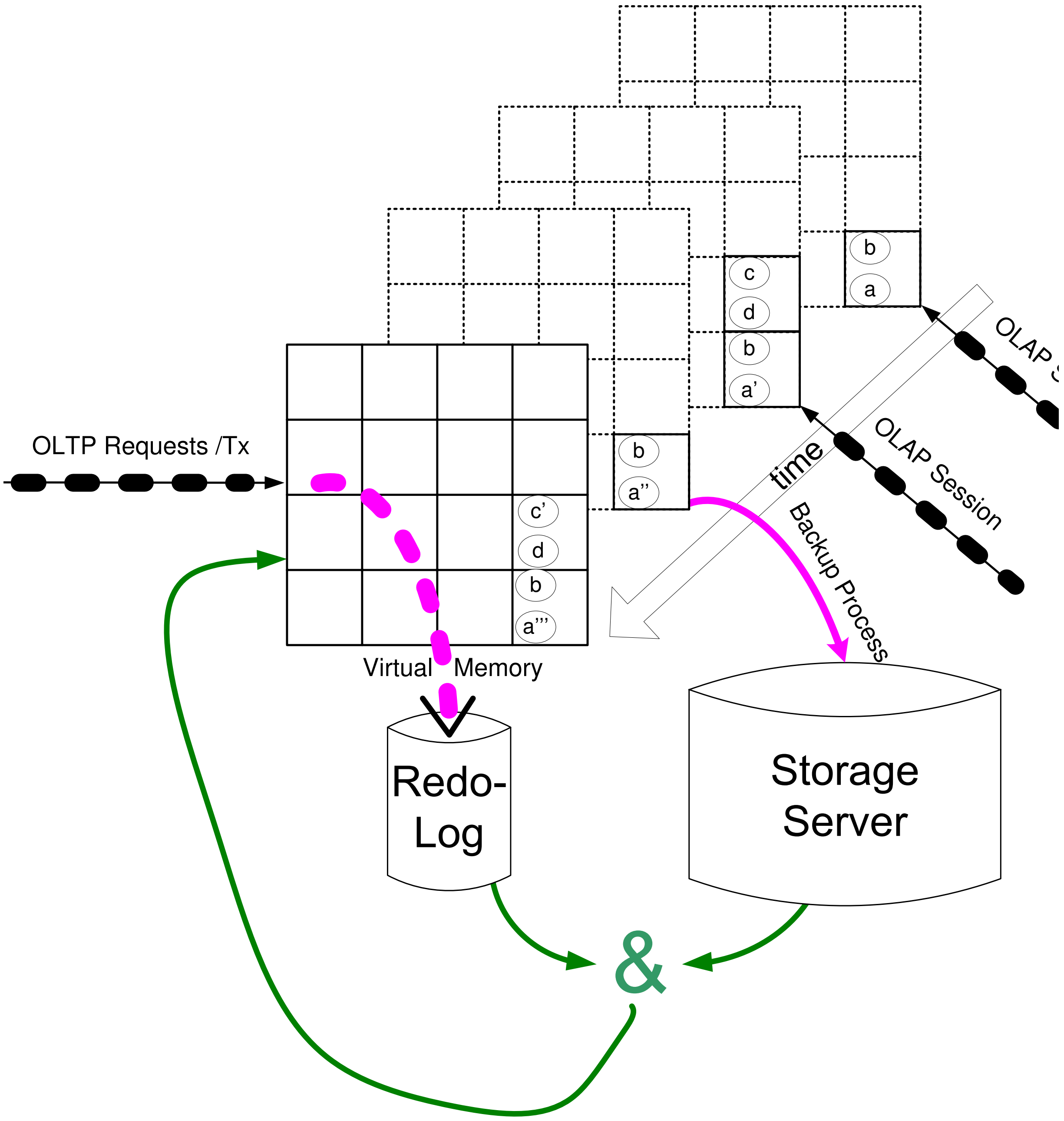

Figure 13: Recovery Process: (1) Load Archived Backup and (2) Apply Redo Log

- *New-Order*: A newly arriving order is not only inserted into the *Order*, but also into the *New-Order* relation. More precisely, a foreign key is inserted here and deleted after the order has been delivered.

- *Order-Line*: Every order has between 5 and 15 (average 10) order lines specifying the ordered items (products). Thus, the initial 5 Warehouse-database has 1.5 mio order lines preloaded.

- *Stock*: In this relation the stocked items of the different warehouses are modelled. Thus, per (Warehouse,Item)-tuple there is one entry in the relation *Stock* – i.e., $W * 100k$ tuples in total. An order line is checked against the stock level of a warehouse, modelled by the relationship *available*.

- *Item*: This relation contains one tuple for each of the 100000 products that can be ordered. The relation *Item* is immutable, that is, it is never updated during the benchmark runs. Furthermore, it remains invariant at 100000 entries against scaling the database by increasing the number of warehouses.

- *History*: This relation contains aggregated information about the order history of each customer.

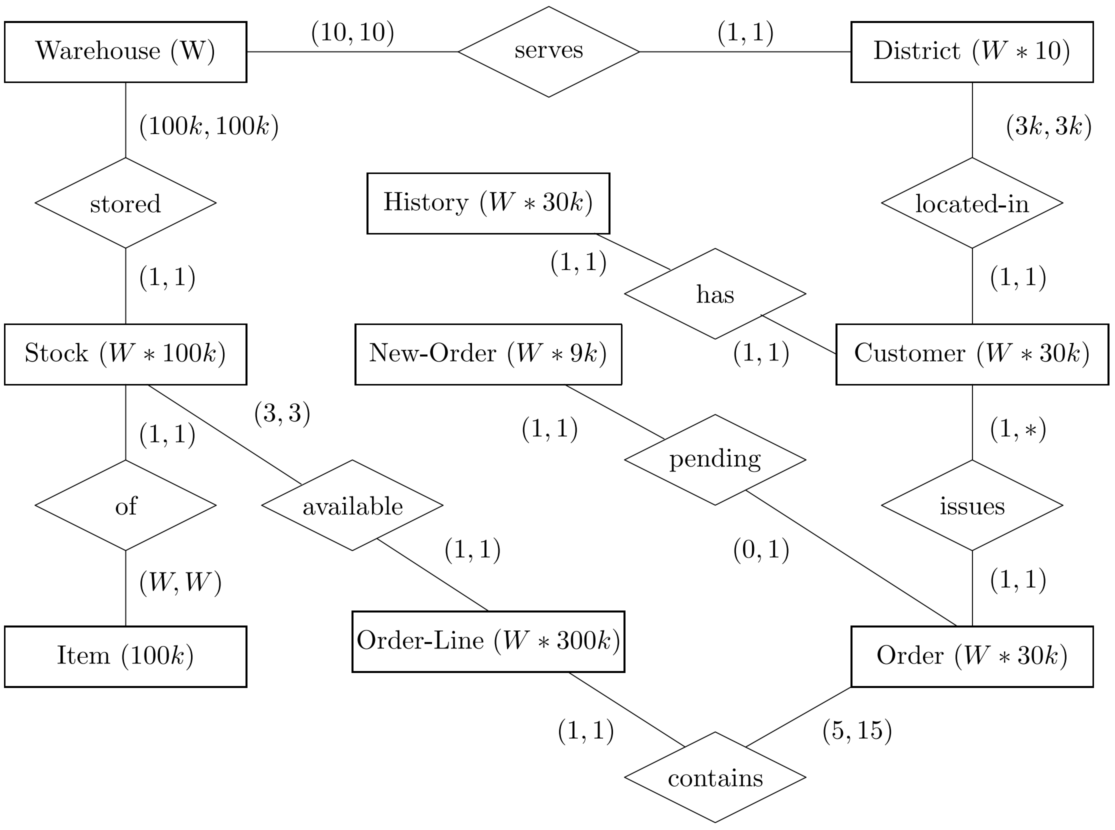

Figure 14: Entity-Relationship-Diagram of the TPC-C Database

The TPC-C-benchmark consists of five transactions that simulate the order processing in a typical merchandising company:

1. *New-Order*: In this transaction a new incoming order is processed, that consists of 5 to 15 order lines. The availability of each order line is checked in the *Stock* relation. For 1% of the New-Orders the last order line is invalid as it contains a non-existing Item number. In this case the entire order processing has to be **undo**ne – using the undo recovery scheme.

2. *Payment*: In this transaction the payment of a customer is processed. The paying customer is either identified by primary key or by name, in which case one of the several customers has to be selected. In addition to updating the customer entry, also aggregated statistics for the corresponding Warehouse and District are updated.

3. *Order-Status*: This is a read-only transaction (query), in which the last order of a particular customer is retrieved. Even though it is a query, we process it as an OLTP transaction to ascertain that the most current transaction consistent state is exposed.

4. *Delivery*: In this (most complex) transaction 10 pending orders of the New-Order relation are selected and processed. The corresponding orders are deleted from the *New-Order* relation.

5. *Stock-Level*: This read-only transaction controls the availability of the most recently ordered items. Again, we process it as an OLTP transaction.

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, that is, 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 frequent as the other two.

The performance of a system is specified in number of New-Order transactions that are processed – while, of course, all the other transactions have to be processed within certain time limits as well.

The benchmark reports two performance criteria:

- *tpmC*: The throughput of *New-Order*-transactions per minute. We will report transactions per second (TPS) instead.

- *Price/Performance*: For this performance parameter the total systems price (hardware, software, and software maintenance for five years) is related to the performance (i.e., tpmC). This performance parameter specifies the cost of one transaction per minute in $.

Obviously, there is a trade-off between ultra high performance (for very large scale companies) and cost effectiveness (for medium sized installations). Currently two of the best performing systems of either category achieve

- 100000 New-Order transactions per minute (i.e., ca 2000 transactions per second – tps) at a system price of ca 65000 US Dollar (that is, about 0.7 Dollar per transaction per minute).

- 6000000 New-Order transactions per minute (i.e., 100000 tps) at a price of ca. 17 Mio. US Dollar (i.e., around 3 Dollar per transaction per minute in the price/performance ratio).

Many more benchmark results can be obtained via `www.tpc.org`.

5.2 OLAP Query

Since HyPer is designed as a hybrid OLTP&OLAP database system we also devised a set of OLAP queries to augment the TPC-C benchmark. For our initial measurements we use just one, albeit comprehensive OLAP query that scans the bulk of the database (the orders and order lines). In this query for a specified District in a specified Warehouse the *top-10 Customers* w.r.t. their generated revenue are determined. The SQL code is sketched below:

```
Select o_c_id, sum(ol_amount)
From order join orderline on ol_o_id=o_id
Where o_w_id = $1 and
      o_d_id = $2 and
      ol_w_id = $1 and
      ol_d_id = $2
Group by o_c_id
Order by sum(ol_amount) desc
Limit 10
```

5.3 Performance of Different HyPer Configurations

All benchmarks were carried out on a TPC-C-setup with 5 Warehouses. Thus, the initial database contained 150000 Customers with 1.5 million order lines – totalling ca. 500 MB of net data. The query session was started (fork-ed) at the beginning of the benchmark and the same top-10 customer query was run successively in parallel to the OLTP transactions on the snapshot of the initial state of the benchmark. The OLTP throughput was derived from measuring the running time for 100000 transactions. The query response times were averaged from 100 executions of the *top 10 Customers* query which were sequentially executed in parallel to the "full-speed" OLTP processing.

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.

5.3.1 Hardware Configuration

- two Intel E5530 Quad-Core-CPU, 8MB Cache, 2.4 GHz
- 8 GB RAM
- Linux operating system RHEL 5.4
- four 1TB SATA-HD (not used in the benchmark)

- 1 Gb/s network interconnect (possibly a logging bottleneck and should be scaled up to 10 Gb/s)
- Price: 3035 Euros (discounted price for universities)

5.3.2 H-Store Setup without Redo-Logging

In this experiment we compare HyPer against the published results of the initial H-Store paper [15]. Only two transaction types (New-Order and Payment) are run in sequential mode.

- 181488 tps w/o parallel query session (H-Store reported less than 50000 tps, i.e., 1/4-th of HyPer's throughput)
- 149476 tps with parallel query session (H-Store cannot do this)
- 29ms per Query (14ms as a Column Store)
  - Traditional DBMS' like IBM DB2 take about 3 to 5 times longer to execute such a query – on a warm cache without any additional OLTP load.

With asynchronous redo-logging to Null-Device

- 164203 tps w/o parallel Query Session
- 136798 tps with parallel query session

With asynchronous redo-logging via slow 1 Gb/s network

- 113895 tps w/o parallel query session
- 96993 tps with parallel query session

5.4 Full TPC-C Transaction Suite

Here the full suite of TPC-C transactions was run, i.e., all five transactions types in the specified workload composition. Again, the OLTP process ran without and with a parallel query session. The throuphputs are given as tuples (new-order tps / count of all five tps).

Without redo-logging

- 73548/161290 tps w/o Query Session
- 66087/144928 tps with parallel query session
- 26ms per Query (8ms as Column Store)

With asynchronous redo-logging to Null-Device

- 68059/149254 tps w/o parallel Query Session
- 61621/135135 tps with parallel query session

With asynchronous redo-logging via slow 1 Gb/s network

- 53647/117647 tps w/o parallel query session
- 46530/102041 tps with parallel query session

With five parallel Workers For this benchmark we partitioned the database horizontally by Warehouse – except for the Items table that was modelled as a shared read-mostly (actually read only) partition. The system configuration resembles the one shown in Figure 9, except that there are five horizontal partition, one for each Warehouse.

- 179928/351168 w/o Query Sessions
- 170519/335649 with parallel query sessions
- 8ms per Query (5ms as Column Store)

With asynchronous redo-logging via slow 1 Gb/s network

- 158676/309521 w/o Query Sessions
- 139969/272919 with parallel query session

This is faster than the fastest currently published TPC-C result. Even the results with a parallel query session constitute a "world record" – even though the competitors were only performing the OLTP transactions, while HyPer was additionally executing the query session.

These throughput results obtained on a single commodity desktop PC correspond to the published throughput results of VoltDB [26] on a 6-node cluster. Again, the VoltDB system cannot support the parallel session of successive OLAP queries.

5.4.1 Fork-ing a New Query Session

Forking a new query session and its corresponding transaction consistent snapshot is surprisingly fast. It took only **8 ms** without substantial load and **11 ms** with query load from another (older) query session. Note: the OLTP process is always quiesced during the fork-operation, so it cannot generate any background load. Being able to fork within 10 ms means that we can afford to create a fresh transaction consistent snapshot every second by giving up only 1% of the OLTP throughput performance.

6 Summary of the Innovation and Potential Impact

Our HyPer architecture is based on virtual memory supported snapshots on transactional data for multiple query sessions. The snapshot consistency and the high processing performance in terms of OLTP throughput and OLAP query response times is achieved as follows:

- Copy on demand (= write) to preserve snapshot consistency
    - detection of shared pages that need replication by OS/Processor

  - as current as needed (fork a new session within 10 ms any time in between OLTP transactions)
  - Replication at page granularity

- Abandon old-style buffer management and page management. Instead, rely on fast operating system/hardware support for VM address translation and snapshot consistency
  - page table management
  - Memory Management Unit (MMU), Translation Lookaside Buffer (TLB)
  - reference counting for shared pages across several snapshots (implicit by virtual memory management)

- Use multi core architecture effectively
  - 1 OLTP process (possibly multi-threaded on partitions)
  - Multiple OLAP sessions (one possibly multi-threaded process per session or separate processes forked from the primary OLAP session's process)

Potential Impact of the HyPer Techniques

Currently, main-memory database systems are explicitly architected for one of the two applications scenarios: OLTP **or** OLAP. Thus, HyPer's snapshot (Copy-on-Write) technique for shadowing the pages of the OLTP process and then merging them back into the OLAP database are beneficial for either class of systems:

- The OLAP-centric systems such as SAP's TREX and MonetDB could thereby gain the high-throughput OLTP functionality and

- the OLTP-centric systems, such as Oracles's TimesTen, SAP's P*Time, or VoltDB's H-Store could be augmented to achieve the high-performance and snapshot-consistent OLAP query processing functionality.

References

[1] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In Wang [27], pages 1195–1206.

[2] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold. A comparison of flexible schemas for software as a service. In Çetintemel et al. [9], pages 881–888.

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

[4] 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 M. J. Carey and D. A. Schneider, editors, *SIGMOD Conference*, pages 1–10. ACM Press, 1995.

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

[6] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In Çetintemel et al. [9], pages 283–296.

[7] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.

[8] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4), 2009.

[9] U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. ACM, 2009.

[10] S. K. Cha and C. Song. P*TIME: Highly scalable OLTP dbms for managing update-intensive stream workload. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *VLDB*, pages 1033–1044. Morgan Kaufmann, 2004.

[11] 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 B. Yormark, editor, *SIGMOD Conference*, pages 1–8. ACM Press, 1984.

[12] S. Finkelstein, D. Jacobs, and R. Brendle. Principles for inconsistency. In *CIDR*. www.crdrdb.org, 2009.

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

[14] T. Härder and E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage*. Springer, 2001.

[15] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In Wang [27], pages 981–992.

[16] E. Jones and A. Pavlo. A specialized architecture for high-throughput OLTP applications. 13th Intl. Workshop on High Performance Transaction Systems, Oct 2009. `http://www.hpts.ws/`.

[17] 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.

[18] A. Kemper and A. Eickler. *Datenbanksysteme - Eine Einführung, 7. Auflage*. Oldenbourg, 2009.

[19] A. Kemper, D. Kossmann, and F. Matthes. SAP R/3: A database application system (tutorial). In L. M. Haas and A. Tiwary, editors, *SIGMOD Conference*, page 499. ACM Press, 1998.

[20] S. Krompass, H. A. Kuno, J. L. Wiener, K. Wilkinson, U. Dayal, and A. Kemper. A testbed for managing dynamic mixed workloads. *PVLDB*, 2(2):1562–1565, 2009.

[21] A.-P. Liedes and A. Wolski. Siren: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases. In L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, editors, *ICDE*, page 99. IEEE Computer Society, 2006.

[22] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

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

[24] 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*, 43(4):92–105, 2009.

[25] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In Çetintemel et al. [9], pages 1–2.

[26] VoltDB. Technical overview. `http://www.voltdb.com/_pdf/VoltDBOverview.pdf`, March 2010.

[27] J. T.-L. Wang, editor. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008.* ACM, 2008.