Transaction Processing in the Hybrid OLTP&OLAP Main-Memory Database System HyPer

Alfons Kemper Thomas Neumann Jan Finis Florian Funke
Viktor Leis Henrik Mühe Tobias Mühlbauer Wolf Rödiger
TU München, Faculty of Informatics
firstname.lastname@in.tum.de

Abstract

Two emerging hardware trends have re-initiated the development of in-core database systems: ever increasing main-memory capacities and vast multi-core parallel processing power. Main-memory capacities of several TB allow to retain all transactional data of even the largest applications in-memory on one (or a few) servers. The vast computational power in combination with low data management overhead yields unprecedented transaction performance which allows to push transaction processing (away from application servers) into the database server and still "leaves room" for additional query processing directly on the transactional data. Thereby, the often postulated goal of real-time business intelligence, where decision makers have access to the latest version of the transactional state, becomes feasible. In this paper we will survey the HyPerScript transaction programming language, the main-memory indexing technique ART, which is decisive for high transaction processing performance, and HyPer's transaction management that allows heterogeneous workloads consisting of short pre-canned transactions, OLAP-style queries, and long interactive transactions.

1 Introduction

Main-memory or in-core/in-memory database systems have been around since the 1980s (e.g., TimesTen), but only recently has DRAM become inexpensive enough to render large enterprise applications possible on solely main-memory-resident data. This has initiated industrial interest in main-memory databases, e.g., SAP's HANA [3] or Microsoft's Hekaton [8]. Along with ever growing main-memory capacities comes a dramatic increase in compute power through multi-core parallelism. This vast performance increase renders many redundancy-based optimization techniques like materialized views obsolete, which were designed to avoid scanning large fragments of a database in an interactive query. The abundance of compute power finally allows to push data-intensive applications directly into the database server as opposed to the currently employed multi-tier architectures where large amounts of data have to be transferred to the application servers. Retaining all transactional data in-core is even possible for the largest companies, as a simple "back of the envelope" calculation reveals: Amazon.com generated a revenue of US$60 billion in 2012. Assuming an average item price of $15 and that each order line incurs stored data of about 54 bytes – as specified for the TPC-C-benchmark that models such a retailer – we derive a total data volume of less than 1/4 TB per year for the order lines, which is the dominating repository in such a sales application. This estimate neither includes 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 transactional sales data can be fit into the main memory of a large scale server with a few TB capacity. To unleash the immense computing power of such a multi-core server, a radical reengineering of database technology is required in several areas: low-overhead data representation, compiled instead of interpreted code, high-performance indexing and low-cost transaction synchronization are addressed here. Such a low-overhead database on a multi-core server would not even be saturated by the transaction (Tx) processing as a simple calculation reveals: On average, Amazon.com has only 127 sales Tx per second (in peak times they report a few thousand), while a main-memory database system like HyPer achieves around 100,000 Tx per second in the TPC-C benchmark. Thus, besides the mission critical transaction processing there is room for additional OLAP-style query processing – if they don't interfere with each other. HyPer allows this via a highly efficient virtual memory snapshotting mechanism. These virtual memory snapshots are created frequently and shield the complex OLAP query processing entirely from the mission-critical OLTP processing without any kind of (software) concurrency control. These snapshots are also leveraged for tentative execution of long transactions whose effect are then, after validation, applied to the main database as a short install transaction.

2 Transaction Scripting and Compilation

Transactions are implemented in HyPerScript, a SQL-based programming language. The SQL query language used for OLAP-style queries thereby is a subset of HyPerScript. For illustration purposes, we show the *complete* implementation of the well-known *newOrder* transaction of the TPC-C benchmark [14].

```
create procedure newOrder (w_id integer not null, d_id integer not null, c_id integer not null,
      table positions(line_number integer not null, supware integer not null,
                      itemid integer not null, qty integer not null),
      datetime timestamp not null)  // note the TABLE-valued parameter above
{  select w_tax from warehouse w where w.w_id=w_id; // w_tax value used later
   select c_discount from customer c // c_discount used in orderline insert
       where c_w_id=w_id and c_d_id=d_id and c.c_id=c_id;
   select d_next_o_id as o_id,d_tax from district d // get the next o_id
       where d_w_id=w_id and d.d_id=d_id;
   update district set d_next_o_id=o_id+1 // increment the next o_id
       where d_w_id=w_id and district.d_id=d_id;

   select count(*) as cnt from positions; // how many items are ordered
   select case when count(*)=0 then 1 else 0 end as all_local
       from positions where supware<>w_id;
   insert into "order" values (o_id,d_id,w_id,c_id,datetime,0,cnt,all_local);
   insert into neworder values (o_id,d_id,w_id); // insert reference to order

   update stock
   set s_quantity=case when s_quantity>qty then s_quantity-qty else s_quantity+91-qty end,
       s_remote_cnt=s_remote_cnt+case when supware<>w_id then 1 else 0 end,
       s_order_cnt=s_order_cnt+case when supware=w_id then 1 else 0 end
   from positions where s_w_id=supware and s_i_id=itemid;

   insert into orderline // insert all the order positions
      select o_id,d_id,w_id,line_number,itemid,supware,null,qty,
             qty*i_price*(1.0+w_tax+d_tax)*(1.0-c_discount),
             case d_id when 1 then s_dist_01 when 2 then s_dist_02 when 3 then s_dist_03
                       when 4 ... when 9 then s_dist_09 when 10 then s_dist_10 end
      from positions, item, stock
      where itemid=i_id and s_w_id=supware and s_i_id=itemid
   returning count(*) as inserted; // how many were inserted?

   if (inserted<cnt) rollback; // not all ==> invalid item ==> abort
};
```

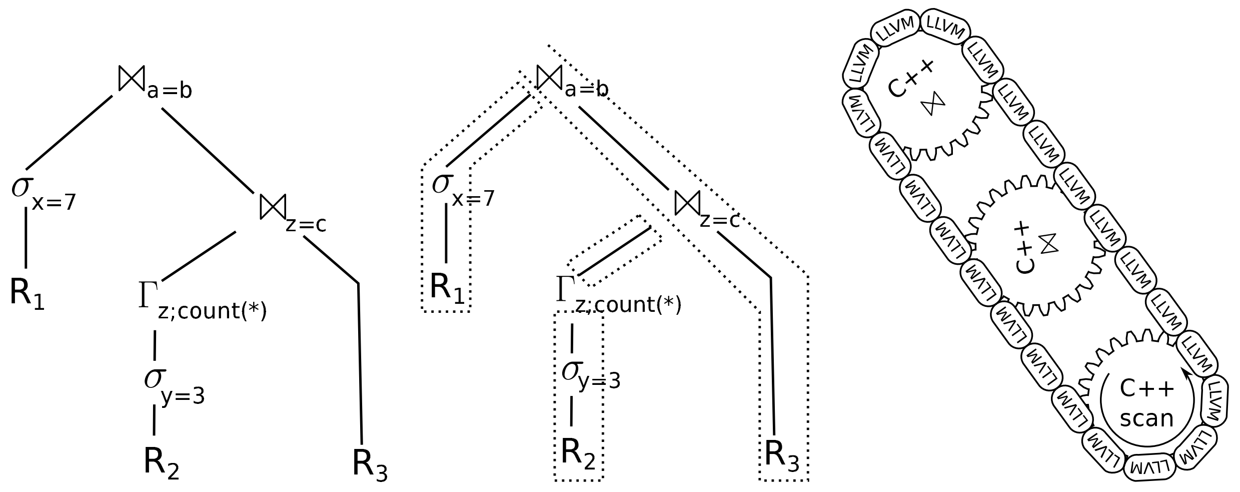

Figure 1: Compiling SQL queries into LLVM Code

The declarative HyPerScript language exhibits the following uncommon constructs that are present in the above script:

Table parameter. Frequently, a stored procedure is invoked for an entire collection of tuples, each consisting of several attribute values. Flattening the collection would result in a large number of parameters; a more elegant solution is provided by HyPerScript's table parameter that allows to pass an entire table – as demonstrated by the *positions* table in the *newOrder* script.

Reusing query results. In HyPerScript a query result can be reused in subsequent statements by referring to a prior assigned variable. For example, the first query of our *newOrder* script determines the tax rate *w_tax* of the particular warehouse. This *w_tax* value is later used in the insert statement that populates the *orderline* table with the item's price including the applicable tax.

Using such a declarative scripting language has a number of advantages: (1) the declarative nature allows for much more elegant and succinct code than in imperative languages – cf. the complete *newOrder* script; (2) the SQL statements can be optimized and executed like regular queries in the same query engine; (3) declarative HyPerScripts are much more amenable for security analysis than imperative programs. Thus, using the declarative HyPerScript language allows us to safely run compiled transactions within the database server process without any costly inter-procedural communication and context switching.

Stored procedures as well as regular (stand-alone) SQL queries are compiled into LLVM code [13]. LLVM constitutes a machine independent assembly language that is then further compiled and optimized for the particular server machine. The left-hand side of Fig. 1 shows a logical algebra plan for a three-way join including some selections and an (early) aggregation. Instead of the recently propagated vector-wise processing technique [2], HyPer's query engine relies on a data-centric execution model that exploits pipelining as much as possible. For this purpose, the query evaluation plan (QEP) is segmented into pipelines that end at pipeline breakers. As shown in the middle of Fig. 1, our example query has four such pipelines: the leftmost pipeline ends at the hash-table build of the upper join; R_2-tuples are scanned and selected up to the (hash) *groupify*. From there on, they are forwarded to the hash-table build of the lower join. Finally, R_3-tuples are propelled "in one go" via the first join all the way up to the upper join and produce output tuples by probing the hash-join table.

Thus, once a data object is accessed it is processed as much as possible. This way the query processing achieves maximal data locality as an object is transferred to a machine register as few times as possible.

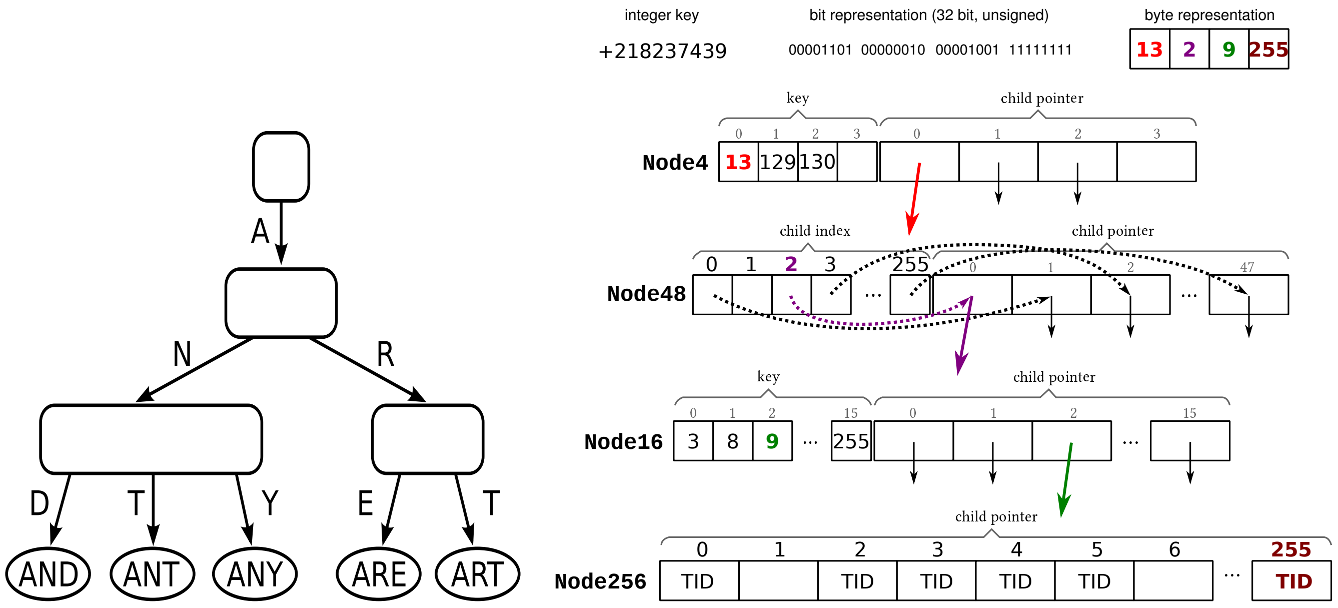

Figure 2: The Adaptive Radix Tree ART: (left) general idea of different sized nodes, (right) a sample path for the key 218237439 traversing all four node types

The right-hand side of Fig. 1 illustrates the translation of queries into corresponding LLVM/C++ code. The typical algorithms of a query engine, such as scans, hash-join table building and probing, grouping and aggregation, are pre-implemented in the high-level programming language C++. These C++ building blocks are "glued together" by generated LLVM – as (metaphorically) sketched for the rightmost pipeline of the example QEP. The generated LLVM code (the chain) makes the C++ query operators (the cog wheels) work together in evaluating an entire pipeline.

3 ARTful Indexing in Main-Memory Databases

The efficiency of transaction processing largely depends on which index structures are used, as exemplified by the first three *select*-statements of the *newOrder* implementation. In main-memory, dictionary-like data structures supporting insert, update, and delete are often implemented as hash tables or comparison-based trees (e.g. self-balancing binary trees or B-trees). Hashing is usually much faster than a tree as it offers constant lookup time in contrast to the logarithmic behavior of comparison-based trees. The advantage of trees is that the data is stored in sorted order, which enables additional operations like range scan, minimum, maximum, and prefix lookup.

The radix tree, also known as trie, prefix tree, or digital search tree, is another dictionary-like data structure. In contrast to comparison-based structures, which compare opaque key values using a comparison function, radix trees directly use the binary representation of the key. Although radix trees are often introduced as a data structure for storing character strings (cf., left hand side of Fig. 2), they can be used to store any data type by considering values as strings of bits or bytes.

The complexity of radix trees for insert, lookup, and delete is $O(k)$ where k is the length of the key. The access time is *independent* of the number of elements stored. Besides the length of the key, the height of a radix tree depends on the number of children each node has. For example, a radix tree with a fanout of 256 that stores 32 bit integers has a height of 4.

So far radix trees suffered from space underutilization problems as typically an array of 256 pointers was

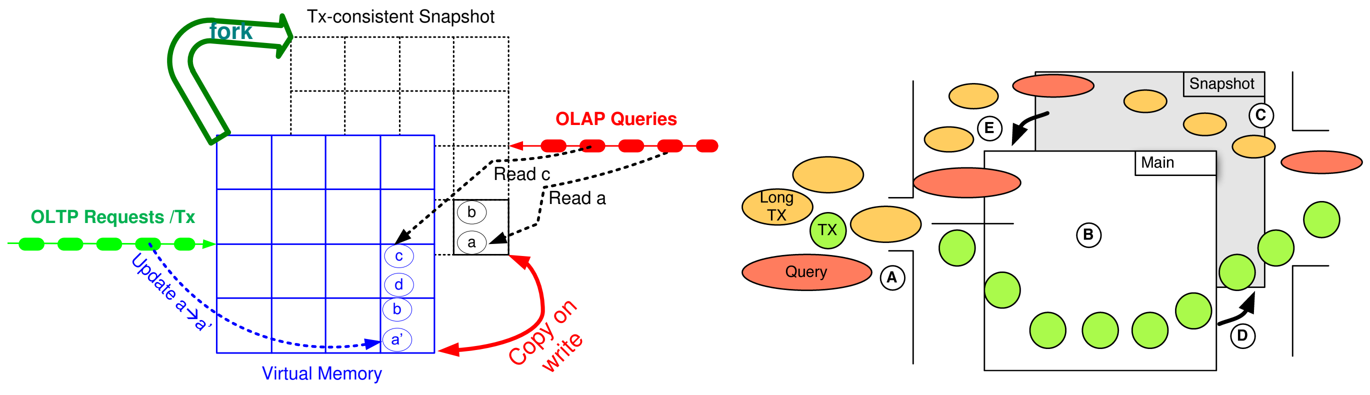

Figure 3: Virtual Memory Snapshots: (left) to separate OLTP & OLAP; (right) OLAP queries and long Tx are delegated to the snapshot (A), short transactions are executed on the main (B), long Tx are optimistically executed on the snapshot (C) and re-executed after validation as an apply transaction (E) on the main

allocated for each node – even though some nodes might have a very low fan-out compared to others. Therefore, we developed the Adaptive Radix Tree (ART), which uses four different node types that can handle up to (i) 4, (ii) 16, (iii) 48, and (iv) 256 entries. Thereby, a good space utilization of ART-trees is guaranteed while still being able to achieve a maximum height of k for k-byte keys. That is, 32 bit integers are indexed with a tree of height 4, 64 bit integers require height 8. These adaptive nodes are exemplified by the sample path for the 32-bit key 218237439 consisting of the the 4 byte-chunks 13&2&9&255 on the right-hand side of Fig. 2. This path starts at the root, which happens to be of type *Node4*, and then covers the three other node types. The structural representation of these node types varies, as illustrated in the figure. In designing the node structure the trade-off between space utilization and intra-node search performance was taken into account.

Besides adaptive nodes, we employ two well-known techniques that reduce both the tree height and the space consumption. First, we build the tree lazily, i.e., any path that leads to a single leaf is truncated. As a consequence, the leaf is stored higher up in the tree. Only when another key with a shared prefix is inserted, an additional inner node is created as a "goalpost" between the two leaves. The second technique, path compression, removes each common path (e.g., "http://" when indexing URLs) and instead stores the path as an additional prefix of the following inner node. This avoids cache-inefficient chains of one-way nodes. When indexing long keys, lazy expansion and path compression are very effective in reducing the tree height. Additionally, the two optimizations allow to bound the worst-case space consumption per key/value pair to 52 bytes – even for arbitrarily long keys [9].

4 Isolation of Long Running Transactions

Originally, HyPer focussed on the execution of short, pre-canned transactions *and* OLAP-style, read-only queries which are executed on a snapshot of the data that is generated using the UNIX *fork*-mechanism [7, 10]. The snapshot is kept consistent by the memory management unit (MMU) via the copy-on-write procedure that will (automatically) detect shared pages and copy them prior to any update, as illustrated on the left of Fig. 3.

In this architecture the OLTP process "owns" the database and periodically (e.g., in the order of seconds or minutes) forks an OLAP process. This OLAP process constitutes a fresh transaction-consistent snapshot of the database. Thereby, the OLAP processing is completely separated from the OLTP processing without any (software) concurrency control mechanism. Whenever an object (such as a in the figure) on a shared page is modified, the MMU automatically creates a page copy via the copy-on-write mechanism.

The OLTP transactions are executed serially on (partitions of) the transactional database state – as pioneered by H-Store/VoltDB [6, 15]. Even though this yields unprecedented performance, serial execution is restricted to short and pre-canned transactions.

This makes main-memory database systems using serial execution unsuitable for "ill-natured" transactions like long-running OLAP-style queries or transactions querying external data – even if they occur rarely in the workload. In our approach [11], which we refer to as *tentative execution*, the coexistence of short and long-running transactions in main-memory database systems does not require recommissioning traditional concurrency control techniques like two phase locking. Instead, the key idea is to tentatively execute long-running transactions on a transaction-consistent database snapshot, that exists for OLAP queries anyway. Thereby, a long transaction is converted into a short "apply transaction" which is then, after validation, re-executed on the main database, as illustrated on the right of Fig. 3.

Different mechanisms can be used to separate the workload into short and long transactions. A simple approach is limiting the runtime or number of tuples each transaction is allowed to use before it has to finish. When a transaction exceeds this allotment – which can vary depending on the transaction's complexity or the number of partitions it accesses – it is rolled back using the undo log and re-executed using tentative execution.

Our approach is optimistic in that it queues and then executes transactions on a consistent snapshot of the database. This is advantageous as no concurrency control is required to execute short and apply transactions. Similar to other optimistic execution concepts a validation phase is required which makes some form of monitoring necessary.

During the apply phase, the effects of the transaction as performed on the snapshot are validated on the main database and then applied. This is done by injecting an "apply transaction" into the serial execution queue of the main database. As opposed to the transaction that ran on the snapshot, the apply transaction only needs to validate the work done on the snapshot, not re-execute the original transaction in its entirety or wait for external resources.

Specifically, we distinguish between two cases: When *serializability* is requested, all reads have to be validated. To achieve this, it is checked whether or not the read performed on the snapshot is identical to what would have been read on the main database. Depending on the monitoring granularity, the action performed here ranges from actually performing the read a second time on the main database to comparing version counters between snapshot and main.

When *snapshot isolation* is used, the apply transaction ensures that none of the tuples written on the snapshot have changed in the main database, therefore guaranteeing that the write sets of both the tentative transaction as well as all transactions that have committed on the main database after the snapshot was created are disjoint. This is achieved by either comparing the current tuple values to those saved in the log or by checking that all version counters for written tuples are equal both during the execution on the snapshot and on the main database.

5 Conclusion and Ongoing Work

The high performance of HyPer is due to several design decisions: (1) compiling SQL queries and HyPerScript transactions into LLVM assembler code instead of interpreted execution; (2) the new radix tree indexing which offers performance similar to hash tables while allowing range scans; (3) the virtual memory snapshot mechanism which entirely shields OLTP from OLAP without *any* (software controlled) synchronization; (4) lock-free partition-serial execution of short transactions while (5) long transactions are executed optimistically on the snapshot and then applied to the main database like a short transaction. Currently we are working on parallelizing the query execution [1], compacting the working set of the transactional database [5], supporting versioned hierarchical data [4], bulk data loading, and scaling out to processor clusters [12].

References

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main-memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

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

[3] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[4] J. Finis, R. Brunel, A. Kemper, T. Neumann, F. Färber, and N. May. DeltaNI: An efficient labeling scheme for versioned hierarchical data. In *SIGMOD*, 2013.

[5] F. Funke, A. Kemper, and T. Neumann. Compacting transactional data in hybrid OLTP&OLAP databases. *PVLDB*, 5(11):1424–1435, 2012.

[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*, 1(2):1496–1499, 2008.

[7] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main-memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[8] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[9] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful indexing for main-memory databases. In *ICDE*, 2013.

[10] H. Mühe, A. Kemper, and T. Neumann. How to efficiently snapshot transactional data: hardware or software controlled? In *DaMoN*, pages 17–26, 2011.

[11] H. Mühe, A. Kemper, and T. Neumann. Executing long-running transactions in synchronization-free main memory database systems. In *CIDR*, 2013.

[12] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. ScyPer: Elastic OLAP throughput on transactional data. In *Workshop on Data analytics in the Cloud (DanaC)*, 2013.

[13] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 2011.

[14] Transaction Processing Performance Council. TPC-C specification. `www.tpc.org/tpcc/spec/TPC-C\_v5-11.pdf`, 2010.

[15] VoltDB. Technical Overview. `http://www.voltdb.com`, March 2010.