The Database Group at TUM

www-db.in.tum.de

Alfons Kemper
Technische Universität München
kemper@in.tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

The database group at TUM was established in 1972 by Rudolf Bayer [4] who retired in 2004. The group is reponsible for the foundational and advanced database curriculum of the department's 3500 undergraduate and graduate students. The group consists of two post-docs and around 15 doctoral students. In the following we will overview the research agenda of the last couple of years only; previous projects and publications can be viewed on our web site.

1. THE HYPER PROJECT

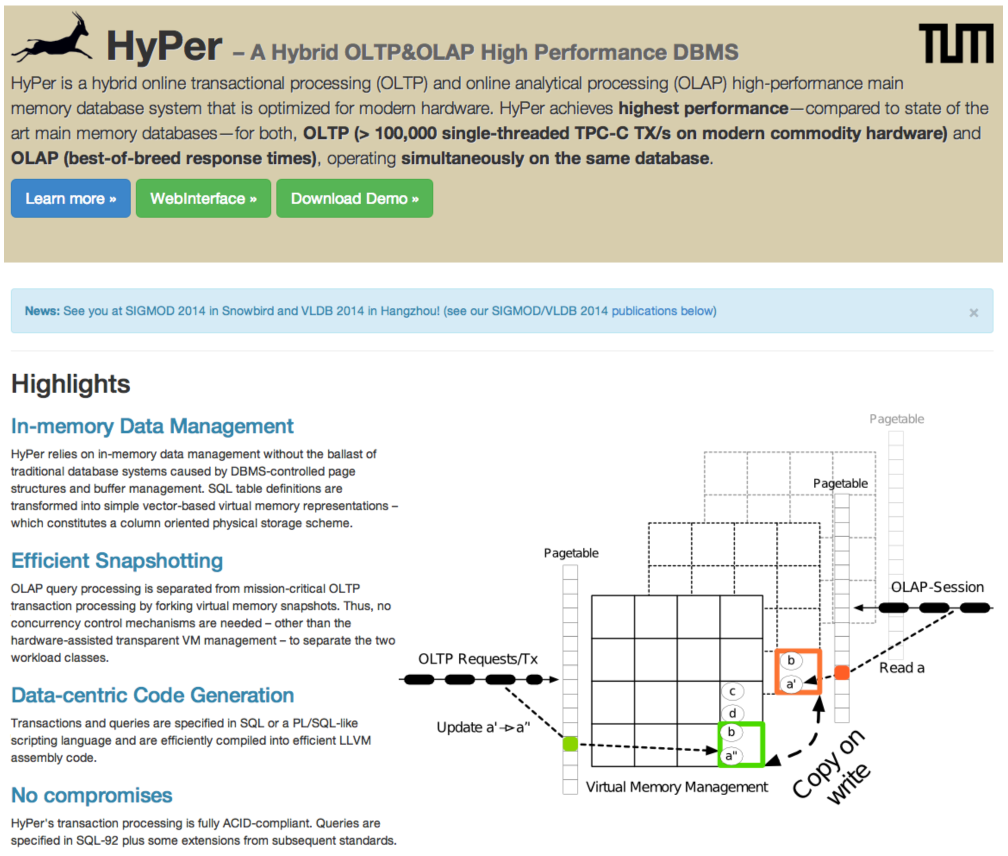
Most of the recent work of the TUM database group was done within the context of the long-term HyPer project. The goal of this project is to develop a high-performance database engine that (finally) unites the two seemingly disparate worlds: OLTP and OLAP. For this purpose we developped a hybrid main memory database system that supports OLTP- **and** OLAP-applications – in parallel on the same database state. The key idea is to exploit the OS/processor-support for virtual memory management. This allows to spawn consistent database snapshots to isolate OLAP queries from the OLTP transactions – even though they share the same database [13]. After about three years of development, HyPer now is a fairly mature database system and can be experimented with by other researchers via our web site `hyper-db.de` as shown in Figure 1.

In HyPer 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, we exploit operating systems functionality to create virtual memory snapshots for new, cloned processes. In Unix, for example, this is done by creating a child process of the OLTP process via the `fork` system call.

The forked child process obtains an exact copy of the parent processes address space, as exemplified on the lower right-hand side in Figure 1 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. These queries can be executed in parallel threads or serially, depending on the system resources or client requirements. In essence, the virtual memory snapshot mechanism constitutes a OS-/hardware supported shadow paging mechanism as proposed decades ago for disk-based database systems by Lorie [17]. However, the original proposal incurred severe costs as it had to be software-controlled and it destroyed the clustering on disk. Neither of these drawbacks occurs in the virtual memory snapshotting as clustering across RAM pages is not an issue. Furthermore, the sharing of pages and the necessary copy-on-update/write is managed by the operating system with effective hardware support of the MMU (memory management unit) via the page table that translates VM addresses to physical pages and traps necessary replication (copy-on-write) actions.

Our performance results demonstrate that HyPer combines the best of the two "worlds": HyPer's OLTP performance is comparable to that of dedicated OLTP engines (like VoltDB or Hekaton) and HyPer's OLAP query response times match those of the best pure OLAP engines (e.g., MonetDB and VectorWise). It should be emphasized that HyPer can match (or beat) these two best-of-breed transaction (VoltDB) and query (MonetDB, VectorWise) processing engines **at the same time** by performing both workloads in parallel on the same database state. This performance evaluation was carried out on the basis of a new business intelligence benchmark, the so-called CH-benCHmark [7], that combines the transactional workload of TPC-C with the OLAP queries of TPC-H – executed against the **same** database state. Hyper's excellent performance is due to the following design choices:

- HyPer relies on in-memory data management without the ballast of traditional database systems that is caused by DBMS-controlled page structures and buffer management. The SQL table definitions are transformed into simple vector-based virtual memory representations – which constitutes a column-oriented physical storage scheme.


HyPer – A Hybrid OLTP&OLAP High Performance DBMS

HyPer is a hybrid online transactional processing (OLTP) and online analytical processing (OLAP) high-performance main memory database system that is optimized for modern hardware. HyPer achieves **highest performance**—compared to state of the art main memory databases—for both, **OLTP (> 100,000 single-threaded TPC-C TX/s on modern commodity hardware)** and **OLAP (best-of-breed response times)**, operating **simultaneously on the same database**.

Learn more » WebInterface » Download Demo »

News: See you at SIGMOD 2014 in Snowbird and VLDB 2014 in Hangzhou! (see our SIGMOD/VLDB 2014 publications below)

Highlights

In-memory Data Management

HyPer relies on in-memory data management without the ballast of traditional database systems caused by DBMS-controlled page structures and buffer management. SQL table definitions are transformed into simple vector-based virtual memory representations – which constitutes a column oriented physical storage scheme.

Efficient Snapshotting

OLAP query processing is separated from mission-critical OLTP transaction processing by forking virtual memory snapshots. Thus, no concurrency control mechanisms are needed – other than the hardware-assisted transparent VM management – to separate the two workload classes.

Data-centric Code Generation

Transactions and queries are specified in SQL or a PL/SQL-like scripting language and are efficiently compiled into efficient LLVM assembly code.

No compromises

HyPer's transaction processing is fully ACID-compliant. Queries are specified in SQL-92 plus some extensions from subsequent standards.

Figure 1: `www.hyper-db.de` Web Site

- The OLAP processing is separated from the mission-critical OLTP transaction processing by **forking** virtual memory snapshots. Thus, no concurrency control mechanisms other than the hardware-assisted VM management are needed to separate the two workload classes.

- Transactions and queries are specified in SQL and are efficiently compiled into LLVM assembly code [21]. The transactions are specified in an SQL scripting language (called HyPer-Script) and registered as precanned, stored procedures. For the JIT-compiled interactive queries the very fast compilation into assembly language (LLVM) as opposed to a slow cross-compilation (into C/C++) is essential. The query evaluation follows a data-centric paradigm by applying as many operations on a data object as possible in between pipeline breakers. This evaluation scheme goes one step beyond cache-locality towards register-locality.

- The lock-free execution model [16, 18] in combination with group committing achieves extreme scalability in terms of transaction throughput – without compromising the "holy grail" of ACID (that was sacrificed by the NoSQL/key value stores).

- While in-core OLAP query processing can be based on sequential scans, this is not possible for transaction processing. Therefore, we have developed a sophisticated main-memory indexing structure, the ART tree [15].

- We developed an extremely efficient loading process [19] that allows to use HyPer for big data exploration by loading data a window at a time for deep data analysis.

- For efficiently scaling out the HyPer database engine we developed a locality-sensitive query engine [23].

- The index structure DeltaNI [10] was developed to support hierarchical data in main-memory database systems.

- HyPer has a very small memory footprint which allows to run the same system on both, brawny nodes with up to several TB of main-memory and a hundred cores as well as on wimpy nodes, such as tablets and smartphones [20].

Considering that it is an active research project the implementation of HyPer is fairly mature. Even though it is not open source, we make it publicly available for experimentation via our web site `hyper-db.de`. We

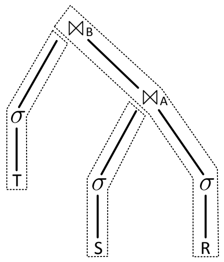


Figure 2: An Execution Plan with visible Pipeline Fragments for a Three-Way-Join

also use it for teching purposes; e.g., our 750 students of the lecture "Foundations of Database Systems" use it for learning SQL programming and query optimizer experiments.

2. OVERVIEW OF RECENT WORK

2.1 Efficient Query Compilation

HyPer uses a lock-free execution model. This assumes that transactions, in particular OLTP transactions, are very fast. HyPer achieves this by using LLVM for just-in-time compilation of SQL queries to machine code [21]. The compilation works in a data-centric manner: Instead of compiling one operator at a time, the compiler generates code for each individual data pipeline of the execution plan. This is illustrated in Figure 2. The pipelines, i.e., the data flow paths from one materialization point to the next, are compiled in a bottom-up manner where every operator pushes tuples towards its consumer. In code this means that most pipeline fragments consist of a few tight loops, which is favorable for modern CPUs and results in excellent performance. This compilation step avoids the high CPU overhead of classical interpreted execution frameworks. For disk-based systems this overhead was largely neglectable, but for in-memory processing any interpretation overhead is very visible.

2.2 NUMA-Aware Many-Core Parallelism

The main impetus of hardware performance improvement nowadays comes from increasing multi-core parallelism rather than from speeding up single-threaded performance. Intel's new mainstream server Ivy Bridge EX (that we recently obtained) can run 120 concurrent threads in a 4-socket configuration. We use the term *many-core* for such architectures with tens or hundreds of cores.

At the same time, increasing main memory capacities of up to several TB per server have led to the development of main-memory database systems. In these systems query processing is no longer I/O bound, and the huge parallel compute resources of many-cores can be truly exploited. Unfortunately, the trend to move memory controllers into the chip and hence the decentralization of memory access, which was needed to scale throughput to huge memories, leads to non-uniform memory access (NUMA). In essence, the computer has become a network in itself as the access costs of data items varies depending on which chip the data and the accessing thread are located. Therefore, many-core parallelization needs to take RAM and cache hierarchies into account. [1, 3] developed and analyzed massively parallel join algorithms. In HyPer [14] we extend this work to an effective end-to-end parallelization scheme covering entire query evaluation plans. In particular, the NUMA division of the RAM has to be considered carefully to ensure that threads work (mostly) on NUMA-local data.

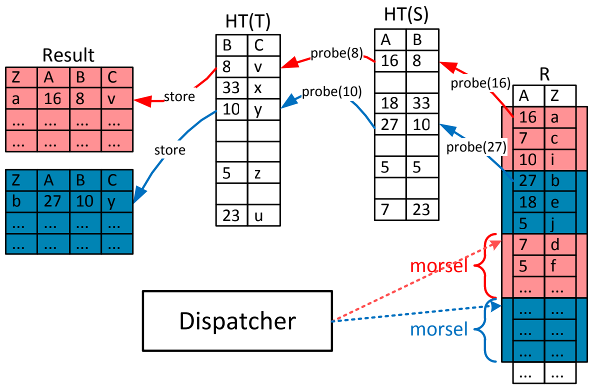


Figure 3: Idea of morsel-driven parallelism: $R \bowtie_A S \bowtie_B T$

HyPer employs an adaptive *morsel-driven* query execution framework – as described in [14]. Our approach is sketched in Figure 3 for the three-way-join query $R \bowtie_A S \bowtie_B T$. Parallelism is achieved by processing each pipeline on different cores in parallel, as indicated by the two (upper and lower) *probe-probe-store*-pipelines in the figure. The core idea is a *scheduling* mechanism (the "dispatcher") that allows flexible parallel execution of an operator pipeline, that can change the parallelism degree even during query execution. A query is divided into segments, and each executing segment takes a morsel (typically 100,000) of input tuples and executes these, materializing results in the next pipeline breaker. The morsel framework enables NUMA local processing as indicated by the color coding in the figure: A thread operates on NUMA-local input and writes its result into a NUMA-local storage area. Our dispatcher runs a fixed, machine-dependent number of threads, such that even if new queries arrive there is no resource oversubscription, and these threads are pinned to the cores, such that no unexpected loss of NUMA locality can occur due to the OS moving a thread to a different core.

The crucial feature of morsel-driven scheduling is that task distribution is done at run-time and is thus *fully elastic*. This allows to achieve perfect load balancing, even in the face of uncertain size distributions of in-

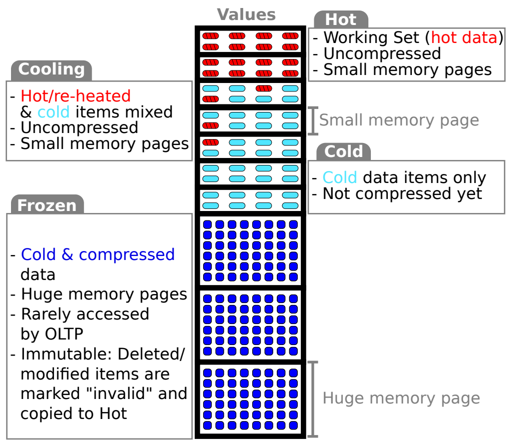


Figure 4: Hot/cold Clustering for Compression.

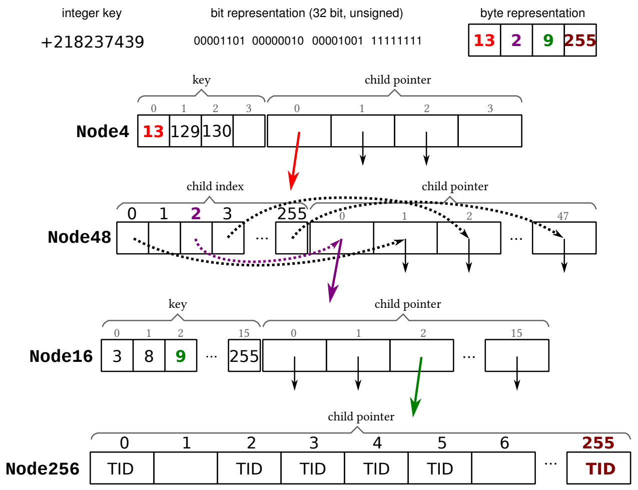


Figure 5: The Adaptive Radix Tree ART: Sample Path for the Key 218237439 Traversing all four Node Types

termediate results, as well as the hard-to-predict performance of modern CPU cores that varies even if the amount of work they get is the same. It is elastic in the sense that it can handle workloads that change at run-time (by reducing or increasing the parallelism of already executing queries in-flight) and can easily integrate a mechanism to run queries at different priorities.

2.3 Compacting the In-Memory Database

Our approach [11] to compression in hybrid OLTP & OLAP column stores is based on the observation that while OLTP workloads frequently modify the dataset, they often follow the working set assumption: Only a small subset of the data is accessed and an even smaller subset of this working set is being modified (cf. Figure 4). In business applications, this working set is mostly comprised of tuples that were added to the database in the recent past, as it can be observed in the TPC-C workload (cf. `www.tpc.org`).

Our system uses a lightweight monitoring component to observe accesses to the dataset and identify opportunities to reorganize data such that it is clustered into hot and cold parts. After clustering, the database system compresses cold chunks to reduce memory consumption and streamline queries. In future versions we will even stage the frozen parts to less expensive memory, e.g., to non-volatile memory NVM, SSD or disk storage media. Our concept of compaction has received a lot of attention and is currently being incorporated in Microsoft's Hekaton [9] as well as in HStore/VoltDB [8].

2.4 Radix Tree 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 of the TPC-C benchmark. 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, 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 allocated for each node – even though some nodes might have a very low fan-out compared to others. Therefore, we developed in [15] 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 in Fig. 5. 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 other compression techniques that bound the worst-case space consumption per key/value pair to 52 bytes – even for arbitrarily long keys [15].

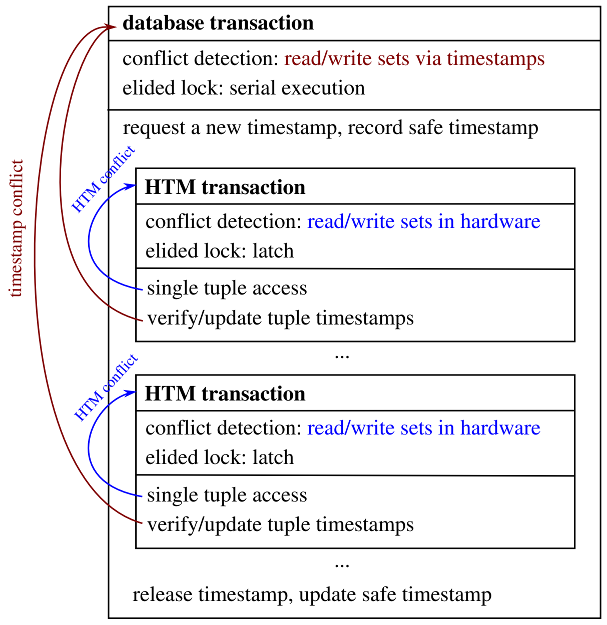
2.5 Lock-Free Synchronization via Hardware Transactional Memory

The upcoming hardware transactional memory (HTM) support in mainstream processors like Intel's Haswell appears to be a perfect fit for optimizing the emerging main-memory database systems. Transactional memory [12] is a very intriguing concept that allows for automatic atomic and concurrent execution of arbitrary code.

However, transactional memory is no panacea for transaction processing. First, database transactions also require properties like durability, which are beyond the scope of transactional memory. Second, at least the current hardware implementations of transactional memory are limited. For the Haswell architecture the read-/write sets have to fit into the L1 cache with a capacity of 32KB, which limits the scope of a transaction. Furthermore, HTM transactions may fail due to a number of unexpected circumstances like collisions caused by cache associativity, hardware interrupts, etc. Therefore, it does not seem to be viable to map an entire database transaction to a single monolithic HTM transaction. In addition, one always needs a "slow path" to handle the pathological cases (e.g., associativity collisions).

We therefore developed in [15] an architecture where transactional memory is used as a building block for assembling complex database transactions. Along the lines of the general philosophy of transactional memory we start executing transactions optimistically, using (nearly) no synchronization and thus running at full clock speed. By exploiting HTM we get many of the required checks for free, without complicating the database code, and can thus reach a much higher degree of parallelism than classical locking or latching. In order to minimize the number of conflicts in the transactional memory component, we carefully control the data layout and the access patterns of the involved operations, which allows to operate largely without explicit synchronization.

Because the maximum size of hardware transactions is limited, only a database transaction that is small can



database transaction
conflict detection: read/write sets via timestamps
elided lock: serial execution
request a new timestamp, record safe timestamp
timestamp conflict
HTM conflict
HTM transaction
conflict detection: read/write sets in hardware
elided lock: latch
single tuple access
verify/update tuple timestamps
...
HTM conflict
HTM transaction
conflict detection: read/write sets in hardware
elided lock: latch
single tuple access
verify/update tuple timestamps
...
release timestamp, update safe timestamp

Figure 6: Transforming database transactions into HTM transactions

directly be mapped to a single hardware transaction. We therefore assemble complex database transactions by using hardware transactions as building blocks, as shown in Figure 6. The key idea here is to use a customized variant of timestamp ordering (TSO) to "glue" together these small hardware transactions. TSO is a classic concurrency control technique, which was extensively studied in the context of disk-based and distributed database systems [6, 5]. For disk-based systems, TSO is not competitive to locking because most read accesses result in an update of the read timestamp, and thus a write to disk. The timestamp updates are obviously much cheaper in RAM. On the opposite, fine-grained locking is much more expensive than maintaining timestamps in main memory, as we showed in [15].

Timestamp ordering uses read and write timestamps to identify read/write and write/write conflicts. Each transaction is associated with a monotonically increasing timestamp, and whenever a data item is read or updated its associated timestamp is updated, too. The *read timestamp* of a data item records the youngest reader of this particular item, and the *write timestamp* records the last writer. This way, a transaction recognizes if its operation collides with an operation of a "younger" transactions (i.e., a transaction with a larger timestamp), which would be a violation of transaction isolation. In particular, an operation fails if a transaction tries to read data from a younger transaction, or if a transaction tries to update a data item that has already been read by a younger transaction.

Lock-free execution is generally 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 [18] we developed an approach whereby long transactions are first executed tentatively on the virtual memory snapshot and then applied to the main database as a short install transaction.

3. AWARDS

In the last couple of years the database group at TUM has obtained the following awards:

- The team "Campers" of Henrik Mühe and Florian Funke (mentored by Alfons Kemper and Thomas Neumann) was among the finalists of the ACM SIGMOD Programming Contest 2013 and achieved second place.
- The team "AWFY" of Moritz Kaufmann, Manuel Then, Tobias Mühlbauer, und Andrey Gubichev (mentored by Alfons Kemper and Thomas Neumann) won the ACM SIGMOD Programming Contest 2014.
- The ICDE 2014 Best Paper Award was presented to Viktor Leis, Alfons Kemper and Thomas Neumann for their paper "Exploiting Hardware Transactional Memory in Main-Memory Databases" [16].
- Thomas Neumann was awarded the Early Career Research Contribution Award at VLDB 2014 for his work on "Engineering High-Performance Database Engines" [22].

4. GUESTS

- Nikolaus Augsten (University of Salzburg) visited the group to work on similarity join processing [2].
- Peter Boncz (CWI/VU Amsterdam) was awarded the Humboldt Prize to work with us on multi-core parallel query processing [14].

5. VLDB 2017 AT TUM

The database group of TUM will organize the VLDB 2017 conference in Munich. It will take place during the last week of August 2017 at the main campus of TUM in downtown Munich.

6. ACKNOWLEDGMENTS

Our work is supported by various industry grants (SAP, Siemens, Oracle Labs, Google, IBM, HP Labs, Intel) as well as by the German Research Foundation DFG, the EU and the German Federal Ministry for Economic Affairs and Energy (BMWi).

7. 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] N. Augsten, A. Miraglia, T. Neumann, and A. Kemper. On-the-fly token similarity joins in relational databases. In *SIGMOD Conference*, pages 1495–1506, 2014.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[4] R. Bayer. Kurzbiographie (in german). `http://www3.informatik.tu-muenchen.de/people/sites/bayer/Memoiren-Bayer.pdf`, 2014.

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

[6] M. J. Carey. *Modeling and evaluation of database concurrency control algorithms*. PhD thesis, 1983.

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

[8] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14), 2013.

[9] A. Eldawy, J. J. Levandoski, and P.-Å. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11):931–942, 2014.

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

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

[12] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[13] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[14] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[15] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, pages 38–49, 2013.

[16] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, pages 580–591, 2014.

[17] R. A. Lorie. Physical integrity in a large segmented database. *ACM TODS*, 2(1), 1977.

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

[19] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *PVLDB*, 6(14):1702–1713, 2013.

[20] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. One DBMS for all: the brawny few and the wimpy crowd. In *SIGMOD Conference*, pages 697–700, 2014.

[21] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9), 2011.

[22] T. Neumann. Engineering High-Performance Database Engines. *PVLDB*, 7(13), 2014.

[23] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*, pages 592–603, 2014.