Foundations and Trends® in Databases
Vol. 8, No. 1-2 (2016) 1–130
© 2017 F. Faerber, A. Kemper, P. Å Larson, J. Levandoski, T. Neumann, and A. Pavlo
DOI: 10.1561/1900000058

Main Memory Database Systems

Franz Faerber
SAP
franz.faerber@sap.com

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

Per-Åke Larson
Microsoft Research
palarson@acm.org

Justin Levandoski
Microsoft Research
justin.levandoski@microsoft.com

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

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Contents

Abstract

This article provides an overview of recent developments in main-memory database systems. With growing memory sizes and memory prices dropping by a factor of 10 every 5 years, data having a “primary home” in memory is now a reality. Main-memory databases eschew many of the traditional architectural pillars of relational database systems that optimized for disk-resident data. The result of these memory-optimized designs are systems that feature several innovative approaches to fundamental issues (e.g., concurrency control, query processing) that achieve orders of magnitude performance improvements over traditional designs. Our survey covers five main issues and architectural choices that need to be made when building a high performance main-memory optimized database: data organization and storage, indexing, concurrency control, durability and recovery techniques, and query processing and compilation. We focus our survey on four commercial and research systems: H-Store/VoltDB, Hekaton, HyPer, and SAP HANA. These systems are diverse in their design choices and form a representative sample of the state of the art in main-memory database systems. We also cover other commercial and academic systems, along with current and future research trends.

F. Faerber, A. Kemper, P. Å Larson, J. Levandoski, T. Neumann, and A. Pavlo. *Main Memory Database Systems*. Foundations and Trends® in Databases, vol. 8, no. 1-2, pp. 1–130, 2016.

DOI: 10.1561/1900000058.

1 Introduction

Research and development of main-memory database systems started in the early eighties [37], with several commercial systems appearing in the nineties (e.g., TimesTen [146], P*Time [28], DataBlitz [16]). Many of these systems were used in targeted, performance-critical applications, mainly in telecommunications and finance. The price and capacity of memory during this time period limited applicability of many of these engines, thus main-memory systems did not - at the time - succeed as a general data processing solution.

Recently, two trends have made this field interesting again: memory prices and multi-core parallelism. For the last 30 years memory prices have dropped by a factor of 10 every 5 years. A server with 32 cores and 1 TB of memory now costs around $40K. Machines such as these make it feasible to fit most (if not all) of the world's OLTP workloads[1] comfortably into memory at a reasonable price. In addition, modern CPUs provide a staggering amount of raw parallelism. Vanilla CPUs contain at least 8 cores, and it is common for modern servers to contain two to four CPU sockets (16 to 32 cores). Core counts continue to rise, with Intel currently shipping a Xeon CPU with 18 cores [1].

Such parallelism coupled with the ability to (practically) store data completely in memory has brought about a recent flurry of research and development into main-memory databases. The result has been astounding. Prominent research systems such as H-Store [142] and

[1]The focus of this survey is primarily on main-memory OLTP databases.

HyPeR [72] reinvigorated research into main-memory and multi-core data processing techniques. Most major database vendors now have an in-memory database solution, such as SAP HANA [137], Oracle TimesTen [74], and Microsoft SQL Server Hekaton [38]. In addition, a number of startups such as VoltDB [143] and MemSQL [2] have carved out a niche in the database vendor landscape.

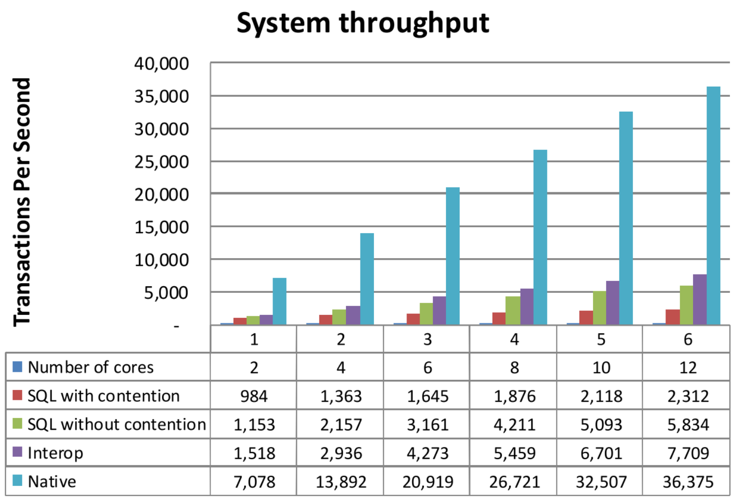
The result of this research and development is a new breed of database system with a radically different design when compared to a traditional disk-based relational system. These systems abandon many of the "textbook" design tenets in favor of new (or revisited) approaches to achieve high performance on modern hardware. For instance, the following examples provide an idea of how different these systems are:

- **Data organization and indexing.** A pervasive trend in all systems is to avoid page-based indirection through a buffer pool and store only records in memory. Indexes usually store direct pointers to records. Several systems also implement novel indexing methods that optimize for CPU cache efficiency [85] as well as multi-core parallelism using latch-free designs [89].

- **Concurrency control.** Most systems avoid pessimistic lock-based concurrency control due to blocking and context switch overheads. Instead, some systems use a multi-version concurrency control variant [11, 77, 74, 111], while others use partitioned serial execution to achieve high performance [72, 142, 143].

- **Durability and recovery.** Aries-style redo/undo logging and recovery is rarely used. Instead, most systems opt for a form of redo-only logging (or command logging) coupled with periodic database snapshots to recover from a crash or restart [38, 74, 142].

- **Query processing and compilation.** To avoid the overhead of virtual function calls, degradation of branch prediction, and byte interpretation, several systems abandon the "get next" iterator processing model. Instead, queries are compiled into highly optimized machine code and run directly over in-memory records [38, 47, 109].



	1	2	3	4	5	6
🟦 Number of cores	2	4	6	8	10	12
🟥 SQL with contention	984	1,363	1,645	1,876	2,118	2,312
🟩 SQL without contention	1,153	2,157	3,161	4,211	5,093	5,834
🟪 Interop	1,518	2,936	4,273	5,459	6,701	7,709
🟦 Native	7,078	13,892	20,919	26,721	32,507	36,375

Figure 1.1: Microsoft Hekaton vs the traditional SQL Server engine taken from [38]. Latch contention plays a large part in limiting the scalability of the traditional engine.

The list above resembles a set topics once thought to be picked over and "closed" in the database research literature. However, over the past several years there has been significant innovation in these and other core areas with the reemergence of main-memory database systems, making it a vibrant and exciting technology space.

The innovations made in main-memory databases come with meaningful performance gains. Figure 1.1 provides an evaluation of the thread-level scalability of the Microsoft Hekaton engine compared to the traditional SQL Server engine running a typical customer workload[2] . Hekaton achieves a roughly 15.7X performance improvement at 12 cores, while the scalability of the traditional engine is limited due to the overheads inherent in a disk-based architecture running a memory-bound workload.

This article provides an overview of the research and development of modern main-memory database systems. We focus our sur-

[2]Graph taken from the original paper [38]

vey mainly on transaction processing engines through the lens of four main-memory systems: (1) *H-Store/VoltDB* [142, 143], a pioneering research system from academic research groups at MIT, Brown, Brandeis, and Yale that subsequently became a commercial database; (2) *Hekaton* [38], Microsoft SQL Server's main-memory OLTP engine; (3) *HyPeR* [72], a prominent research system from TU Munich that aims to support both high performance OLTP and OLAP workloads in the same engine; and (4) *SAP HANA* [137], the first main-memory optimized engine to ship from a major database vendor. As we will see, each of these systems are diverse in their design choices and form a representative sample of the state of the art in main-memory database systems. Our coverage focuses on five issues that influence the architecture and design of the system: (a) data storage and layout, (b) indexing and data structure design, (c) concurrency control, (d) durability and recovery, and (e) query processing and compilation. We also summarize the design of other modern commercial and academic main-memory systems.

The organization of this survey is as follows. Chapter 2 summarizes the past research and development of main-memory database systems prior to the "modern era" (starting around 2007). Chapter 3 summarize the issues used in our system survey. Chapter 4 provides case studies describing the design of our four representative systems. Chapter 5 concludes this survey by summarizing interesting current and future research trends in main-memory databases.

2

History and Trends

While this survey primarily focuses on modern main-memory database system architecture and implementation techniques, research on main-memory databases goes all the way back to the mid nineteen-eighties. This section covers the history of main-memory database research. In addition, there is a broader trend in main-memory systems that reaches beyond databases. This section also highlights several areas of main-memory optimizations outside of the database systems space.

2.1 History

While there is currently a flurry of activity in the main-memory database space, there is actually a long history of research and development of main-memory optimized systems going back to the mid-eighties. This section summarizes this early work and compares an contrasts this early work with architectures adopted by today's systems.

2.1.1 The Early Years (1984–1994)

Research papers on main-memory database technology started to appear in the early eighties. In general, much of this work looked at improving performance of traditional relational database systems assuming most or all of the database fit in memory.

IMS Fast Path [50], released in 1976, is one of the first known systems to optimize for memory-resident data. Fast Path requires the user designate a database as memory-resident. Optimizations included fine-grained record-level locking and installing record updates at commit time. Early work from the University of Wisconsin [37] explored optimizing a relational database system when most or all of the relations fit comfortably in main memory. This work explored a number of areas, including access method performance, join techniques, and recovery optimizations such as fast-commit, group-commit, and parallel log I/O and checkpointing techniques.

The MM-DBMS project from the University of Wisconsin [80, 79, 81] performed seminal research in various areas of main-memory databases. MM-DBMS explored use of pointers for direct record access, new memory-optimized indexing methods (T-Trees), optimized two-phase locking, and new recovery techniques. The IBM Starburst memory-resident storage engine [82] was based on several of the optimizations proposed in MM-DBMS (e.g., T-Trees, recovery techniques). Follow-up work [52] explored lock and latching optimizations in Starburst that proposed a single latch to protect a table, all of its indexes, and related lock information. This work also proposed direct addressing of lock data to avoid indirection.

The MARS system [41, 58] partitioned the system into an in-memory database processor (DP) and recovery processor (RP), where RP lazily copied volatile updates to stable storage. Recovery work in MARS showed the benefits of avoiding undo logging in a main-memory system. System M [134, 135] was a prototype built to study recovery techniques in a main-memory environment. It also explored a partitioned technique that used message servers to process transactions and log servers to persist updates.

TPK [90] explored transaction processing in a multi-processor main-memory environments. TPK executed transactions serially and employed a collection of specialized threads (input, output, execution, and recovery) to process transactions.

Office-by-example (OBE) from IBM [21, 156] explored performance and optimization of query processing for main-memory data in an integrated office system. OBE explored several main-memory optimizations for read-mostly data, including sorted inverted indexes that relied on pointers for direct memory access.

PRISMA/DB [12, 13] explored two-phase locking and two phase commit over a partitioned DBMS, where each node stored data in main memory.

2.1.2 The New Millennium (1994–2007)

Main-memory database systems research in the mid-nineties onward started to reflect the trends and architectures we see in today's main-memory systems. For instance, systems like ClustRa [66] explored a distributed main-memory architecture, which is a design tenant followed by VoltDB and several other research prototypes. Some of the prototype systems built in this era matured into today's high-performance main-memory engines. For instance, P*Time is the main-memory transaction processing engine in SAP HANA, while TimesTen was acquired by Oracle.

Dalí was a main-memory database engine built by Bell Labs starting in the early nineties [67, 23]. Dalí was later commercialized as DataBlitz [16]. This system targeted high-performance multi-threaded execution of OLTP workloads, servicing many of AT&T's internal high performance applications. Dalí allowed applications direct shared-memory access to data records, a technique that allows for very fast execution but is not a common in modern main-memory systems. DataBlitz/Dalí supported data replication for fast failover. It also reduced I/O during recovery by performing redo-only logging (a transient undo log is maintained in memory and discarded after commit). Redo-only logging is a common practice in today's main-memory systems. DataBlitz/Dalí also featured fuzzy action-consistent checkpointing that reduced lock contention on checkpointing worker.

Several distributed main-memory architectures started to appear in the mid-to-late nineties. The ClustRa Database [66] was a distributed main-memory system custom-built for telecom workloads. ClustRa was built for high performance and availability and fully replicated database state at each node. It used a 2-safe replication scheme with independent failure modes.

System K from NYU [157] explored a programming language approach to high-performance in-memory transaction processing. This system processed transactions sequentially at in-memory partitions assigned to a CPU core (it did not distribute across machines). System K used logical logging, common in several of today's main-memory databases such as Hekaton and VoltDB, and also user-provided hints for specifying types of transactions that conflict.

The TimesTen main-memory database [146, 147, 148] started as a research project at HP Labs named "Smallbase" that was later spun off into a separate company. Like other main-memory systems, TimesTen implements a checkpointing and logging scheme that optimizes for main-memory execution. One unique aspect of TimesTen is that it is available as both a server as well a library that can be directly linked by applications. TimesTen also supports fully ACID transactions, but allows the user to relax the ACID properties for higher performance. This flexibility allows the TimesTen engine to be used in other application scenarios, such as mid-tier caching [148]. TimesTen was acquired by Oracle, and now serves as Oracle's high-performance main-memory database solution.

P*Time [28] is a light-weight main-memory OLTP database built for scalability on commodity machines. The system is optimized for L2-cache consciousness. Several internal data structures in P*Time are completely latch-free: a trend that is common in other modern main-memory systems such as Hekaton. P*Time also optimizes for durability and recovery by performing fine-grained parallel differential logging and recovery. Concurrency control involves multi-level locking. While P*Time started as a research prototype, it has become main-memory OLTP engine inside the SAP HANA database.

2.2 Trends

Current research in the broader systems and storage space assumes memory-resident data. While the focus of this survey is main-memory databases, we highlight two related trends in main-memory storage.

Embedded Key-Value Stores

Main-memory key-value stores are engines that provide atomic operations (e.g., update, delete) on a single record. Non-distributed key-value stores are usually available as a library that embeds in a larger application or system to provide high-performance record storage and access. The current trend in embedded key-value stores is to provide a high-performance, memory-optimized data structure (hash or range-based access) that enables fast data manipulation and retrieval. If persistence is needed, the key-value store makes updates durable by writing to secondary storage (usually flash) or through replication.

Examples of memory-optimized key-value store designs include Masstree [97], the Bw-tree [89], and MICA [92]. The Masstree is a memory-optimized key value store that provides range queries and scales well on modern symmetric multiprocessing machines. Masstree employs a cache-conscious index layout using a “trie-of-B+-trees” and implements an optimistic concurrency-control scheme that does not block readers. The Bw-tree is a completely latch-free (lock-free) B+-tree in memory. For durability, it implements a log-structured page store that writes sequentially back to stable storage (e.g., Flash SSD). The in-memory portion of the Bw-tree also serves as the range index in Hekaton. MICA is a very high-performance key-value store that optimizes the complete request-handling stack, from network request handling at the network interface card, through to data structure design. MICA uses a lossy hash index that scales well for concurrent thread access performing both reads and writes.

Distributed Key-Value Stores

Scale-out distributed in-memory key-value storage is another trend made possible by decreasing memory prices and price/performance improvements in commodity network infrastructure (e.g., RDMA). Perhaps the most common use for a distributed key-value store in practice is caching. Memcached [99] is one of the most widely used caching systems. While generic in nature, in practice it is used as a front-end cache for web applications. For instance, Facebook in 2009 reportedly used upwards of 2,000 memcached servers to cache roughly 25% of its data [117].

Recent projects have built distributed main-memory key-value stores for primary data storage. Unlike a cache, these systems do guarantee durability, all while maintaining high performance. RAMCloud [116] was the first project to explore such an approach, aiming for 100x to 1,000x performance improvements over disk-based distributed storage systems, along with 5μs-10μs latencies. RAMCloud organizes a cluster of storage servers as a distributed, scale-out memory pool for record storage. To achieve its performance goals, the RAMCloud project made a number of advances in fundamental areas such as recovery [114] and memory allocation [133]. FaRM [39] is another scale-out distributed main-memory key-value store. FaRM has a lower-level interface than RAMCloud, exposing memory from a cluster of machines as a shared address space. Features of FaRM include lock-free reads over RDMA as well as function shipping for running transactions locally on a single machine.

3

Issues and Architectural Choices

The architecture and implementation techniques in main-memory database systems differ from those of traditional relational databases. In this section, we cover a number of issues that allow main-memory systems to achieve high performance. Each of these issues influences architectural choices and ultimately determine the system design. For each issue, we first summarize how it is addressed by traditional disk-based systems. We then describe how main-memory environments differ and summarize various design alternatives available to modern systems. Later in Chapter 4, we detail how our representative systems address the issues in practice.

3.1 Data Organization and Layout

This section covers the basic organization and layout issues in main-memory database systems. As we will see, a primary design tenet is to avoid page-based indirection as done in disk-based relational systems. In addition, we summarize a number of organizational decisions made by modern main-memory systems such as partitioning, multi-versioning, and row/columnar layout.

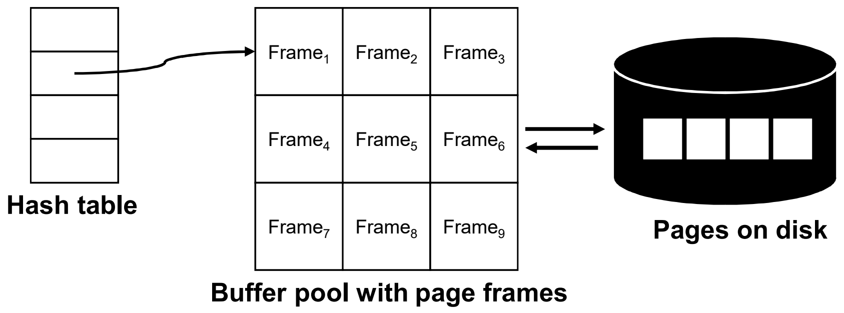


Figure 3.1: Buffer pool

3.1.1 Organization for Disk-Based Relational Systems

Traditional relational database systems were built under the assumption that not all data fits into memory, which was indeed a precious resource in the early days of relational systems. The "final resting place" for data in these systems is disk. This means data must be paged into memory (and back) as appropriate during processing.

Database pages are a fixed-sized block usually several Kilobytes in size (e.g., 8KB or larger is typical) and have the same representation both on disk and in memory to avoid translation overheads between representations. Almost all data in the system maps to a page. For instance, database records map to a specific offset and length on a page, while index data structures like B+-trees use page-size nodes.

Paging data to and from disk is handled by the database storage engine. In memory, pages are stored in a shared buffer pool as depicted in Figure 3.1. The buffer pool is an array of page frames, where a frame is the size of a database page. While implementation details differ, most buffer pools use a hash table to map a page identifier to the page's physical location in the buffer pool, along with other information like disk location and other metadata (see [63] for more details on buffer pool implementation). When the storage engine receives a page request, it first performs a lookup in the hash table to determine whether the page resides in the buffer pool. If the page is not in memory, the storage engine issues an I/O to bring the page from disk into the buffer pool. It then returns a pointer to the in-memory page frame.

A buffer pool is an elegant and simple solution for paging that abstracts disk away from other software components in the system (e.g., the query processor). However, as we will see in the next section, buffer pool indirection severely impacts the performance of main-memory database systems.

3.1.2 Data Organization in Main-Memory Systems

The primary home for records in main-memory databases is RAM. These systems are not constrained by the need to page data to and from disk on demand during query processing. It is for this reason that modern main-memory databases avoid page-based indirection through a buffer pool. These systems do not use logical record identifiers of the form `(page id, offset)` as done in traditional database systems. Instead, it is common practice for main-memory to use in-memory pointers for direct access to records.

Performance Benefits of Avoiding Page Indirection

Avoiding buffer pool indirection can improve performance of a database engine by up to an order of magnitude. There are two main reasons for such a large improvement: (1) avoiding indirection to resolve physical record pointers and (2) page-level latch contention in the buffer pool. We expand on both of these overheads below.

Avoiding page-based indirection. Accessing records through a buffer pool leads to unnecessary overhead in a main-memory system. To see why this is, notice from Figure 3.1 that an access to an in-memory buffer pool page requires two layers of indirection to resolve a physical record pointer: (1) accessing the page in the buffer pool through its directory hash table and (2) calculating a pointer to the record memory using its offset within the page's memory block. Avoiding such indirection on every record access is a huge performance win. Experimental evaluation from the Starburst main-memory engine project provided experimental evidence for such a speedup [82]. These experiments showed up to an order of magnitude performance improvement when using physical record pointers within the Starburst engine versus using paged buffer-pool indirection.

Avoiding page latching. Avoiding buffer pool indirection removes overhead associated with page-level latching. For example, accessing pages through a buffer pool requires that the accessor set either a shared page latch in the case of a read, or an exclusive page latch in the case of an update. Manipulating these page-level latches can become a bottleneck. For instance, an update will set a latch variable to mark that it has exclusive access to manipulate page memory. Even a reader using a shared latch will manipulate a latch variable such as a reference count to denote the number of concurrent readers. Updates to latch variables is done using an atomic CPU instruction, which can lead to performance degradation especially on multi-socket machines[1]. In a commercial system, it is not uncommon to have several latch types based an accessor's behavior. For example, Microsoft SQL Server has six latch types [35], meaning a thread could acquire multiple latches for a single page access.

An end-to-end example. To illustrate page indirection and latch overhead, consider a secondary B+-tree traversal in a disk-based system. This operation first requires traversing the secondary index that requires four page lookups in the buffer pool, along with four latch/unlatch operations, and four binary searches. If the secondary index does not contain all the required columns, then a traversal of the primary index may be necessary[2], doubling the cost of the operation. In addition, there may be additional overheads due to transactional locking (depending on the isolation level and concurrency control scheme). Overall, this secondary index traversal is much more expensive when

[1]Commercial systems try to address this bottleneck using partitioned latches (sometimes called superlatches), while useful, this approach does not avoid all contention.

[2]This would be necessary in a system like SQL Server that store primary keys in the secondary index.

compared to a main-memory system such as Hekaton that stores direct pointers to records. In Hekaton, a secondary index lookup involves simply traversing the index to perform four index node searches, avoiding buffer pool and latching overheads completely.

A Historical Overview

Some of the earliest research in main-memory databases from the University of Wisconsin looked at optimizing database operations assuming the database fit comfortably in the buffer pool [37]. This work shed light on a number of performance improvements for memory-resident data. However, even when database pages are comfortably cached in memory, record access still required buffer pool indirection as described above. The MM-DBMS project from the University of Wisconsin [80, 79, 81], and later the IBM Starburst variant [82], was one of the first to explore the use of pointers for direct record access within the database engine. From this point on, most main-memory systems avoided buffer pool indirection as well. Systems like PRISMA/DB [12, 13], TimesTen (and its research predecessor 'Smallbase") [146, 147, 148], and P*Time [28] all used direct record pointers. One notable systems was Dalí from Bell Labs that went to an extreme and allowed applications to have direct shared-memory access to data records. While this technique leads to very fast execution, it is not common in modern main-memory systems.

Organization Choices for Modern Main-Memory Systems

While direct pointer access to records is common amongst modern main-memory systems, they differ in how records are organized in memory. This section summarizes three primary organizational choices. Later in chapter 4 we cover how each of our modern representative systems chooses to organize data.

Partitioning. One of the highest level differences between modern main-memory systems is whether they physically partition the database. H-Store and VoltDB are examples of partitioned systems, while Hekaton, HANA, MemSQL, and Oracle TimesTen are non-partitioned systems.

The advantage of partitioning is that it greatly simplifies thread and transaction-level concurrency issues. For example, partitioned systems such as H-Store run transactions serially within a partition, thereby avoiding transaction concurrency control protocols for single partition transactions. In addition, partitioned systems usually run single-threaded within a partition, for instance, by assigning a partition to single a CPU core. Single-threaded access means internal data structures such as indexes do not need to support multi-threaded updates, which can greatly simplify the database kernel implementation.

The advantage of a non-partitioned approach is that any thread can access and update any record in the database. These systems thus avoid load-balancing issues that might arise due to "hot" partitions, which usually requires workload re-assignment amongst threads and possibly data movement (e.g., across NUMA sockets). Also, non-partitioned systems do not need to coordinate transactions that span partitions, since by default the system allows shared data access to all threads. On the other hand, the complexity of the engine implementation in non-partitioned systems is higher than that of partitioned systems; in effect, non-partitioned systems trade engine implementation complexity for partition management complexity.

Multi-versioning. A second important architectural choice is whether the systems supports multi-versioning of data. While multi-versioning is not a new concept in databases, it has received renewed interest in the scope of main-memory systems. The primary reason is that it enables concurrency control schemes where readers never block writers. This non-blocking behavior leads to less context switch overhead compared to a blocking-based protocols that cause threads to sleep on conflict (e.g., within a lock manager). Since context switching is a large overhead in main-memory systems, such protocols can lead to profound performance improvements. We discuss concurrency control details later in Section 3.3. For example, Hekaton, HyPer, and SAP HANA implement record multi-versioning for this reason. Another use of multi-versioning is to allow for snapshot reads into the past in order to support, for instance, analytic queries on "near" real time data. Both HyPer and SAP HANA support this type of functionality.

Row/Columnar layout. A third organizational choice is whether to store the records in row or columnar format. The row vs column layout tradeoffs have been well studied in the database literature [4], with row-stores being used for update intensive OLTP workloads while column stores are more appropriate for analytics workloads. This rule of thumb does not change much for main-memory systems. Since this survey is primary concerned with operational OLTP systems, the majority of systems we cover opt for row layout to allow for efficient updates. However, as we will see later in Chapter 4, there are notable exceptions. For example, SAP HANA supports an execution mode that allows OLTP workloads to operate directly on data in the column store. In this mode, HANA is willing to sacrifice some OLTP performance (e.g., compared to a pure row-store) for the advantages of having the primary database be stored in column format. HyPer supports a hybrid storage format that clusters frequently accessed columns. Hekaton stores tables in row format but a column store index can be created on a table to speed up analytical queries.

3.2 Indexing

Modern main-memory systems have brought about a renewed interest in high-performance indexing methods. In this section we first summarize index design in traditional database systems. We then explores two important topics when indexes fit entirely in main-memory: (1) optimizing for CPU cache efficiency and (2) multi-core parallelism. Throughout this section, we use B+-trees as an example of a typical index, since all major commercial relational database systems support them.

3.2.1 Indexing in Disk-Based Relational Systems

In a typical disk-based system, indexes map to pages managed by a buffer pool. This allows indexes to be paged to and from disk similar to pages that contain records. Figure 3.2 depicts the organization of a B+-tree in a relational database. While specific node layout might differ between systems (e.g., keys and other metadata), index nodes

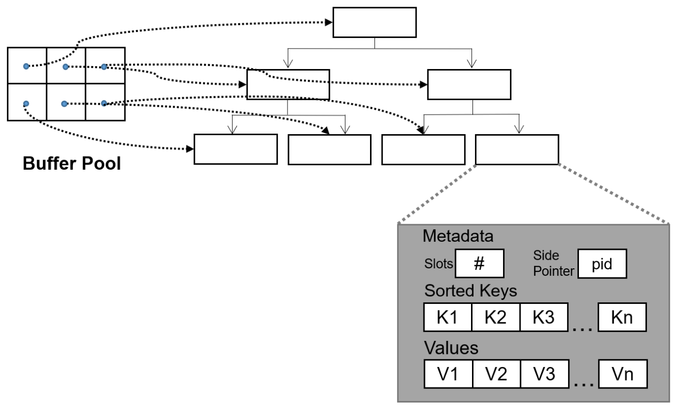


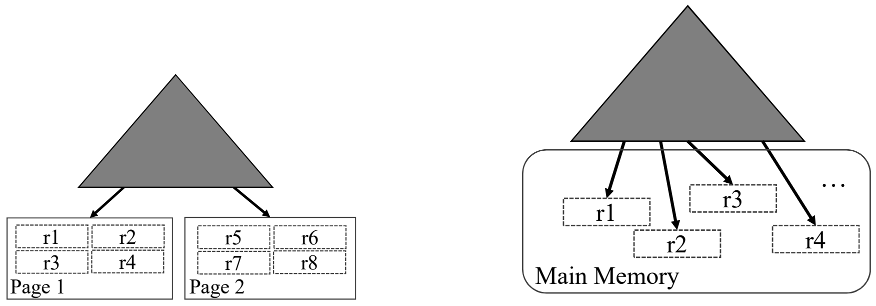
Figure 3.2: Typical B+-tree layout in disk-based RDBMS. Index nodes map to pages managed by the buffer pool.

have the exact size of a page. This means that nodes might be underutilized (i.e., some of the page might be empty). Further, nodes must split when a page becomes full.

Disk-based systems also typically support clustered indexes that store records in the order defined by the index. Clustering ensures that records remain in sorted order on disk, making operations like range scans more efficient by reducing random disk seeks. Only one index per table can be clustered, while the rest must be secondary indexes. Commercial systems differ in how they implement secondary indexes. For instance, Microsoft SQL Server stores the row's primary key in its leaf pages, requiring a second lookup in the primary index to retrieve the full record. Oracle stores both the primary key and direct record pointer in its leaf nodes, using the direct pointer as a "fast path" to find a record, and using the primary key as a slow path in case the record moves [63].

3.2.2 Indexing in Main-Memory Systems

In main-memory databases, index organization is somewhat different from that used by disk-based system. First, there is no page-based indirection as discussed in Section 3.1, therefore index nodes need not



(a) Clustered Index (b) In-Memory Index

Figure 3.3: Differences in disk-based and in-memory index organization.

map to a database page. For example, the ART radix index used in HyPer uses four separate size classes for its index nodes. Likewise, the Bw-tree index used in Hekaton allocates exactly the node memory needed to store its keys and payloads; it does not leave unused space within the node. Furthermore, the Bw-tree uses a flexible split policy, choosing to split B+-tree nodes when convenient, not when node sizes reach a hard threshold as in disk-based indexes.

It is also common for main-memory indexes to store direct pointers to records, rather than logical record ids or primary keys (in the case of secondary indexes) as done in disk-based systems. Figure 3.3 provides a visual example of this difference. Figure 3.3(a) depicts a disk-based clustered index, where the index points to data (leaf) pages that store records in sorted order. On the other hand, Figure 3.3(b) depicts the main-memory approach that indexes records sitting in memory without any particular clustering or organization.

Cache Awareness

Cache-aware indexing structures have been a topic of interest for quite some time in both research and practice. The reason for this is the chasm between CPU and memory speeds. Starting in the 1990s, CPU speeds at the time were increasing 60% per year, while memory speeds improved only 10% per year. Not surprisingly, cache miss stalls started

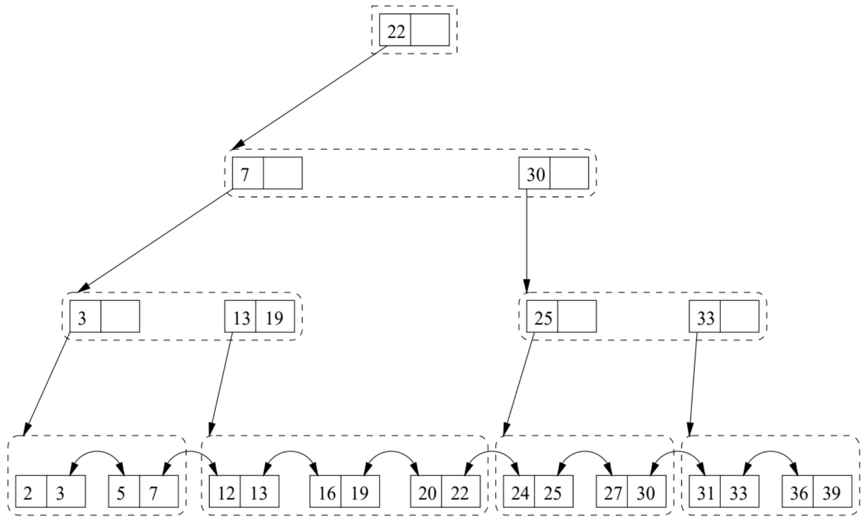


Figure 3.4: Example of the CSB+-Tree (taken from [127].

to show up as a major bottleneck on memory-bound database workloads, with a significant portion of time spent servicing last-level cache misses [8, 113]. There is still a large CPU/memory performance gap today.

To address this bottleneck, indexing research focused on how to optimize the use of a cache line, which is the "unit of transfer" in a main-memory system. Techniques like the CSB+-Tree [127] (depicted in Figure 3.4) used cache line-sized B+-tree nodes that removed as many search pointers as possible in order to make space for keys. To deal with removal of search pointers, the CSB+-tree used "node groups" that packed several nodes together in the same memory block. The parent node stores one or more pointers to child node groups (the number of pointers/node groups is configurable), and the child nodes are accessed by using an offset into the node group.

Cache line prefetching was also explored in the context of main-memory indexing. The Pb+-Tree [30] explored prefetching of B+-tree nodes in order to make searches and scans more efficient, as well as allow node sizes larger than a cache line. At the time, this technique showed a 1.5x speedup for B+-tree searches. Scan performance improved by 6x, however it required an additional structure that stored pointers to future nodes in the scan.

Multi-core Parallelism

While cache-efficient indexing remains an important issue today, multi-core scalability has become equally important. Today, main-memory databases run on CPUs with a staggering amount of parallelism, with each generation of processor increasing the core count even more. For example, today, it is common to have 16 to 32 processor cores on a server spanning multiple sockets, while high-end machines have up to 144 cores[3]. Attention to parallelism is especially important for OLTP systems, since indexing is the hot path for updates and retrieval.

In general, there are two ways to achieve high concurrency in main-memory indexing structures.

Partitioned. This approach partitions the indexing structure and assigns a single thread to each partition. Each thread has exclusive access to its partition. Since threads do not interfere with each other, the index implementation remains relatively simple, since thread-level concurrency is not an issue. Techniques like PLP [119] that use a multi-rooted B+-tree (with partitions defined by each root) as well as H-Store are representative of this approach. While the partitioned approach is conceptually easy to implement, the drawback with the approach is that skew in the workload may cause some partitions to become hot spots, which limits the throughput of the system. Thus rebalancing is required to remove the hot spots.

Shared. A shared approach allows any thread to read or update any part of the index. The difficulty with a high-performance shared approach is maintaining high concurrency and scalability of the index. The key to achieving high concurrency is to avoid (as much as possible) critical sections that provide mutual exclusion by blocking other threads. To address this issue there has recently been work on “latch free” (lock-free) indexing structures that use atomic CPU prim-

[3]The Lenovo x3950 can be configured with 12TB of RAM and 144 cores.

itives (e.g., compare and swap, fetch and increment) to update index state. The advantage of shared indexing methods is that they are self-balancing: any thread can access any part of the index. The draw-back is implementation complexity, especially for latch-free approaches that cannot rely on critical sections. The Bw-tree used in Hekaton and skiplists use by MemSQL are examples of highly concurrent latch-free indexes used in practice. The Mass-Tree [97] is also representative of state-of-the-art work in this area. We cover the details of these ap-proaches in later chapters.

While there are several latch-free index implementations being used in practice, recent research has started looking at the pros and cons of latch-free synchronization. Failero and Abadi analyzed several latched and latch-free synchronization techniques and performed microbench-marks to experimentally verify the relative advantages of each [44]. They found the benefit of latch-free synchronization to be nuanced. Latch-free techniques lead to high performance, but require careful memory reclamation protocols. The study also found that one of the most important factors to scalability on modern CPUs is avoidance of contention on global memory locations, regardless of whether the synchronization approach is latch-based or latch-free.

3.3 Concurrency Control

Concurrency control is a long-standing topic in database research, going back to the early days of relational databases [55]. There have been a number of books written on the subject [20, 57, 155] along with several comprehensive surveys [18, 19]. Main-memory databases, however, have caused a reexamination of the concurrency control landscape with the goal of achieving high performance on modern hardware.

At a high level, there are two dimensions in the concurrency control design space:

- *Pessimistic vs. Optimistic*: Pessimistic methods detect concur-rency conflicts during transaction execution and either abort the transaction (causing a restart) or block transaction execution un-til the conflict clears. Optimistic methods allow a transaction to execute without blocking. During commit, these methods require a validation step to detect (and abort/rollback) transactions with concurrency conflicts.

- *Single-Version vs. Multi-Version*: Single-version methods maintain only one version of a record. Records are updated in place, and there is only one version available to readers at any point in time. Thus, readers may block writers. Multi-versioned methods create new record versions on every update and allow readers to read old versions and thus avoid blocking writers.

A number of early studies explored the tradeoffs of different concurrency control techniques. Many of these studies led to conflicting results. For example, some comparisons suggested pessimistic blocking-based approaches outperformed pessimistic approaches that restarted transactions [7, 25], while other studies suggested the opposite [145]. In other studies, optimistic concurrency schemes were shown to outperform pessimistic locking-based approaches [46], while the opposite was found in other comparisons [7, 25]. One of the more comprehensive comparisons done by Agrawal et al [6] clarified many of these conflicting results as a result of assumptions on computing resources, and suggested that optimistic approaches perform very well when conflict rates are low, but performance degrades (due to rollback and restart) for higher conflict rates.

In general, many results of these early studies still hold, such as the fundamental finding on the effect of contention on optimistic approaches. However, due to the increase in parallelism and the lack of disk overhead in modern main-memory environments, many system architectures have taken a fresh look at the concurrency control design space. In the rest of this section, we first review the two-phase locking approach commonly used in disk-based systems. We then provide an overview of techniques used in main-memory systems. As we will see, a common theme in modern systems is exploit parallelism by using partitioned serial execution, or to use some form of optimistic multi-versioning. However, other recent work shows that in some cases, single-versioned pessimistic approaches also work well.

3.3.1 Concurrency Control in Disk-Based Systems

Commercial disk-based relational databases all use a form of two-phase locking (2PL). 2PL is pessimistic, and requires all readers obtain a shared read lock before reading a record, and an exclusive lock when writing to a record. Locks can be taken on larger-granularity items (e.g., a group of records on a page, or an entire table) to reduce the overhead of acquiring individual locks on many records. If a transaction cannot acquire a lock, it blocks until the lock is granted. As a transaction executes, it acquires locks and holds them until the end of the transaction (the first phase) and releases the locks all at once upon termination (the second phase).

Most commercial disk-based systems also support multi-versioning and to a lesser extent optimistic methods. Both are usually supported as add-ons to 2PL [63]. For instance, SQL Server uses multi-versioning to support snapshot isolation, though versions are temporary and not recoverable. We focus solely on 2PL for the remainder of this section.

The lock manager in a relational engine is responsible for implementing the 2PL protocol. When requesting a lock, the transaction provides the lock manager with its transaction id, a resource id (a unique identifier of the item to lock), and the lock mode it is requesting (see [57] for details). The lock manager either grants the request, allowing the transaction to proceed, or blocks the transaction, restarting it once the resource becomes available.[4]

While lock manager implementations differ by system, it is common to use two main data structures to handle lock bookkeeping: (1) A lock table is the structure for managing resources and detecting lock conflict. It is usually implemented as a hash table keyed on resource id with entries that hold lock metadata such as the mode and a wait queue of blocked transactions. (2) A transaction table holds metadata about transactions executing in the DBMS. The table stores information like thread state (e.g., for resuming a transaction if it is blocked) and a list of all locks held by a transaction so it can release all its locks upon transaction termination. While we do not provide detail of 2PL

[4]Most lock managers also provide non-blocking requests to check if a lock is available.

implementations (see [63] for more detail), the high-level idea is that all major disk-based systems use a lock manager for concurrency control.

3.3.2 Concurrency Control in Main-Memory Systems

A common trend across main-memory databases is to avoid implementing a separate lock manager. This is done for performance reasons. A lock manager is a large monolithic software subsystem that acts as the "traffic cop" for concurrency control. A transaction must take a round trip through the lock manager's lock table at least once before accessing a record. This indirection might be tolerable in a disk-based system due to large overheads that are present elsewhere in the system (e.g., I/O). In a main-memory system, however, this indirection is a major performance bottleneck. Thus, a major trend in main-memory systems is to embed concurrency control metadata in the records themselves. Chapter 4 will cover implementation details of several approaches in practice. In the rest of this section we summarize the general design space for concurrency control in main-memory systems.

Multi-versioning and Optimism. A common approach in several commercial and research systems is to implement multi-version concurrency control. Systems like Hekaton, HANA, and HyPer all use some form of multi-versioning. One of the primary reasons to take this approach is to unlock concurrency since readers never block writers. While concurrency is an advantage, a disadvantage of multi-versioning is overhead from (a) new version creation during an update and (b) the need for some form of garbage collection to remove obsolete versions.

Within the multi-version design space, systems tend to skew toward using *optimistic* approaches, but there are exceptions. Hekaton and HyPer both use a form of optimistic multi-version concurrency control. The advantage of using optimism is that it is much cheaper than traditional locking. It also scales to a high number of cores, since threads simply execute unblocked until transaction completion, avoiding overheads like context switching present pessimistic lock-based approaches. The fundamental disadvantage of optimism – as noted previously – is that high contention may cause high abort rates, causing transactions to roll back and restart several times [6]. SAP HANA opts for a pessimistic multi-versioned approach by using row-level write locks.

Partitioning. Systems that opt for partitioning assign a disjoint piece of the database by core or machines if the database size is sufficiently large. Within a partition, transactions execute serially. This approach is simple: since transactions execute one after the other, neither locks nor validation are needed. This leads to very fast execution within a single partition, and leads to great overall system performance when transactions touch a single partition. However, when a transaction spans multiple partitions, it must exclusively lock all partitions it accesses for the duration of the transaction. This can lead to reduced and unpredictable performance.

H-Store/VoltDB and Calvin [150] are examples systems that use the partitioning approach. H-Store protects each partition with a single exclusive lock. Before accessing a partition, a transaction must acquire the partition lock. A transaction will abort if it discovers that it needs to access another partition (e.g., due to a secondary index lookup) and restarts once it has acquired all locks on the required partition(s). Calvin is a system that can be layered on top of a partitioned non-transactional storage system (main-memory or disk-based storage). It implements a deterministic locking protocol that eliminates the overhead typical in distributed commit protocols.

Apriori knowledge of write/read sets. If the system can determine the read and write set of a transaction before execution (e.g., using static analysis) a number of optimizations can be made to the concurrency control protocol. For example, the deterministic locking protocol in Calvin relies on apriori knowledge of read and write sets in order to determine a serialization order. In addition, a number of pessimistic concurrency control techniques for main-memory systems rely on apriori write set knowledge. Very lightweight logging [130] is a lock-based approach that stores lock metadata (e.g., counts for exclusive and shared locks) directly within records. A transaction requests all of its locks before transaction execution begins, and starts executing only when all locks have been granted. In addition, BOHM [43] is a pessimistic multi-version concurrency control protocol that determines the serialization order of transactions prior to execution. BOHM then creates a record placeholder in memory for the new version that the transaction will write. This approach avoids using a centralized timestamp generator along with a validation phase common in optimistic multi-versioned approaches.

3.4 Durability and Recovery

Durability and recovery techniques ensure a system can recover to a consistent state after a crash. By far, the dominant crash-recovery approach used by disk-based systems is the ARIES [102] logging and checkpointing protocol. In general, durability and recovery in main-memory systems is still based on logging and checkpointing. However, when examining the details, the difference between disk-based and main-memory systems is vast. Many main-memory systems avoid using textbook ARIES-style protocols for performance reasons. In the rest of this section, we briefly cover the ARIES-style approach used by disk-based systems. We then highlight the most notable areas where main-memory systems differ. Chapter 4 covers the details of what specific systems do in practice.

3.4.1 Durability and Recovery in Disk-Based Systems

Disk-based relational databases all use a form of the ARIES [102] protocol to ensure durability of committed transactions and to recover to a consistent state. As we discussed previously, disk-based systems update database pages in place within the buffer pool. The ARIES protocol uses a write-ahead logging (WAL) scheme that writes each page modification to a sequential log before the page is written back to disk. Each log record is given a log sequence number (LSN) that determines its position in the log, and a log record cannot be flushed to disk until all previous records with lower LSNs are also on disk. Before externally acknowledging the success of a transaction to the user, a commit record is written to the log and flushed, ensuring the effects of a transaction will succeed even across failures.

Log records contain enough information to redo and undo the effects of an update. A before image of the update byte sequence on a page (or logical description of how to undo the page update), is necessary to roll back an update in case a transaction aborts. The redo information is necessary to apply the update to a page during recovery.

For performance, databases try to keep log records as small as possible in order to increase I/O throughput to durable storage. There are a variety of approaches to transaction logging in DBMSs [59, 62, 68, 96]. At one extreme, there is *physical logging*, where the DBMS records the before and after image of an element in the database (e.g., a tuple, block, or internal data structure for an index) being modified by the transaction [59, 103].

Another approach is known as *logical logging* (sometimes called *event logging*) where the DBMS only records the high-level operation that the transaction executed (e.g., an update). Logical logging reduces the amount of data that needs to be written to disk compared to physical logging, since it only needs to record what the operation was rather than what it actually did to the database. But recovering the database using logical logging will take longer because the DBMS will need to re-execute each operation. It is common to use physical or physiological logging (e.g., describing the logical operation on a particular page) to support redo, and use logical logging to describe undo of an update. Disk-based systems also log updates to index structures, such as B+-tree or heap structure modifications. This allows for fast recovery of the indexes during recovery.

To support fast recovery, relational databases periodically checkpoint the database by asynchronously flushing pages in the buffer pool back to disk. This allows for truncation of the write-ahead log, and allows recovery to start at a point of history that reduces the amount of redo (and undo) steps to bring the database back to a consistent state. Details of recovery are outside of our scope (see ARIES [102] for a complete and thorough discussion). However, for the purposes of comparison, we note that disk-based systems typically use some flavor of the ARIES protocol and checkpoint by writing pages back to disk.

3.4.2 Durability and Recovery in Main-Memory Systems

Logging in main-memory systems is optimized for high throughput and low latency. Since log I/O is the major bottleneck, these systems attempt to reduce log volume as much as possible, even more than disk-based systems. Hekaton, for example, performs redo-only logging by writing the newest record version for only transactions guaranteed to commit [38]. This approach avoids writing undo information altogether. H-Store/VoltDB uses a slimmed-down variant of logical logging called *command logging* [96] that logs only the transaction invocation request. The log record for this scheme consists solely of the stored procedure name along with its input parameters and the transaction id.

To further minimizing log traffic, several systems also avoid logging index data altogether. Only updates to base data are logged. After a crash, indexes are built from scratch after recovery of the base records. In addition, main-memory systems try to parallelize log I/O as much as possible. For example, since Hekaton uses a multi-version concurrency control scheme, it can easily parallelize log traffic across several I/O devices. This is also the approach taken in the Silo recovery prototype [159]. For performance reasons, SSD is the log device of choice for almost all systems.

Checkpointing is also different in main-memory systems. It is common for main-memory checkpoints to be larger than their counterparts in disk-based systems; this is because main-memory systems usually write all (or most) rows to storage. For example, the checkpointing approach in Hekaton constantly scans the tail of the log and writes new record versions to versioned checkpoint files. Deletes on the log are written as deltas on top of existing checkpoint files to signify the "tombstone" records in the file. Other lightweight main-memory checkpointing techniques have been created, such as the CALC [129] non-blocking asynchronous approach that goes into a temporary copy-on-write mode, writing only record updates that have occurred since the previous checkpoint. H-Store/VoltDB also uses a similar copy-on-write scheme to take asynchronous snapshots of the data on each node.

The recovery process in main-memory systems differs from the typical ARIES-style "analysis, redo, undo" sequence used by disk-based systems. Most systems recover by loading the last valid checkpoint, then replay the tail of the log, avoiding undo entirely. For example, since Hekaton only logs redo information for updates from committed transactions, it recreates an initial database from its checkpoint files. It then replays the relevant portion of the log before going online. Likewise, H-Store/VoltDB recovers by loading the last full consistent snapshot. It then replays transactions from its command log (starting from the last snapshot time) to bring the database to a consistent state.

In a main-memory systems, the entire database must be loaded into memory before going online; the database cannot partially exist on storage like in a disk-based system. Therefore, recovery time is dominated by the time to rebuild the database, along with indexes, using the checkpoint and log. To make this as efficient as possible, several systems attempt to parallelize recovery as much as possible. Hekaton, for example, attempts to maximize I/O bandwidth during recovery by using multiple log storage devices. Silo, which uses value logging, also implements a highly parallel log replay scheme aimed at avoiding unnecessary allocations and replay work.

3.5 Query Processing and Compilation

The query processing component of a database system encompasses many tasks, from the "upper" layers that perform authorization, parsing, and query optimization, to the "lower" layers that execute the physical query plan. When comparing a main-memory system with a disk-based system, the difference in the upper layer of a query processor is not large: the implementation of parsing and query optimization remain similar. There is a vast difference in the lower layer of the query processing stack. Main-memory systems reduce runtime overhead by compiling queries directly to machine code. These queries execute directly against in-memory data. Disk-based systems, on the other hand, execute queries using an iterator model and often use runtime interpretation to perform operations like predicate evaluation.

This section summarizes the differences between disk-based and main-memory systems in the lower layer of the query execution stack.

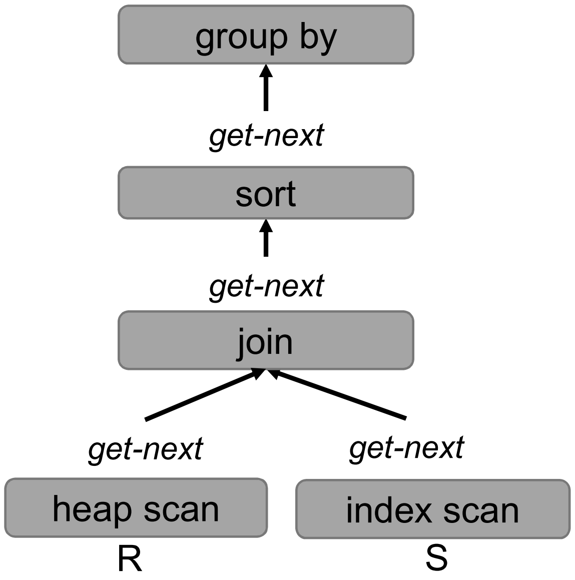


Figure 3.5: Query plan

We first cover how disk-based systems implement the iterator model. We then discuss the query compilation and execution strategies in modern main-memory systems.

3.5.1 Query Execution in Disk-Based Systems

After query rewrite and optimization, the query processor is responsible for executing a the physical query plan. The query plan is essentially a data flow graph consisting of various query operators. Figure 3.5 provides an example query plan consisting of scan, join, sort, and group-by operators.

Queries execute using an iterator model (also called Volcano-style processing [54]). Each operator implements a generic interface, consisting of the following basic functions (1) `init` that initializes the operator, (2) `get-next` that instructs the operator to produce the next tuple and return it to the caller, and (3) `close` that informs the operator to uninitialize. Each operator takes as input one or more iterators (depending on the semantics of the operation). While simple, this interface is incredibly powerful as it allows for an arbitrary combination of operators without special-purpose "glue" code.

A `get-next` call to the root operator of the query plan produces a tuple satisfying the query. To produce this tuple, the root operator makes a `get-next` call to its input operators, and so on down the plan tree. This is often referred to as a pull-based operator model, where tuples flow from the base operators to the root based on the initial request from the root. See [53] for an extensive survey on query processing and the iterator model.

Due to the generic nature of the iterator model, operator implementations are general-purpose. This means that the operator code must be written to handle a large number of scenarios. One primary example is handling data types: relational operators *interpret* the bytes within a tuple at runtime, casting them to the appropriate data type and performing runtime error checks as necessary. This avoids the need to compile an operator for each possible combination of data type that it might encounter.

3.5.2 Query Execution in Main-Memory Systems

While the iterator model served disk-based systems well, it comes from an era where I/O was the dominant overhead. When used in a main-memory database, the iterator approach introduces several inefficiencies. For example, the query invokes the `get-next` call for each tuple produced in the query pipeline, meaning this function might be called millions of times. On top of this, the `get-next` function is usually virtual or called via a function pointer, meaning it is more expensive than a vanilla function call and can degrade branch prediction on modern CPUs. In addition, due to its generic nature, the iterator model often leads to poor code locality. For example, while processing each tuple, the operator must call another function to *interpret* the bytes within the tuple. In more complex cases, e.g., a table scan on a compressed relation, the operator must invoke code per-tuple for both book-keeping and decompression.

Needless to say, this indirection and poor code locality adds up to unnecessary overhead in a main-memory system. As we will see in Chapter 4, modern main-memory databases avoid iterator-style processing altogether. Instead, these systems aggressively compile queries and transactions into machine code to avoid interpretation, avoid unnecessary runtime overhead, and make efficient use of the CPU cache hierarchy.

As we will see, each system differs slightly in how they compile queries. HIQUE [73], one of the earlier compilation works in the modern main-memory database era, proposed compiling a query into C using code templates for each operator. HIQUE eliminates the iterator model by inlining result materialization inside the operator execution. Hekaton compiles both queries and table definition data using the Microsoft C compiler. Compiling table definitions allows data types to be known at compile time and also allows inter-operation with the SQL Server interpreted query executor. HyPer maximizes data locality by trying to keep record attributes in CPU registers as long as possible. This is done by compiling large pipeline fragments that end pipeline breakers: materialization points that require tuples to spill from registers. HyPer makes use of LLVM, where operators are partially implemented in C++ while performance-sensitive and query-specific logic is generated.

We end this section by noting that query compilation is not a new topic in database systems. In fact, the original System R prototype compiled queries directly to machine code. However, as noted in [63], when commercialized as SQL/DS, the System R query executor was changed to use an interpreter. Since then, interpreted query execution was the norm in disk-based database systems.

4

Systems

In this section, we take an in-depth look into four modern main-memory database systems. Each system is distinct in its design and approach to addressing the issues just discussed in Chapter 3. We begin in Section 4.1 by describing Hekaton, Microsoft SQL Server's main-memory OLTP engine. We then cover H-Store and its commercial offshoot VoltDB in Section 4.2. Section 4.3 describes the HyPer system from TU-Munich, while Section 4.4 covers SAP HANA. Finally, we conclude this Chapter by summarizing the design of other modern commercial and academic main-memory systems in Section 4.5.

4.1 SQL Server Hekaton

4.1.1 Introduction

Hekaton is a database engine optimized for memory resident data integrated into SQL Server. The official name of the feature is "In-Memory OLTP" but it is more commonly called Hekaton. Exploration and prototyping began in 2009 and the initial release was in SQL Server 2014. Functionality was significantly expanded in SQL Server 2016 with, for example, improved support for real-time analytics by the addition of column store indexes also on Hekaton tables.

A Hekaton table, more precisely, a table managed by Hekaton, is stored entirely in memory and can have several hash indexes and/or range indexes plus at most one column store index. Tables are durable and transactional, though non-durable tables are also supported. They are accessed using T-SQL in the same way as disk-based tables. A query can reference both Hekaton tables and disk-based tables and a transaction can update both types of tables. A T-SQL stored procedure that references only Hekaton tables can be compiled into native machine code to further improve performance.

Hekaton is targeted primarily for OLTP applications and designed for high levels of concurrency. Data is not partitioned - any thread can access any row in a table. The engine uses latch-free data structures to avoid physical interference among threads and a new optimistic, multiversion concurrency control technique to reduce interference among transactions.

A database can contain both in-memory tables and disk-based tables; only the most performance-critical tables need to be in main memory. This allows gradual adoption of the new technology, one table and one stored procedure at a time.

Architectural Principles

The design of Hekaton was guided by three architectural principles, all aimed at achieving low latency and high throughput on transactional workloads.

- **Optimize data structures for main memory.** Hekaton tables and indexes live entirely in memory. Rows are immutable - every updates creates a new version. This makes it possible to use data structures, in particular, indexes that are optimized for main memory. For example, a reference to a row in an index can be a direct physical pointer. There is no need for a buffer pool so the associated overhead and complexity is avoided entirely.

- **Non-blocking execution.** Achieving good scaling on multicore systems is critical for high throughput. Scalability suffers when the systems has shared data structures that are updated at high rate such as latches and spinlocks or highly contended resources such as the lock manager. All Hekaton's internal data structures, for example, memory allocators, indexes, and the transaction map, are entirely latch-free (lock-free). There are no latches or spinlocks on any performance-critical paths in the system. Hekaton uses optimistic multi-version concurrency control so there are no locks and no lock table. The result is in a system where threads execute transactions without stalling or waiting.

- **Compile requests to native code.** To maximize run time performance stored procedures that access only Hekaton tables can be compiled into customized, highly efficient machine code.

4.1.2 Data Organization

A Hekaton table can have three types of indexes: hash indexes which are implemented using lock-free hash tables, range indexes which are implemented using Bw-trees [89], and at most one column store index. Rows are accessed via index lookups, index range scans, column store scans, or heap scans (physical scan of the memory areas storing a table). Hekaton uses multi-versioning; an update always creates a new version.

Figure 4.1 shows a simple bank account table containing six row versions. Ignore the numbers (100) and text in red for now. The table has three (user defined) columns: Name, City and Amount. A version includes a header and a number of link (pointer) fields. A version's valid time is defined by timestamps stored in the Begin and End fields in the header.

The example table has two indexes, a hash index on Name and a range index on City. Each index requires a link field in the row. The first link field is reserved for the Name index and the second link field for the City index. For illustration purposes we assume that the hash function just picks the first letter of the name. Versions that hash to the same bucket are linked together using the first link field. The leaf nodes of the Bw-tree store keys and pointers to records. If multiple rows have the same key value, the duplicates are linked together using

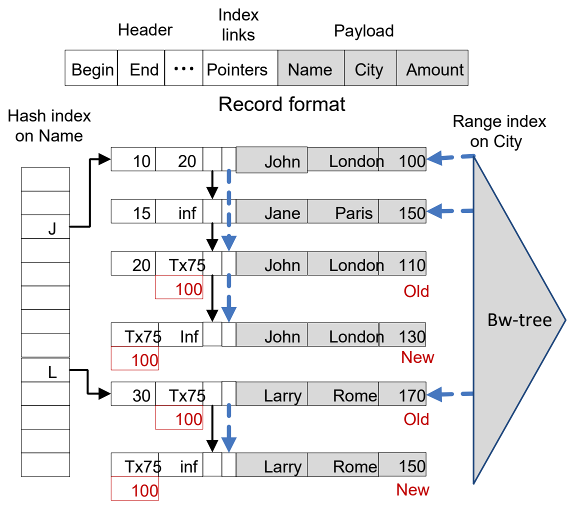


Figure 4.1: Record and index structures

the second link field and the Bw-tree points to the first row on the chain.

Hash bucket J contains four records: three versions for John and one version for Jane. Jane's single version (Jane, Paris, 150) has a valid time from 15 to infinity meaning that it was created by a transaction that committed at time 15 and is still valid. John's oldest version (John, London, 100) was valid from time 10 to time 20 when it was updated. The update created a new version (John, London, 110). We will discuss John's last version (John, London, 130) in a moment.

Reads

Every read operation specifies a logical (as-of) read time and only versions whose valid time overlaps the read time are visible to the read; all other versions are ignored. Different versions of a row have non-overlapping valid times so at most one version of a row is visible to a read. A lookup for John with read time 15, for example, would trigger a scan of bucket J that checks every version in the bucket but returns only the one with Name equal to John and valid time 10 to 20.

Updates

Bucket L contains two records that belong to Larry. Transaction 75 is in the process of transferring $20 from Larry's account to John's account. It has created the new versions for Larry (Larry, Rome, 150) and for John (John, London, 130) and inserted them into the two indexes.

Note that transaction 75 has stored its transaction Id in the Begin and End fields of the new and old versions, respectively. One bit in the field indicates the field's content type. A transaction Id stored in the End field prevents other transactions from updating the same version and it also identifies which transaction is updating the version. A transaction Id stored in the Begin field informs readers that the version may not yet be committed and identifies which transaction created the version.

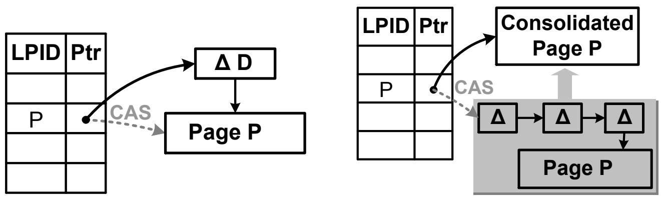
Now suppose transaction 75 commits with end timestamp 100. After committing, transaction 75 returns to the old and new versions and sets the Begin and End fields, respectively, to 100. The final values are shown in red below the old and new versions. The old version (John, London, 110) now has the valid time 20 to 100 and the new version (John, London, 130) has a valid time from 100 to infinity. Larry's record is updated in the same way.

Multi-versioning improves scalability because readers no longer block writers. Writers may still conflict with writers though. Read-only transactions have little effect on update activity; they simply read older versions of rows as needed. Multi-versioning also speeds up query processing by reducing copying of rows. Since a version is never modified it is safe to pass around a pointer to it instead of making a copy.

The system must discard obsolete versions that are no longer needed to avoid filling up memory. A version can be discarded when it is no longer visible to any active transactions. Cleaning out obsolete versions, a.k.a. garbage collection, is handled cooperatively and continuously by all worker threads [38].

4.1.3 Indexing

As previously mentioned, all data structures in the Hekaton engine (including indexes) are completely latch-free. Hekaton currently uses



(a) Delta updates (b) Page consolidation

Figure 4.2: Latch-free delta updates and consolidation.

statically-sized hash tables with overflow buckets implemented as latch-free lists. Its range index is a latch-free B+-tree (the Bw-tree) [89]. The key to latch-freedom in the Bw-tree is the use of a mapping table that maps logical B+-tree page identifiers (PIDs) to physical page memory. The mapping table is the central location for managing the “paginated" tree. All links between Bw-tree pages are PIDs, not physical pointers. The mapping table enables the physical location of a Bw-tree node page to change on every update without requiring that the location change propagate to the root of the tree, because inter-page links are PIDs that do not change.

The Bw-tree performs page updates via copy-on-write, not via update-in-place (updating the existing page memory). Avoiding update-in-place reduces CPU cache invalidation, which is especially important on multi-socket machines. Reducing cache misses also increases the instructions executed per cycle. A delta record describes the change of a single record on a page P (e.g., insert, update, delete). This delta physically points to P. The update installs the (new) memory address of the delta record into the P’s slot in the mapping table using a compare-and-swap (CAS) instruction. If the CAS succeeds, the delta record’s virtual memory address becomes the new physical “root" address of the page, thus successfully updating the page. Delta updating simultaneously enables latch-free access in the Bw-tree and preserves processor data caches by avoiding update-in-place. Figure 4.2(a) depicts a delta update record D prepended to page P; the dashed line represents P’s original address, while the solid line to D represents P’s new address. We occasionally consolidate pages by creating a new page

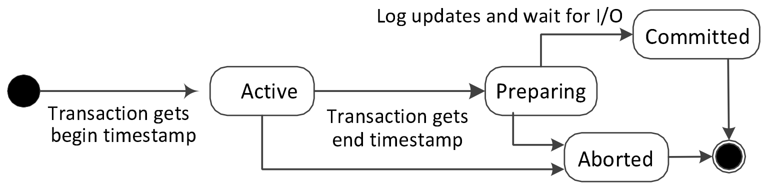

Log updates and wait for I/O
Committed
Active
Preparing
Aborted
Transaction gets begin timestamp
Transaction gets end timestamp

Figure 4.3: Transaction state transitions

that applies all delta changes to a search optimized base page. This reduces memory footprint and improves search performance. A consolidated form of the page is also installed with a CAS, as depicted in Figure 4.2(b) showing the consolidation of page P with its deltas into a new "Consolidated Page P". B+-tree structure modifications (page splits and deletes) are also performed in a latch-free manner.

4.1.4 Concurrency Control

Hekaton uses optimistic concurrency control to provide transaction isolation; there are no locks and no lock table [77]. Pessimistic concurrency control prevents conflicts by locking. Optimistic concurrency control does not attempt to prevent conflicts but instead detects when a conflict has occurred by validating an update transaction's reads before commit. If validation fails the transaction aborts.

A transaction can be in one of four states: Active, Preparing, Committed, or Aborted. Figure 4.3 shows the possible transitions between states. A transaction goes through three different phases during its lifetime.

1. The transaction is created; it acquires a begin timestamp and sets its state to Active.

2. **Normal processing phase.** The transaction does all its normal processing during this phase. A transaction never blocks during this phase. For update operations, the transaction copies its transaction ID into the Begin field of the new versions and into the End field of the old or deleted versions. If it aborts, it changes

its state to Aborted and skips directly to step 4. When the transaction has completed its normal processing and requests to commit, it acquires an end timestamp and switches to the Preparing state.

3. **Preparation phase.** During this phase the transaction performs validation to determine whether it can commit or is forced to abort. If it has to abort, it switches its state to Aborted and continues to the next phase. If it is ready to commit, it writes all its new versions and information about deleted versions to a redo log record and waits for the log record to reach stable storage. The transaction then switches its state to Committed.

4. **Postprocessing phase.** If the transaction has committed, it proceeds to replace its transaction ID with its end timestamp from the Begin field of the new versions and from the End field of the old or deleted versions. If the transaction has aborted, it marks all its new versions as garbage.

5. The transaction is now terminated. When the transaction's old versions are no longer visible to any active transaction, they assigned to the garbage collector, which is responsible for physically deleting them.

Timestamps are drawn from a global, monotonically increasing counter. A transaction gets a unique end timestamp by atomically reading and incrementing the counter.

The extent of validation in the preparation phase depends on the transaction's isolation level. Read-only transactions, regardless of isolation level, and update transactions running under snapshot isolation require no validation at all. Write-write conflicts are detected immediately when a transaction attempts to update a version and result in the transaction rolling back.

Transactions running under repeatable read or serializable isolation require validation before commit. During validation a transaction T checks whether the following two properties hold.

- **Read stability.** If T read some version V1 during its processing, we must ensure that V1 is still the version visible to T as of the end of the transaction. This is implemented by validating that V1 has not been updated before T commits. Any update will have modified V1's end timestamp so all that is required is to check V1's end timestamp. To enable this, T retains a pointer to every version that it has read.

- **Phantom avoidance.** For serializable transactions we must also ensure that the transaction's scans would not return additional versions. This is implemented by re-scanning to check for new versions before commit. To enable this, a serializable transaction keeps tracks of all its index scans and retains enough information to be able to repeat each scan.

4.1.5 Query Processing

In-memory tables are accessed using normal T-SQL either through natively compiled stored procedures or through query interop. For maximum speed, stored procedures that access only in-memory tables are compiled into customized, highly efficient machine code. The T-SQL procedure is first converted into C code which is compiled by the Microsoft C compiler producing a DLL that is then loaded into the SQL Server process. The generated code contains exactly what is needed to execute the request, nothing more. As many decisions as possible are made at compile time to reduce runtime overhead. For example, all data types are known at compile time allowing the generation of efficient code.

The classical SQL Server engine can access or update in-memory tables through special operators built for this purpose. There is, for example, an index scan operator for performing lookups or range scans of an index on a Hekaton table. The caller specifies a search key or key range and the operator outputs the qualifying rows with the requested columns in the internal row format used by the classical engine. This interoperation capability is crucial; it is used, for example, for ad-hoc queries, for queries combining data from disk-based tables and in-memory tables, and for queries requiring features not available in natively compiled stored procedures.

4.1.6 Durability and Recovery

Transaction durability is ensured by logging and checkpointing rows to external storage. Only user data, not indexes, are logged. During recovery Hekaton tables and their indexes are rebuilt entirely from the latest checkpoint and the tail of the log.

A transaction that successfully passes validation is ready to commit. At this point it writes to the log all new versions that it has created and keys of all versions it has deleted. This is done in a single write (except for very large transactions) and if the write succeeds, the transaction is irrevocably committed. Aborted transactions are not logged so aborting a transaction is cheap.

Checkpointing

To reduce recovery time Hekaton also implements checkpointing. The checkpointing scheme is designed to satisfy three key requirements.

- **Continuous checkpointing.** Checkpoint related I/O should occur incrementally and continuously as needed to avoid sudden I/O spikes that negatively affect the transactional workload.
- **Sequential I/O.** Checkpointing should rely on sequential I/O rather than random I/O for most of its operations. Even on SSD devices random I/O is slower than sequential and can incur more CPU overhead.
- **Parallel recovery.** Loading data into memory during recovery should be highly parallelizable to fully exploit available I/O bandwidth and minimize recovery time.

Checkpoint data is stored in two types of checkpoint files: data files and delta files. A complete checkpoint consists of multiple pairs of data and delta files. A data file contains all new versions created within a specific timestamp range. Data files are append-only while open and once closed strictly read-only. At recovery time the versions in data files are reloaded into memory and re-indexed, subject to filtering by delta files.

A delta file stores information about which versions contained in its associated data file have been subsequently deleted. Delta files are also append-only. At recovery time, the delta file is used as a filter to avoid reloading deleted versions into memory. The choice to pair one delta file with each data file means that the smallest unit of work for recovery is a data/delta file pair leading to a recovery process that is highly parallelizable.

A complete checkpoint combined with the tail of the transaction log enable Hekaton tables to be recovered. A checkpoint has a timestamp which indicates that the effects of all transactions before the checkpoint timestamp are included in the checkpoint and thus need not be recovered from the transaction log.

A checkpoint task takes a section of the transaction log not covered by a previous checkpoint and converts the log contents into one or more data files and updates to delta files. New versions are appended to either the most recent data file or into a new data file and the IDs of deleted versions are appended to the delta files corresponding to the data file where the original inserted versions are stored.

The set of files involved in a checkpoint grows with each checkpoint but the active content of a data file degrades as more and more of its versions are marked deleted in its delta file. Since crash recovery reads the contents of all data and delta files in the checkpoint, recovery performance degrades as the utility of each data file drops. The avoid this problem temporally adjacent data files are merged when their active content (the percentage of undeleted versions in a data file) drops below a threshold. Merging two data files DF1 and DF2 results in a new data file DF3 covering the combined range of DF1 and DF2. All deleted versions are dropped during the merge so the new delta file is empty immediately after the merge.

4.1.7 Performance and Further Reading

Performance Matters

Main memory databases can provide much higher throughput and lower latency than traditional disk-based database systems. If the performance improvements are sufficiently large this can make a real difference. The following example illustrates one such case.

EdgeNet provides optimized product data for suppliers, retailers, and search engines. To optimize performance under a heavy workload, the application would return product information from a cache instead of querying production databases. Because the company received high volumes of data from multiple sources, it was unable to write directly to the production database without conflicting with read processes and locking transactions. Instead, they received files into a staging environment, where the data was transformed and structured before loading into the databases. It could take a day to prepare and load the information so Edgenet looked for an alternative. They wanted an online transaction processing (OLTP) solution that would provide their customers with real-time access to information.

By switching to Hekaton, EdgeNet could allow read and write activities to run concurrently on the same database, thereby enabling continuous data ingestion. They no longer had to wait a day or even an hour to stage and prepare data. Furthermore, they could also eliminate the front-side caching layer and the staging area for data loading. In summary, the substantial performance gained from switching to an main-memory database enabled both much better customer service and a simplified system configuration.

Further Reading

Reference [38] provides the most comprehensive overview of the design of Hekaton, covering data storage and indexing, concurrency control, query processing and compilation, and logging, checkpointing and recovery. A detailed description of Bw-trees used for range indexes is provided in [89]. The concurrency control algorithms are described in great detail in [77]. The compilation process is also covered in [47].

4.2 H-Store and VoltDB

4.2.1 Introduction

H-Store (and its commercial successor VoltDB) is a DBMS that is designed for efficient execution of modern OLTP workloads. Beyond using main memory as the primary storage location of a database, its architecture is based on four design principles:

Partitioning and Serial Execution: Instead of allowing transactions to execute concurrently, H-Store partitions data and executes transactions serially at each partition [81, 90, 135, 157]. This means that it does not need to employ a heavy-weight concurrency control scheme to manage fine-grained locks. When a transaction executes, it never has to stall because all of the memory that it needs is already in main memory and it will never block waiting to acquire a lock held by another transaction.

Stored Procedure Execution: Rather than sending SQL commands at runtime, the application registers a set of SQL-based procedures with H-Store and only invokes transactions through these procedures. Encapsulating all transaction logic in a stored procedure prevents application stalls mid-transaction and also avoids the overhead of query planning at run-time. Although this scheme requires all transactions to be known in advance, this assumption is reasonable for OLTP applications [142, 96].

Distributed Deployments: In order to support databases that are larger than the amount of memory available on a single node, the H-Store will split databases across shared-nothing [140] compute nodes into disjoint segments called *partitions* [36, 26]. Partitions are replicated across multiple nodes to provide the transaction processing system with high-availability and fault-tolerance.

Compact Logging: To avoid the overhead of a heavy-weight recovery mechanism [102], H-Store uses a lightweight logical logging scheme that only needs to record what transactions were rather than the individual physical changes that it made to the database.

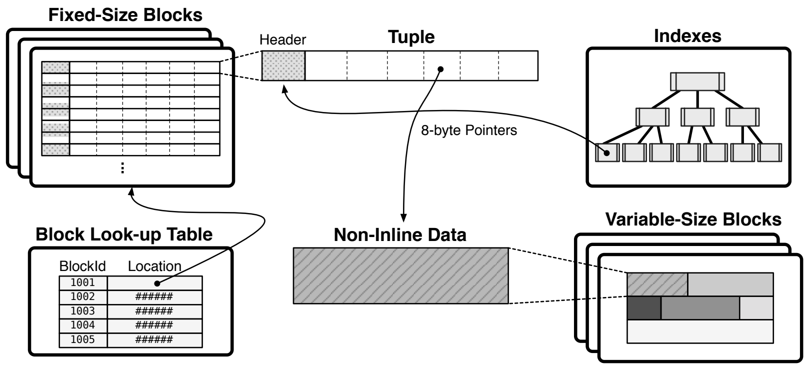


Figure 4.4: An overview of the in-memory storage layer for H-Store.

Over the years, there have been several incarnations of the H-Store system. The initial proof-of-concept was a single-node engine developed at MIT in 2007 that only could execute a simplified version of the TPC-C benchmark [142]. The full-featured, general purpose version of H-Store was developed in 2008 by Brown, MIT, Yale, and Vertica Systems [70]. In 2009, this version of H-Store was forked and commercialized as VoltDB [3]. In 2010, some of the changes from VoltDB were merged back into the original H-Store source code, but the runtime transaction management and coordination subsystems were rewritten. Since then, H-Store's development has continued in academia as research test-bed while VoltDB continued on to develop features that were important for real-world deployments.

H-Store's architecture is divided into two parts: (1) the front-end transaction management component and (2) the back-end query executor and storage manager. The front-end consists of all the networking libraries for communicating with the application's clients, the transaction coordinator, and the stored procedures. This part of the system is written in Java. The back-end C++ execution engine for each partition contains the storage manager, indexes, and query plan executors.

4.2.2 Data Organization

The diagram in Figure 4.4 shows the storage layout for H-Store's tables and indexes. All of the execution engines at a single node operate in the same address space, but their underlying partitions do not share any data structures. Each partition maintains separate indexes for the database tables that only contain entries for the tuples associated with that particular partition. This means that an execution engine is unable to access data stored in another partition at the same node.

The in-memory storage area for tables is split into separate pools for fixed-sized blocks and variable-length blocks. The fixed-size block pool is the primary storage space for the tables' tuples. All tuples are a fixed size (per table) to ensure that they are byte-aligned. Any field in a table that is larger than 8-bytes is stored separately in a variable-length block. The 8-byte memory location of this block is stored in that field's location in the tuple [135]. All other fields that are less than 8-bytes are stored in-line. Each tuple is prefixed with a 1-byte header that contains meta-data on whether a tuple has been modified or has been deleted by the current transaction. This information is used for H-Store's snapshot mechanism (Section 4.2.6).

The DBMS maintains a look-up table of the id numbers of blocks to their corresponding memory location. This look-up table allows the execution engine to reference individual tuples using a 4-byte offset in the table's storage area rather than an 8-byte pointer. That is, from a 4-byte offset the storage layer can compute the id of the block with the tuple and the tuple's location within that block.

The DBMS stores the tables' tuples unsorted within the storage blocks. For each table, the DBMS maintains a list of the 4-byte offsets of unoccupied (i.e., free) tuples. When a transaction deletes a tuple, the offset of the deleted tuple is added to this pool. When a transaction inserts a tuple into a table, the DBMS first checks that table's pool to see if there is an available tuple. If the pool is empty, then the DBMS allocates a new fixed-size block to store the tuple being inserted. The additional tuple offsets that are not needed for this insert operation are added to the table's free tuple pool. H-Store does not compact blocks if a large number of tuples are deleted from a table.[1]

Before a new application can be deployed on H-Store, the administrator has to provide the DBMS's *Project Compiler* with (1) the

[1]Note that VoltDB supports automatic block compaction and reorganization

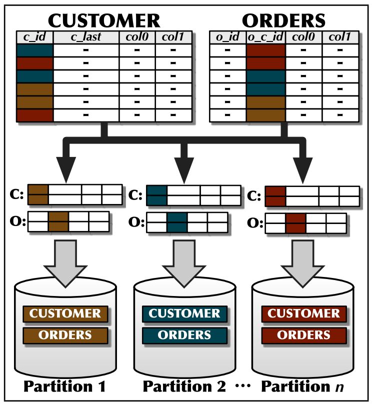
database's schema, (2) the application's stored procedures, and (3) the database's design specification. The compiler will generate a catalog that contains the meta-data for the components in the application's database (e.g., tables, indexes, constraints) and the compiled query plans for each of the application's stored procedures.

Database Design

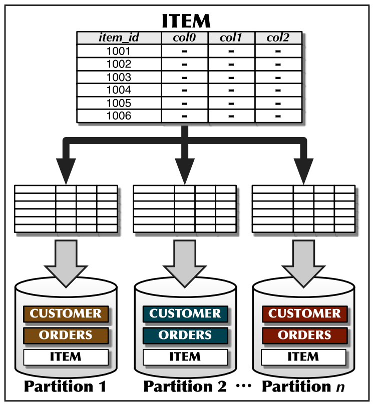
An application's database design specification defines the physical configuration of the database, such as whether to divide a particular table into multiple partitions or to replicate it at every node. This determines the partitions that each transaction will access at runtime. A design determines whether the H-Store executes a transaction request from the application as a fast, single-partition transaction, or as a slow, distributed transaction. That is, if tables are divided amongst the nodes such that a transaction's base partition has all of the data that the transaction needs, then it is single-partitioned. Determining the optimal configuration for an arbitrary application is non-trivial, especially for a complex enterprise application with many dependencies. There is a large amount of research on automatic database design that minimizes distributed transactions and the amount of skew [33, 120].

Table Partitioning. A table can be horizontally divided into multiple, disjoint fragments whose boundaries are based on the values of one (or more) of the table's columns (i.e., the *partitioning attributes*) [160]. The DBMS assigns each tuple to a particular fragment based on the values of these attributes using either range partitioning or hash partitioning. Related fragments from multiple tables are combined together into a partition [51, 118]. Most tables in OLTP applications will be partitioned in this manner. In the example database in Figure 4.5(a), each record in the `CUSTOMER` table has one or more `ORDERS` records. Thus, if both tables are partitioned on their `WAREHOUSE` id (e.g., `CUSTOMER.W_ID` and `ORDERS.W_ID`), then all transactions that only access data within a single warehouse will execute as single-partitioned, regardless of the state of the database.

Table Replication. Instead of splitting a table into multiple partitions, the DBMS can replicate that table across all partitions. Table



(a) Horizontal Partitioning



(b) Table Replication

Figure 4.5: A database design for H-Store consists of the following: (a) splits tables into horizontal partitions and (b) replicates tables on all partitions

replication is useful for read-only or read-mostly tables that are accessed together with other tables but do not share foreign key ancestors. This is different than replicating entire partitions for durability and availability. For example, the read-only `ITEM` table in Figure 4.5(b) does not have a foreign-key relationship with the `CUSTOMER` table. By replicating this table, transactions do not need to retrieve data from a remote partition in order to access it. But any transaction that modifies a replicated table has to be executed as a distributed transaction that locks all of the partitions in the cluster, since those changes must be broadcast to every partition in the cluster. In addition to avoiding additional distributed transactions, one must also consider the space needed to replicate a table at each partition.

4.2.3 Indexing

H-Store supports hash table and B-tree data structures for unique and non-unique indexes. The values of the entries in the indexes are offsets for tuples. Since execution within a partition is single-threaded, the index implementations do not need to be thread-safe. This leads to simpler index implementations and shorter code paths.

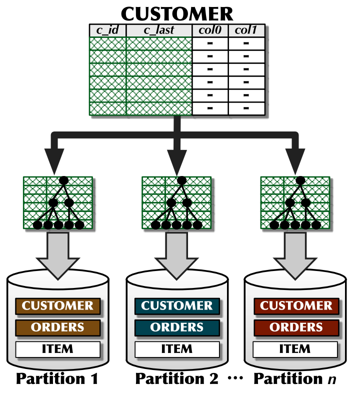


Figure 4.6: Replication of secondary indexes on all partitions.

Secondary Index Replication

When a query accesses a table using a column that is not that table's partitioning attribute, it is broadcast to all partitions. This is because the DBMS does not know what partition has the tuple(s) that the query needs. In some cases, however, these queries can become single-partitioned if the database includes a secondary index for a subset of a table's columns that is replicated across all partitions. Consider a transaction for the database shown in Figure 4.6 that executes a query to retrieve the id of a `CUSTOMER` using their last name. If each partition contains a secondary index with the id and the last name columns, then the DBMS can automatically rewrite the stored procedures' query plans to take advantage of this data structure, thereby making more transactions single-partitioned. This technique only improves performance if the columns chosen in these indexes are not frequently updated.

4.2.4 Concurrency Control

Instead of using a heavy-weight concurrency control scheme where multiple transactions execute simultaneously at a partition [17], H-Store executes transactions one-at-a-time at each partition. That is, when a transaction executes in H-Store, it has exclusive access to the data and indexes at the partitions that it needs. Transactions never stall waiting to acquire a latch held by another transaction because no other transaction will be running at the same time at either its base partition or its remote partitions.

H-Store uses timestamp-based scheduling for transactions [17]. When a transaction request arrives at a node, the coordinator assigns the request a unique identifier based on its arrival timestamp. This id is a composite key comprised of the current wall time at the node (in milliseconds), a counter of the number of transactions that have arrived since the last tick of the wall time clock (in case multiple transactions enter the system at the exact same time), and the transaction's base partition id [149].

Each partition is protected by a single lock managed by its coordinator that is granted to transactions one-at-a-time based on the order of their transaction ids [5, 17, 32, 157]. A transaction acquires a partition's lock if (1) the transaction has the lowest id that is not greater than the one for last transaction that was granted the lock and (2) it has been at least 5 ms since the transaction first entered the system [142]. This wait time ensures that distributed transactions that send their lock acquisition messages over the network to remote partitions are not starved. We assume that the standard clock-skew algorithms are used to keep the various CPU clocks synchronized at each node.

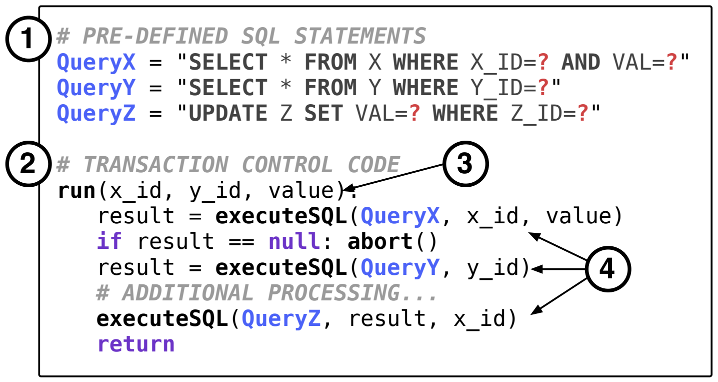
Serializing transactions at each partition in this manner has several advantages for OLTP workloads. In these applications, most transactions only access a single entity in the database at a time (e.g., a transaction that operates on a single customer). That means that if the DBMS will perform significantly faster than a traditional DBMS if the database is partitioned in such a way that most transactions only to access a single partition. Smallbase was an early proponent of this approach [65], and more recent examples include K [157] and Granola [32]. The downside of this approach, however, is that it means transactions that need to access data at two or more partitions are significantly slower. If a transaction attempts to access data at a partition that it does not have the lock for, then the DBMS aborts that transaction (releasing all of the locks that it holds), reverts any changes, and then restarts it once the transaction re-acquires all of the locks that it needs again. Employing such an approach removes the need for distributed deadlock detection, resulting in better throughput for short-lived transactions in OLTP applications [60].

The coordinator queues the request at that all of the nodes that contain the partitions that the transaction will access. When the transaction acquires a partition's lock, the coordinator prepares an acknowledgement message to send back to the transaction's base partition. Once the transaction acquires all the locks that it needs from a node's partitions, the coordinator sends this acknowledgement. Once a transaction receives all of the lock acknowledgements for the partitions that it needs, the coordinator for its base partition schedules the transaction to run immediately on its base partition's execution engine [18, 24].

4.2.5 Query Processing

Every partition in H-Store is managed by a single-threaded execution engine that has exclusive access to the data at that partition. An execution engine is comprised of two parts, one written in Java and one written in C++. In the Java-level component, the execution engine's thread blocks on a queue waiting for messages to perform work on behalf of transactions. This work can either instruct the engine to invoke a procedure's control code to start a new transaction or to execute a query plan fragment on behalf of a transaction running at another partition. Note that for the latter, H-Store's transaction coordination framework ensures that no transaction is allowed to queue a query request at an execution engine unless the transaction holds the lock for that engine's partition.

The execution engine's C++ library is where H-Store stores databases and processes queries. The Java layer uses the Java Native Interface (JNI) framework to invoke the methods in the C++ library and passes it the query plan identifiers that the transaction invoked. This library is not aware of other partitions or nodes in the cluster; it only operates on the input that it is provided.



```
# PRE-DEFINED SQL STATEMENTS
QueryX = "SELECT * FROM X WHERE X_ID=? AND VAL=?"
QueryY = "SELECT * FROM Y WHERE Y_ID=?"
QueryZ = "UPDATE Z SET VAL=? WHERE Z_ID=?"

# TRANSACTION CONTROL CODE
run(x_id, y_id, value):
   result = executeSQL(QueryX, x_id, value)
   if result == null: abort()
   result = executeSQL(QueryY, y_id)
   # ADDITIONAL PROCESSING...
   executeSQL(QueryZ, result, x_id)
   return
```

Figure 4.7: A stored procedure defines (1) a set of parameterized queries and (2) control code. For each new transaction request, the DBMS invokes the procedure's `run` method and passes in (3) the procedure input parameters sent by the client. The transaction invokes queries by passing their unique handle to the DBMS along with the values of its (4) query input parameters.

Stored Procedures

Each stored procedure is identified by a unique name and consists of user-written Java *control code* (i.e., application logic) that invokes pre-defined parameterized SQL commands. The application initiates transactions by sending a request to the DBMS that contains the procedure name and input parameters to the cluster. The input parameters to these stored procedures can be either scalar or array primitive values.

As shown in the example in Figure 4.7, a stored procedure has a "run" method that contains the application logic for that procedure. There are no explicit `begin` or `commit` commands for transactions in H-Store. A transaction begins when the execution engine of its base partition invokes this method and then completes when this method returns (either through the `return` or `abort` commands). When this control code executes, it makes query invocation requests at runtime by passing the target query's handle along with the input parameters for that invocation to the H-Store runtime API (e.g., `queueSQL`). The values of these input parameters will be substituted for the query's parameter placeholders (denoted by the "**?**" in the SQL statements in Figure 4.7). The DBMS queues each invocation and then immediately returns back to the control code. Multiple invocations of the same query are treated as separately even if they use the same input parameters.

After adding all of the invocation requests that it needs to the current batch, the control code then instructs the DBMS to dispatch the batch for execution (e.g., `executeBatch`). At which point, the control code is blocked until the DBMS finishes executing all of the queries in the current batch or aborts the transaction due to an error (e.g., if one of the queries violates a integrity constraint). This command returns an ordered list of the output results for each query invocation in the last batch executed.

H-Store includes special "system" stored procedures that are built into the DBMS. These procedures allow users to execute administrative functions in the system, such as bulk loading data into tables, modifying configuration parameters, and shutting down the cluster.

Although stored procedures in H-Store contain arbitrary user-written Java code, we require that all of their actions and side-effects are deterministic. That is, each stored procedure must be written such that if the DBMS executes a transaction again with the same input parameters and in the same order (relative to other transactions), then the state of the database after that transaction completes will be the same. This means that the procedure's control code is not allowed to execute operations that may give a different result if it is executed again. This requirement is necessary for H-Store recovery (Section 4.2.6).

The types of non-deterministic operations that are forbidden in a stored procedure's control code include (1) using an RPC library inside of a procedure to communicate with an outside system, (2) retrieving the current time from the node's system clock, or (3) using a random number generator. As an example of why these are problematic, consider a procedure that contacts an outside third-party fraud detection system to determine whether a financial transfer is fraudulent during the transaction. The transaction will then choose whether to commit or abort based on the response from this system. One problem with doing this in an H-Store transaction is that this service may report a false positive if the same request is sent to it multiple times by different invocations of the same transaction running on different nodes (i.e., replicated deployments). Additionally, the service may be unavailable at a later date when the DBMS replays the transaction (i.e., crash recovery). In either case, the database will be inconsistent. Thus, in order for this application to work reliably in H-Store, the developer would need to move the fraud detection operation outside of the procedure.

Stored Procedure Routing

In addition to partitioning or replicating tables, a database design can also ensure that each transaction request is routed to the partition that has the data that it will need (i.e., its base partition) [126]. H-Store uses a procedure's *routing attribute(s)* defined in a design at runtime to redirect a new transaction request to a node that will execute it [112]. The best routing attribute for each procedure enables the DBMS to identify which node has the most (if not all) of the data that each transaction needs, as this allows them to potentially execute with reduced concurrency control [120]. Figure 4.8 shows how transactions are routed according to the value of the input parameter that corresponds to the partitioning attribute for the `CUSTOMER` table. If the transaction executes on one node but the data it needs is elsewhere, then it must execute with full concurrency control. This is difficult for many applications, because it requires mapping the procedures' input parameters to their queries' input parameters using either a workload-based approximation or static code analysis.

4.2.6 Durability and Recovery

Since H-Store is a main memory DBMS, it must ensure that all of a transaction's modifications are durable and are recoverable if a node crashes. The key, however, is to provide this guarantee without significantly hindering the performance advantages of the system. Since H-Store is designed to run on commodity hardware, we cannot assume that the DBMS will be deployed using special-purpose components (e.g., battery-backed up memory) [49].

Given this, H-Store uses a lightweight, logical logging scheme that has less overhead than existing approaches for disk-oriented sys-

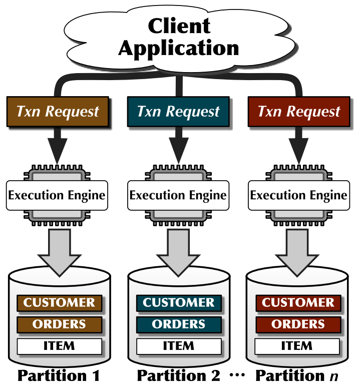


Figure 4.8: Routing of transaction stored procedure to the best base partition.

tems [102]. It will also take periodic checkpoints of the database to reduce the recovery time of the system after a crash. An overview of this process is depicted in Figure 4.9. We now discuss these two mechanisms. We then discuss in Section 4.2.6 how H-Store restores the state of the database from these logs and checkpoints.

Command Logging

As discussed in Section 3.4, there are several techniques for logging, ranging from physical to logical. H-Store uses a variant of logical logging, known as *command logging*, where the DBMS only records the transaction invocation requests to the log [96]. Each log record contains the name of the stored procedure and the input parameters sent from the application, and the transaction's id. Because one log record represents the entire invocation of the transaction, command logging does not support transaction save points [103]. This is not a significant limitation because OLTP transactions are short-lived.

H-Store writes out the log records using a separate thread; the execution engines are never blocked by the logging operations. The DBMS writes the entries after the transaction has executed but before it returns the result back to the application. This is different than write-ahead logging [59], where the DBMS logs the transaction request before it executes. This has two advantages. The first is that the DBMS

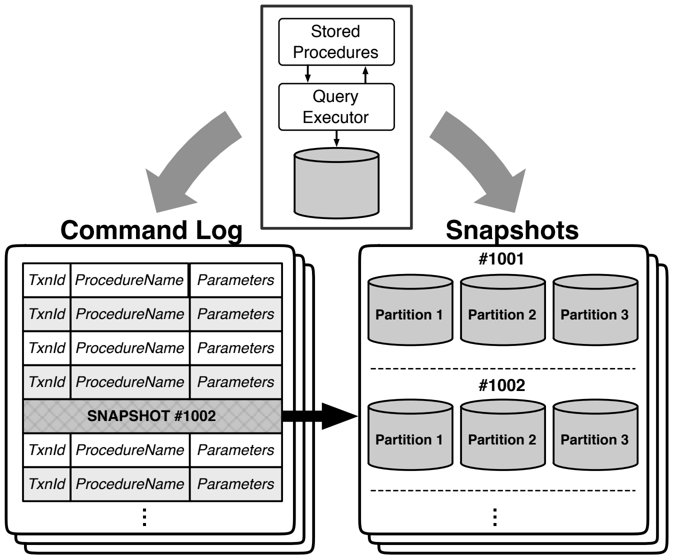


Figure 4.9: An overview of H-Store's logging and checkpoint scheme.

does not need to write out entries for transactions that abort, since the system ensures that all of the changes made by an aborted transaction are rolled back first before executing the next transaction. Thus, an aborted transaction's affect on the database's state is the same as if it was never executed at all. Second, because H-Store can restart transactions due to internal control mechanisms, a transaction may be assigned multiple transaction identifiers. For example, when a transaction attempts to access a partition that it does not have the lock for, the DBMS will restart that transaction, rollback any changes that it made, assign it a new transaction identifier, and then resubmit the lock acquisition requests at the partitions that it needs. If the DBMS logged the transaction's command prior to execution, it would need to write a new entry every time a transaction restarted that marked the previous entry as voided.

The DBMS combines command log entries together for multiple transactions and writes them in a batch to amortize the cost of writing to disk [36, 61, 157]. Modifications made by transactions are not visible to the application until their log record has been flushed. Similarly, a transaction cannot be released to the application until all transactions that executed before it have been written to the command log.

Snapshots

As the DBMS executes transactions and writes their commands out to the log, the DBMS also creates non-blocking snapshots of the database's tables [90, 142]. The snapshot for each partition is written to the local disk at their host node. When the system needs to recover the database after a crash, it loads in the last checkpoint that was created and then replays only the transactions that appear in the command log after this checkpoint [96]. This greatly reduces the time needed to recover the database. H-Store's snapshots only contain the tuples in the tables and not the indexes.

The DBMS can be configured to take checkpoints periodically or manually using a system stored procedure. The system also maintains a catalog of the snapshots that it has taken that is retrievable by the application through another system procedure.

When H-Store starts a new checkpoint, one node in the DBMS is elected as the coordinating node for the next checkpoint. This node is either selected at random (if it for is a scheduled checkpoint initiated by the DBMS) or the node with the transaction's base partition (if it was initiated through a system procedure). The DBMS at this node sends a special transaction request to every partition in the cluster to instruct them to begin the checkpoint process. This request locks all of the partitions to ensure that each node starts writing the checkpoint from a transactionally consistent database state [122]. This system procedure causes each execution engine at all of the partitions to switch into a special "copy-on-write" mode. Any changes made by future transactions do not overwrite the tuples that existed when the current checkpoint started and any new tuples that were inserted after the checkpoint was started are not included in the snapshot data. Once all of the partitions send back acknowledgements, the DBMS commits the special transaction and each partition starts writing out the snapshot to disk in a separate thread. The execution engines then return to processing transactions while the snapshot processing occurs in the background.

The amount of time that it takes the execution engine at a partition to complete the database snapshot out to disk depends on the size of the database and the write speed of the storage device. After a partition's engine finishes writing the snapshot, it disables the "copy-on-write" mode and sends a notification message back to the coordinating node. Once the coordinating node has notifications from every partition, it sends a final finish message to each partition that instructs them to clean up transient data structures and marks the snapshot as complete.

Crash Recovery

The process for restoring a database from the command log and snapshot is straightforward [96]. First, when the H-Store node starts, each execution engine at the node reads in contents from the last snapshot taken at its partition. For each tuple in a snapshot, the DBMS has to determine what partition should store that tuple, since it may not be the same one that is reading in that snapshot. This situation can occur if the administrator changes number of partitions in the cluster when the system was brought off-line. As the engines load in each row, it will also rebuild the tables' indexes at its partitions.

Once the snapshot has been loaded into memory from the file on disk, the DBMS will then replay the command log to restore the database to state that it was in before the crash. A separate thread scans the log backwards to find the record that corresponds to the transaction that initiated the snapshot that was just loaded in. It then scans the log in the forward direction from this point and re-submits each entry as a new transaction request at that node. The transaction coordinator ensures that these transactions are executed in the exact order that they arrived in the system; this differs from the normal execution policy where transactions are allowed to get re-ordered and re-executed.

The state of the database after this recovery process is guaranteed to be correct, even if the number of partitions during replay is different from the number of partitions at runtime. This is because (1) transactions are logged and replayed in serial order, so the re-execution occurs in exactly the same order as in the initial execution, and (2) replay begins from a transactionally-consistent snapshot that does not contain any uncommitted data, so no rollback is necessary at recovery time [59, 122, 96].

4.3 HyPer

4.3.1 Introduction

The in-memory database system HyPer was developed at the Technical University of Munich to accommodate hybrid OLTP&OLAP workloads on the same database state with full ACID guarantees. This allows OLAP data exploration on the most recent transactional database state; thereby enabling the often cited "real time analytics". Traditionally, the two workloads, OLTP and OLAP, were separated in two dedicated systems: The transactional database with a normalized schema and the data warehouse for read-mostly analytical query processing. The necessary ETL process incurs the problems of complexity and staleness of the data warehouse because of the time delay until data is refreshed.

The advent of ever more powerful database servers, even in the form of commodity servers with many dozens of cores and several Terabytes of main memory capacity has finally paved the way to consolidate these two heterogeneous workloads in a single system instance.

4.3.2 Data Organization

Isolating OLTP-Transactions and OLAP-Queries by Snapshotting

In order to accommodate both OLTP and OLAP workloads simultaneously on the same database state, it is necessary to effectively isolate these two tasks from each other. One possible approach would be some kind of update staging that separates data between the main database system that contains older, mostly immutable data objects and the delta store that contains most recently inserted and updated data objects. Periodically the delta store is merged into the main store. The disadvantage of the update staging approach of incurring extra reorganization effort for merging the delta with the main database system led

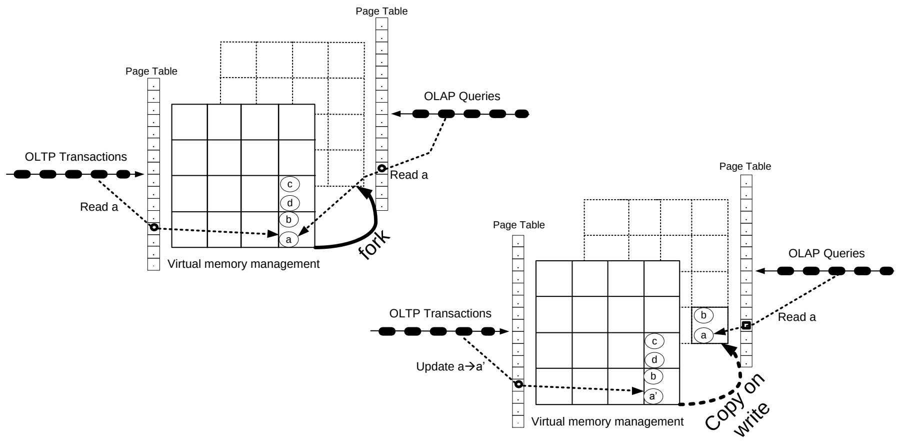


Figure 4.10: Snapshotting via virtual memory copy-on-write mechanism

us in the design of an alternative architecture in which the transactional database is entirely maintained in a coherent state.

HyPer exploits 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. This constraint can be relaxed 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 on the left of Figure 4.10 by the overlayed page frame panel. This virtual memory snapshot that is created by the `fork()`-operation will be used for executing a session of OLAP queries – as indicated on the right hand side of Figure 4.10.

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-write/update* strategy – as sketched out on the right in Figure 4.10. Initially, parent process (OLTP) and child process (OLAP) share the same physical memory segments by translating either virtual addresses (e.g., to object a) to the same physical main memory location. The sharing of the memory segments is highlighted in the graphics by the dotted frames. A dotted frame represents a virtual memory page that was not (yet) replicated. Only when an object, like data item a, is updated, the OS- and hardware-supported copy-on-update mechanism initiate the replication of the virtual memory page on which a resides. Thereafter, there is a new state denoted a' accessible by the OLTP-process that executes the transactions and the old state denoted a, that is accessible by the OLAP query session. Unlike the figure suggests, the additional page is really created for the OLTP process that initiated the page change and the OLAP snapshot refers to the old page – this detail is important for estimating the space consumption if several such snapshots are created. This snapshotting mechanism completely separates OLTP transaction processing from OLAP query evaluation by means of the operating system in combination with the memory management unit (MMU).

Hybrid Storage Structures

As highlighted in Chapter 3, HyPer avoids database-specific buffer management and page structuring. The data resides in simple main-memory optimized data structures within the virtual memory. Thus, HyPer exploits the OS/CPU-implemented address translation at "full speed" without any additional indirection. 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.

For organizing relations in the virtual memory there are two well known extremes at the borders of the design space: In the *row store* approach relations are maintained as arrays of entire records and in the *column store* approach the relations are vertically partitioned into vectors of attribute values. HyPer can be configured to operate as a column or a row store – but the table layout can also be adjusted according to the access patterns. In the hybrid storage format, it is possible to cluster those attributes that are frequently accessed together into a single vector constituting a non-redundant vertical fragmentation.

Assume that the following query occurs frequently:

```
select oDate, sum(oPrice)
from Orders
where oDate >= 20130101
group by oDate
```

Then clustering the two attributes, *oDate* and *oPrice* into the same vector would be beneficial. Below we show such a storage layout (for simplicity the example uses C++/STL for vector collections, though HyPer makes use of its own data structures):

```
/// An Order
struct Order { unsigned id; unsigned customer; unsigned product;
               unsigned oDate; double oPrice; };
struct OrderDatePrice { unsigned oDate; double oPrice; };
/// All Orders in hybrid format
struct Orders {
 vector<unsigned> data_id;
 vector<unsigned> data_customer;
 vector<unsigned> data_product;
 vector<OrderDatePrice> data_oDate_oPrice;

 void insert(Order&& order);
};
```

Code Generation for Queries. The above shown example query could then be translated into the following C++ code, which relies on a scan of the *one* clustered vector *data_oDate_oPrice*:

```
unordered_map<unsigned, double> revenueByDate(Orders& orders)
{
   unordered_map<unsigned, double> groupBy;
   for (OrderDatePrice date_price : orders.data_oDate_oPrice) {
      if (date_price.oDate >= 20130101) {
         groupBy[date_price.oDate] += date_price.oPrice;
      }
   }
   return groupBy;
}
```

This program fragment simplifies many aspects of the HyPer query engine; nevertheless it is meant to demonstrate that translating declarative SQL queries can indeed result in executable code that is as fast as (or due to multi-core parallelization even much faster than) handwritten code.

The actual HyPer engine deviates from this simple code generation in some major ways:

1. HyPer employs a full-fledged advanced query optimizer for optimizing the join order, unnesting nested subqueries, predicate pushdown, etc.

2. HyPer compiles SQL queries and transactional programs written in HyPerScript into LLVM assembler code. The foremost advantage is the elimination of the C++ compiler which takes seconds and thus impedes interactive query processing.

3. HyPer automatically parallelizes the query execution to make best use of a multi-core server that, nowadays, has dozens or hundreds of cores.

4. HyPer employs sophisticated data structures and algorithms that are particularly cache-conscious and incur low synchronization overhead for thread-level parallelization.

Dynamic Storage Allocation. HyPer initially assumes that a relation remains small and allocates only a fixed (small) vector – in order to avoid the bookkeeping overhead for dynamically allocated storage. However, once a relation grows beyond a certain threshold dynamically growing partition vectors that are addressed via a direct mapping table *DM*, as shown in Figure 4.11, are used in HyPer. All data vectors of one horizontal fragment are stored contiguously. If a pure columnar format is used, all attribute vectors are concatenated. In our example, we chose one clustered data vector which happens to be stored at the end of the fragment. Each time a fragment overflows, a new fragment is allocated such that the overall number of (possible) rows doubles. Therefore, the first two fragments in the figure have space for two rows, the next one

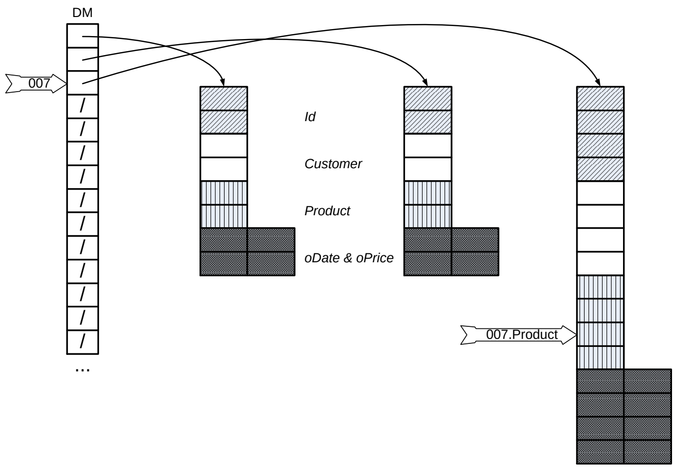


Figure 4.11: The dynamic storage allocation

for four rows, then for eight rows, etc. Accessing an attribute at a particular position, say retrieving the attribute value of *Product* of row 007 is done in two steps:

1. Look up the fragment's start address via the direct mapping table *DM*. The address within *DM* is found by a rather efficient transformation by counting the leading zeros of the 64-bit row number and subtracting it from the number of positions in *DM*. In the example this yields the marked position in the direct mapping table.

2. Having retrieved the starting address of the corresponding fragment (the third fragment in the example) an address calculation yields the marked position of the attribute value. This address calculation is optimized by pre-computing a materialized table based on the number of rows in the fragment and the attributes' widths.

The indirection via the direct mapping table incurs little overhead because it is small enough to fit into the L1 cache and all address transformations are supported by pre-materialization.

4.3.3 Indexing

HyPer uses a new main-memory index that relies on radix-segmentation of the search key. Therefore, mostly branch- and comparison-free code can be used to navigate the radix tree index from root to leaf. Each level of the tree maintains a particular radix fragment – in our case one byte – of the search key. Before HyPer, radix trees (often called tries) suffered from poor storage utilization as the maximum degree of fan-out was maintained in every node. In comparison to balanced search trees, like AVL- or red/black-trees tries have the "nice" property that their height is not dependent on the number of indexed objects. Rather, the length of the search keys determines the height of a radix tree.

The HyPer radix tree design, called Adaptive Radix Tree ART [85], uses adaptive node sizes depending on the actual fan-out of a node in order to guarantee a good space utilization. Thus, a node starts out small with space for a four-way fan-out, then grows to a size for a 16-way fan-out. If further search keys are inserted, it grows to a 48-way fan-out and ultimately to a fan-out of 256.

As mentioned above ART employs four adaptively sized node types:

- **Node4**: This node type has a fan-out of up to four by storing a maximum of four sorted search keys.
- **Node16**: Here, up to 16 search keys are store in sorted sequence.
- **Node48**: In this node type a 256-element array is used with one entry for each search key. The entries constitute short (i.e., one byte) pointers to one of the 48 positions where the "real" child pointer is to be found.
- **Node256**: Here the 256-element array is used where an entry holds a pointer to the child node.

Figure 4.12 provides an example of adaptive node types that shows a sample path from root to leaf of an ART tree that indexes four-byte integers. The height of this tree is four. The sample path was, for illustration purposes, constructed in such a way that all four node types appear in it. The path in the search tree was constructed for

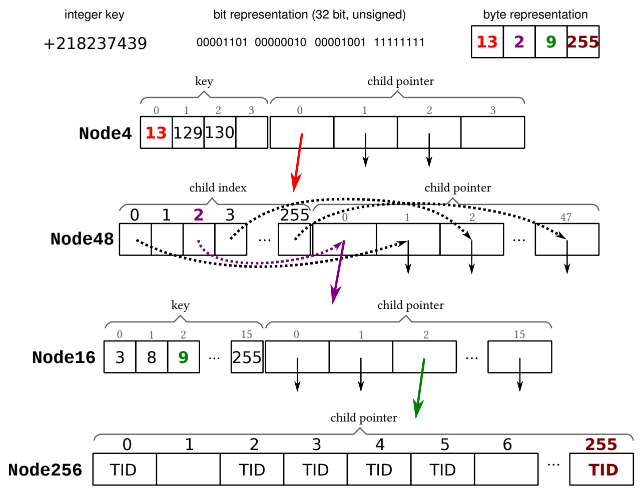


Figure 4.12: A sample path with adaptive nodes in the radix tree

the integer search key 218237439, which consists of the four single-byte portions 13&2&9&255. The root node happens to be of type *Node4*. On the way to the leaf the other three node types are encountered.

In designing the node types the focus was on storage efficiency as well as search efficiency. The node type *Node16*, for example, was constructed such that on modern processors with vector instructions all stored search keys, i.e., all 16 bytes, can be compared in parallel.

Besides node adaptivity two further optimizations have been incorporated in ART: Long search keys which lead to a single leaf are collapsed since no fan-out is needed. As a consequence these leaves are pulled into the interior of the search tree. Upon insertion of another key with identical prefix, these nodes sink in the resulting tree to lower positions. Furthermore, prefixes that are common to all search keys of a node are “factored out” and stored only once. This is particularly useful for indexing keys like URLs all prefixed with “http://”. These techniques in combination with the adaptive node sizing guarantee that the worst-case space usage per key is bound by 52 bytes.

4.3.4 Concurrency Control

As HyPer is a hybrid OLTP/OLAP engine it is essential to retain fast scan performance. Therefore, in designing a multi-version concurrency control scheme for transaction isolation it was important to install the latest update in place in order to preserve contiguous data placement such that the processor's prefetcher can proactively move data into the caches while scanning the relation. The MVCC was primarily designed to isolate parallel OLTP transactions. It is still possible to fork OLAP snapshots for compute-intensive analytical queries.

Figure 4.13 illustrates the version maintenance using a traditional banking example. For simplicity, the database consists of a single *Accounts* table that contains just two attributes, *Owner* and *Bal*ance. HyPer refrains from creating new versions; it instead *updates in-place* and maintains the backward delta between the updated (yet uncommitted) and the replaced version in the undo buffer of the updating transaction. Updating data in-place retains the contiguity of the data vectors that is essential for high scan performance.

Upon committing a transaction, the newly generated version deltas have to be re-timestamped to determine their validity interval. Clustering all version deltas of a transaction in its undo buffer expedites this commit processing tremendously. Furthermore, using the undo buffers for version maintenance, the MVCC model incurs almost no storage overhead as it needs to maintain the version deltas (i.e., the before-images of the changes) during transaction processing anyway for transactional rollbacks. The only difference is that the undo buffers are (possibly) maintained for a slightly longer duration, i.e., for as long as an active transactions may still need to access the versions contained in the undo buffers. Thus, the *VersionVector* shown in Figure 4.13 anchors a chain of version reconstruction deltas (i.e., column values) in "newest-to-oldest" direction, possibly spanning across undo buffers of different transactions. Even for the column store back-end, there is a single *VersionVector* per record, so the version chain in general connects before-images of different columns of one record.

Only a tiny fraction of the database will be versioned, as old versions are continuously garbage collected when no longer needed. A version

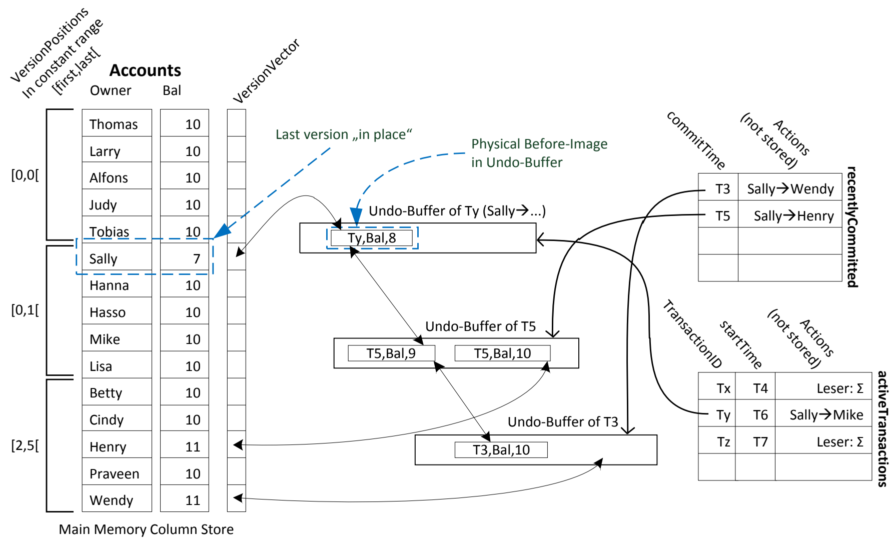


Figure 4.13: Multi-version synchronization of two transaction types: Transfer between two Accounts (from → to) and summing all Account Bal-ances (Σ)

(reconstruction delta) becomes obsolete if all active transactions have started after the delta was timestamped. The *VersionVector* contains *null* whenever the corresponding record is unversioned and a pointer to the most recently replaced version in an undo buffer otherwise.

In the example there are only two transaction types: *transfer* transactions are marked as "*from* → *to*" and transfer \$1 *from* one account *to* another by first subtracting 1 from one account's *Bal* and then adding 1 to the other account's *Bal*. Initially, all *Bal*ances are set to 10. The read-only transactions denoted Σ sum all *Bal*ances and—in our "closed world" example—should always compute \$150, no matter under what *startTime*-stamp they operate.

All new transactions entering the system are associated with two timestamps: *transactionID* and *startTime*-stamps. Upon commit, update transactions receive a third timestamp, the *commitTime*-stamp that determines their serialization order. Initially all transactions are assigned identifiers that are higher than any *startTime*-stamp of any transaction.

Update transactions modify data *in-place*, retaining the old version of the data in the undo buffer. This old version serves two purposes: (1) it is needed as a before-image in case the transaction is rolled back (undone) and (2) it serves as a committed version that was valid up to now. While the updater is still running, the newly created version is marked with its *transactionID*, whereby the uncommitted version is only accessible by the update transaction itself. At commit time an update transaction receives its *commitTime*-stamp with which its version deltas (undo logs) are marked as being irrelevant for transactions that start from "now" on. This *commitTime*-stamp is taken from the same sequence counter that generates the *startTime*-stamps.

In the example, the first update transaction that committed at timestamp T_3 (Sally $\rightarrow$ Wendy) created in its undo buffer the version deltas timestamped T_3 for Sally's and Wendy's balances, respectively. The timestamp indicates that these version deltas have to be applied for transactions whose *startTime* is below T_3 and that the successor version is valid from there on for transactions starting after T_3. At *startTime* T_4 a reader transaction with *transactionID* T_x entered the system and is still active. It will read Sally's *Bal*ance at reconstructed value 9, Henry's at reconstructed value 10, and Wendy's at value 11. Another update transaction (Sally $\rightarrow$ Henry) committed at timestamp T_5. Again, the versions belonging to Sally's and Wendy's balances that were valid just before T_5's update are maintained as before images in the undo buffer of T_5. Note that a reconstructed version is valid *from* its predecessor's timestamp *until* its own timestamp. Sally's *Bal*ance version reconstructed with T_5's undo buffer is thus valid from timestamp T_3 until timestamp T_5. If a version delta has no predecessor (indicated by a null pointer) such as Henry's balance version in T_5's undo buffer its validity is determined as from virtual timestamp "0" until timestamp T_5. Any read access of a transaction with *startTime* below T_5 applies this version delta and any read access with a *startTime* above or equal to T_5 ignores it and thus reads the in-place version in the *Accounts* table.

The deltas of not yet committed versions receive a temporary timestamp that exceeds any "real" timestamp of a committed transaction.

This is exemplified for the update transaction (Sally $\rightarrow$ Henry) that is assigned the *transactionID* timestamp T_y of the updater. This temporary, very large timestamp is initially assigned to Sally's *Bal*ance version delta in T_y's undo buffer. Any read access, except for those of transaction T_y, with a *startTime*-stamp above T_5 (and obviously below T_y) apply this version delta to obtain value 8. The uncommitted in-place version of Sally's balance with value 7 is only visible to T_y.

Serializability Validation

HyPer's MVCC approach deliberately avoids write-write conflicts, as they may lead to cascading rollbacks. If another transaction tries to update an uncommitted data object, it is aborted and restarted. Therefore, the first *VersionVector* pointer always leads to an undo buffer that contains a committed version — except for unversioned records where the pointer is null. If the same transaction modifies the same data object multiple times, there is an internal chain of pointers within the same undo buffer that eventually leads to the committed version.

In order to retain a scalable lock-free system, HyPer relies on optimistic execution in its MVCC model. Without any further validation the described CC scheme guarantees (only/already) snapshot isolation. To guarantee full serializability, a validation phase is needed at the end of a transaction that ensures all reads during transaction processing can be read (logically) at the very end of the transaction without any observable change. Validation detects four relevant transitions: modification, deletion, creation, and creation & deletion of an object that is "really" relevant for the transaction T. For this purpose, transactions draw a *commitTime*-stamp from the counter that is also produces the *startTime*-stamps. The newly drawn number determines the serialization order of the transaction. Only modifications that were committed during T's lifetime, i.e., in between the *startTime* and the *commitTime*, are relevant if these modified/deleted/created objects *really* intersect with T's read *predicate space*.

As opposed to previous validation schemes that may need to recheck large read sets (and scan sets), HyPer limits the validation to the number of recently changed and committed data objects, no matter how

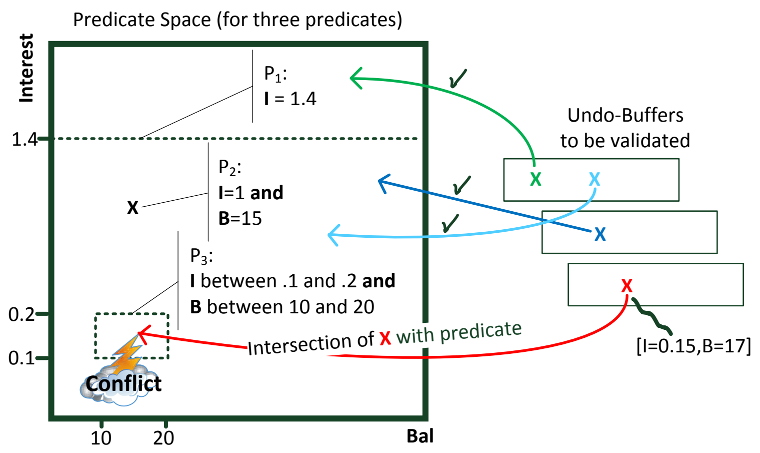


Figure 4.14: Validating data (Points) in the redo buffers against the predicate space of a transaction

large the read set of the transaction was. For this purpose, HyPer uses an old (and largely "forgotten") technique called *Precision Locking* [69] that eliminates the inherent satisfiability test problem of predicate locking. This variation of precision locking tests discrete writes (updates, deletions, and insertions of records) of recently committed transactions against predicate-oriented reads of the transaction that is being validated. Thus, a validation fails if such an extensional write intersects with the intensional reads of the transaction under validation [155]. The validation is illustrated in Figure 4.14, where transaction T has read objects under the three different predicates P_1, P_2, and P_3, which form T's predicate space. We need to validate the three undo buffers at the right and validate that their objects (i.e., data points) do not intersect with T's predicates. This is done by evaluating the predicates for those objects. If the predicates do not match, then there is no intersection and the validation passes, otherwise, there is a conflict.

In order to find the extensional writes of other transactions that committed during the lifetime of a transaction T, HyPer maintains a list of *recentlyCommitted* transactions, which contains pointers to the corresponding undo buffers. Validation starts with the undo buffers of the oldest transaction that committed after T's *startTime* and traverses to the youngest one (Figure 4.13 at the bottom of the list). Each of the undo buffers is examined as follows: For each newly created version, validation checks whether it satisfies any of T's selection predicates. If this is the case, T's read set is inconsistent because of the detected phantom and it has to be aborted. For a deletion, validation checks whether or not the deleted object belonged to T's read set. If so, T is aborted. For a modification (update) both the before and after image is validated. If either intersects with T's predicate space, it is aborted. Figure 4.14 depicts this situation, where the data point x of the lowest undo buffer satisfies predicate P_3, meaning that it intersects with T's predicate space.

After successful validation, a transaction T is committed by first writing its commit into the redo-log (which is required for durability). Thereafter, all of T's *transactionID* timestamps are changed to its newly assigned *commitTime*-stamp. Due to version maintenance in the undo buffers, all these changes are local and therefore very cheap. In case of an abort due to a failed validation, the usual undo-rollback takes place, which also removes the version delta from the version chain. Note that the serializability validation in the MVCC model can be performed in parallel by several transactions whose serialization order has been determined by the *commitTime*-stamps.

4.3.5 Query Processing

An expressive scripting language is the key to pushing application logic (as stored procedures) directly into the database system, instead of relying on an application server. For this purpose, HyPer uses the HyPer-Script language that integrates declarative SQL with imperative constructs, such as loops and branches. As an example, we use the skeleton of the *newOrder* procedure of the TPC-C benchmark. This procedure inserts a new customer order consisting of a variable number of order positions that are passed as a table-valued parameter *positions*.

```
create procedure newOrder (w_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) // TABLE-valued parameter above
{
   select w_tax from warehouse w where w.w_id=w_id;
   ...  // w_tax value used later
   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),
             ...
      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
};
```

HyPerScript allows “normal” SQL queries whose result is used later in the program. This is exemplified by the first query that selects the tax rate of a particular warehouse and later uses this variable *w_tax* when the order position is inserted into the relation *orderline*. This sample script first extracts the relevant information from the underlying tables using SQL queries. Then the new *order* record is created, a reference to this record is also inserted into the table *neworder*. Then the *stock* table is updated for bookkeeping of this new order. In the last steps, the individual order positions, as passed in the table parameter *positions*, are inserted into the *orderline* table. For this purpose the aggregated price is calculated, including taxes and reduced according to the discount. At the end, the script tests whether or not all order positions were successfully inserted. If not, the entire transaction (i.e., the *newOrder* script) has to be rolled back.

Using a declarative scripting language has many advantages:

1. It is easier to analyze declarative scripts to detect security problems. This is crucial if the stored procedures run in the same process as the actual database server.

2. The embedded SQL queries can be optimized by the regular query optimizer. HyPer also uses the same compilation technique that was developed for queries – as described in the next section.

3. The resulting scripts are very concise and, therefore, quite readable as our sample application demonstrates.

Compilation of Queries and Transactions

As introduced in Chapter 3, HyPer deviates from the interpretative processing model and compiles entire (logically optimized) query plans into machine-level code. Other than traditional iterator-based execution models the HyPer query evaluation code is generated for an entire pipeline. For this purpose, the logically optimized algebra tree is segmented into its largest-possible pipelines, i.e., all algebra operations in between two pipeline breakers. This is exemplified for the query plan that evaluates the following join query

```
select *
from R, S, T
where T.x=7 and S.y=3 and R.z>5 and
      T.B=S.B and S.A=R.A
```

In terms of the relational algebra it constitutes the following three-way join with prior (pushed down) selections:

$$\sigma_{z>5}R \bowtie_A \sigma_{y=3}S \bowtie_B \sigma_{x=7}T$$

The logical optimization may have resulted in the algebra plan shown in Figure 4.15 . Figure 4.16 shows the generated code – for simplicity this is pseudo-code instead of LLVM code.

The HyPer compiler is constructed in a modular fashion by invoking the produce/consume interface function as associated with each supported operator. In contrast to the interpreted iterator model, these functions are invoked at compile time. In order to generate data centric code, pipeline operators directly invoke the consume-function of their parent operator. Thereby, the once materialized data object remains in processor registers for seamless evaluation of all operations within one

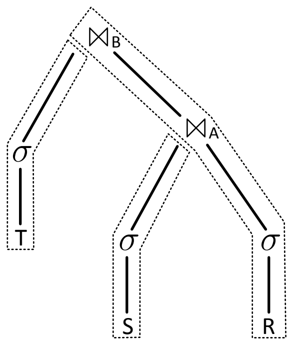


Figure 4.15: Algebra tree

```
initialize memory of hash tables $\bowtie_A$, $\bowtie_B$
for each tuple $t$ in $T$
  if $t.x = 7$
    materialize $t$ in hash table of $\bowtie_B$
for each tuple $s$ in $S$
  if $s.y = 3$
    materialize $s$ in hash table of $\bowtie_A$
for each tuple $r$ in $R$
  if $r.z > 5$
    for each match $s$ in $\bowtie_B$ $[r.B]$
      for each match $t$ in $\bowtie_A$ $[s.A]$
        output $r \circ s \circ t$
```

Figure 4.16: Generated pseudo-code

pipeline. This is particularly prominent in our example pseudo-code for the probing pipeline that starts with a tuple r, determines whether or not the selection predicate $(z > 5)$ holds, then probes the first hash table $\bowtie_A$ to retrieve one matching tuple s that has the same A-value as r (one after another) and then probes the hash table $\bowtie_B$ to, again, retrieve one-by-one matching t-tuples and materializes the combined tuple $(r \circ s \circ t)$.

The algebraic operator model is very useful for reasoning about queries during query optimization, but does not mirror how queries are executed at runtime. For example, the three lines in the first code fragment of Figure 4.16 belongs to the table scan T, the selection $t.X = 7$, and the building of the hash join table for $\bowtie_B$ respectively. Nevertheless, the query compiler logically works on an operator tree, which was produced by the query optimizer, and statically transforms the operator tree into executable code. Conceptually, each operator offers a unified interface that is quite different from the iterator model but almost as simple: It can *produce* tuples on demand, and it can *consume* incoming tuples from a child operator. This conceptual interface allows to generate data-centric code, while retaining the composability of the algebraic operator model. Note, however, that this interface

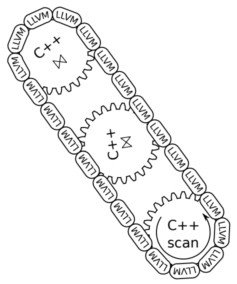


Figure 4.17: Combining C++ and LLVM code

is only a concept used during code generation – it does not exist at runtime. That is, these functions are only used to generate the appropriate LLVM code for producing and consuming tuples, but they are not called at runtime.

In theory one could generate the entire code for every individual query. However, to reduce the size and complexity of the code that is generated for the subsequent just-in-time compilation it is beneficial to implement the core of the operators (e.g., hash join, index nested loops join, hash aggregation, sorting) beforehand and merely generate the query dependent code dynamically. To ease the implementation this functionality is typically written in C++ instead of LLVM. In that respect, the generated LLVM code constitutes the chain that drives the cog wheels of the pre-fabricated code base – as exemplified in Figure 4.17.

Massively Parallel Query Processing: Joins

The HyPer Query Engine fully utilizes the multi-core compute power of modern processors by intra-operator parallelism. This section summarizes HyPer's innovation in this area by describing strategies for one of the most important operators: the join. In the near future, database servers will have hundreds of compute cores. This requires that the parallel processing tasks have to be as autonomous as possible without synchronization points which would likely lead to idle waiting times of many cores. The more cores the more severe are the effects of Amdahls law. Because of the non-uniform memory access (NUMA) characteristics of multi-core servers with large DRAM capacity it is not sufficient to merely parallelize the computation. It is also essential to allocate computational tasks locally.

Massively Parallel Sort/Merge-Join. HyPer's massively parallel sort-merge (MPSM) join is designed to take NUMA architectures into account which were not yet in the focus of prior work on parallel join processing for main memory systems. MPSM relies on chunking the arguments into as many equally sized fragments as there are available cores for parallel processing. Unlike traditional sort-merge joins, MPSM refrains from merging the sorted runs to obtain a global sort order and rather joins them all in a brute-force but highly parallel manner, opting to invest more into scanning in order to avoid the hard-to-parallelize merge phase. Obviously, this decision does not result in a globally sorted join output but exhibits a partial sort order that still allows for sort order based subsequent operations, e.g, early aggregation. During the subsequent join phase, data accesses across NUMA partitions are sequential, so that the prefetcher mostly hides the access overhead. Details of MPSM can be found in [9].

Parallel Radix-Hash-Join. The sort/merge-join uses sorting to efficiently join objects via (almost) linear synchronous scans. Unfortunately, sorting is still costly. The hash join idea is quite simple in the case of equi-joins. One of the join argument relations is inserted into a hash table (called the build input) and the other relation (called the probe input) is sequentially scanned and probed into this hash table to find matching join partners.

One possibility to parallelize this hash join algorithm consists of partitioning the argument relations prior to the actual hash join evaluation. Here, a radix partitioning is obviously a very efficient way as it allows to partition relations without costly value comparisons and branching. After partitioning, a thread can independently build hash tables for their partitions.

It is obvious that the hash tables exhibit a low degree of cache locality during the build as well as during the probing phase. Basically, each access can be expected to induce a cache fault. Therefore, it is beneficial to partition the argument relations into smaller fragments such that the resulting hash table fits into the processor's cache in order to avoid these costly cache misses. However, partitioning the relation "in one go" into too many partitions is also detrimental to performance because then the copying into the fragments incurs too many cache misses – in particular, the *translation lookaside buffer* (TLB) becomes too small to hold all page table entries for all write positions. Therefore, a multi-step partitioning can be used.

Parallel Hash-Join without Partitioning. The radix join incurs relatively high copy costs which is, hopefully, amortized by the higher cache locality during building and probing the hash table. In any case, the radix join incurs additional storage cost for maintaining the partitions. Therefore, in a main-memory setting it is a straightforward idea to leave the (typically much larger) probe input in place and merely copy the (smaller) build argument relation into the hash table. In this procedure the build phase requires special attention because many workers are inserting data in parallel into this hash table. This requires short-term latches on the hash table bucket into which a worker inserts a new data item. These so-called latches could be implemented with efficient *compare-and-swap* machine code statements. The latches which are set with this instruction should be associated directly with the hash bucket to guarantee that they are stored in the same cache line.

After building the hash table, the probe phase can be carried out in parallel without any synchronization overhead because the workers only read from the hash table. Every worker works on a (at best, NUMA-local) chunk of the probe argument and determines join partners for these tuples within the hash table. Obviously, this join method is particularly effective if one of the argument relations, i.e., the build input, is (much) smaller than the other probe argument. This simple join method is, in addition, particularly useful for pipeline parallelism [76] – as exemplified in Figure 4.18 where the probe pipeline covers the two hash table probes. Note that pipelining across multiple join hash tables is not possible with the radix join as every binary join requires its individual partitioning.

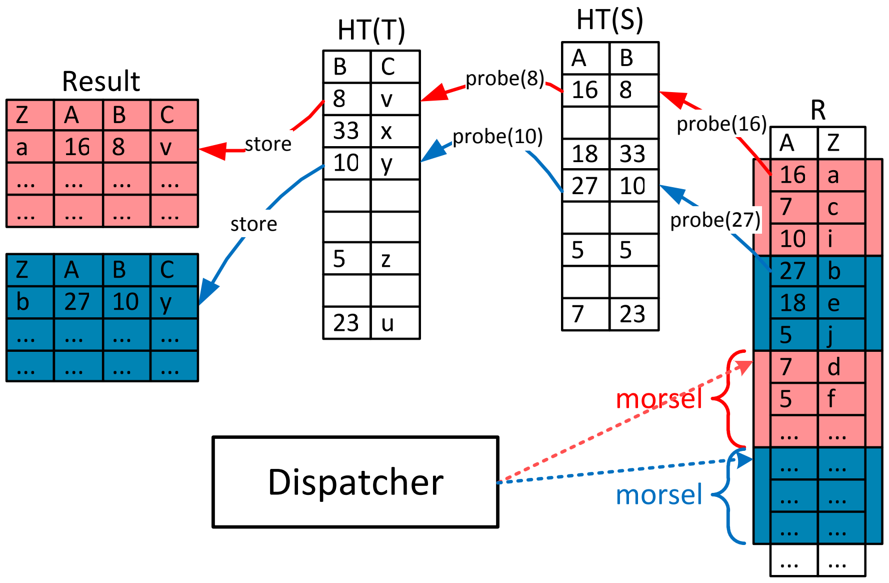


Figure 4.18: Idea of fine-granular parallelization: $R \bowtie_A S \bowtie_B T$. The data chunks (morsels) are assigned to NUMA-local worker threads

Adaptive Morsel-Wise Parallelization and Workload Management

The main impetus of hardware performance improvement nowadays comes from increasing multi-core parallelism rather than from speeding up single-threaded performance. We use the term *many-core* for architectures with tens or (soon) hundreds of cores.

In main-memory database 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 lead 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. In particular, the NUMA division of the RAM has to be considered carefully to ensure that threads work (mostly) on NUMA-local data.

To address these challenges the adaptive *morsel-driven* query execution framework [83] was developed that controls HyPer's parallel operators. The approach is sketched in Figure 4.18 for a 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/red and lower/blue) 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. The dispatcher runs a fixed, machine-dependent number of threads, such that if new queries arrive there is no resource over-subscription. 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 intermediate 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.

The morsel-driven idea extends from just scheduling into a complete query execution framework in that all physical query operators must be able to execute morsel-wise in parallel in all their execution stages (e.g., both hash-build and probe), a crucial need for achieving many-core scalability in the light of Amdahl's law. An important part of the morsel-driven framework is awareness of data locality. This starts from the locality of the input morsels and materialized output buffers, but

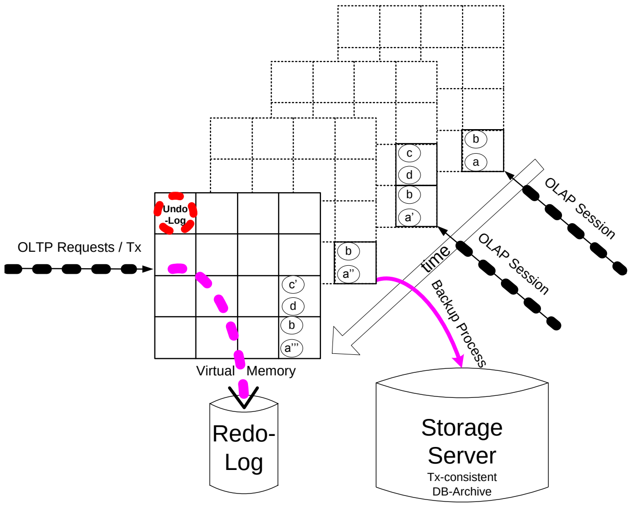


Figure 4.19: Redo-logging and database backup

extends to the state (data structures, such as hash tables) possibly created and accessed by the operators. This state is shared data that can potentially be accessed by any core, but does have a high degree of NUMA locality. Thus morsel-wise scheduling is flexible, but strongly favors scheduling choices that maximize NUMA-local execution. This means that remote NUMA access only happens when processing a few morsels per query, in order to achieve load balance. By accessing local RAM mainly, memory latency is optimized and cross-socket memory traffic (that can slow other threads down) is minimized.

4.3.6 Durability and Recovery

For atomicity HyPer writes an undo log that need not be written to stable storage. It is merely maintained as a ring buffer in DRAM as its entries can be safely overwritten as soon as a transaction is finished. The *durability* of transactions requires that all effects of committed transactions have to be restored after a failure. To achieve this HyPer employs classical redo logging. This is highlighted by the gray ovals emanating from the transaction stream leading to the non-volatile Redo-Log storage device in Figure 4.19. Initially, HyPer used *logical* redo logging 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 it restarts from a transaction consistent archive (cf. Figure 4.19). Nevertheless, logical logging turned out to be problematic as it requires fully deterministic transactions, which is hard to achieve in practice. In particular, aborted transactions may alter the sequence in which tuples are retrieved in SQL statements. Therefore, HyPer switched to physical logging which incurs higher log volumes because all updates are logged by pushing the corresponding after-image values into the log stream.

HyPer can also exploit the VM snapshots for creating backup archives of the entire database on non-volatile storage. This process is sketched in Figure 4.19. Typically, the archive is written via a high-bandwidth network of 10 to 100 Gb/s or even via RDMA in an Infiniband cluster 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.

In the *ScyPer* scale-out extension of HyPer the redo log was used to "feed" secondary server(s) as stand-by OLTP processors in case of primary server failure. THE VM snapshotting mechanism allows to fork consistent snapshots for OLAP processing on the secondary server – thereby load-balancing analytical query processing between primary and secondary servers.

4.3.7 Further Reading

The HyPer architecture including its virtual memory snapshotting was first introduced in 2011 [72]. The virtues and performance of the snapshotting based on virtual memory management versus software-controlled mechanisms was analyzed in [104]. The pioneering JIT compilation of queries and transaction scripts was developed in [109]. Pirk et al [123] analyzed the effect of hybrid storage representations that span the entire design space between column- and row-stores. While column-stores excel for analytics and row-stores are best for transactions, processing the optimal representation is work-load dependent.

The scale-out of HyPer to multiple nodes in a cluster is described in a series of papers [106, 132]). For very fast Infiniband networking infrastructures specialized RDMA-based communication protocols of the query engine were analyzed by Roediger et al [131]. Muhe et al analyzed the use of HyPer for multi-tenancy applications [105]. The small footprint of HyPer allows to allocate a dedicated instance of HyPer for every tenant; thereby achieving complete separation of the data of different tenant which is beneficial from a security point of view.

The massively parallel sort-merge join MPSM was developed by [9]. [76] describes the pipelined hash join that relies on a global hash table, as proposed by [22]. It is particularly beneficial compared to radix-partitioning joins like the one analyzed in [15] if multiple joins can be performed in one pipeline or when one of the join arguments is smaller than the other. Based on these algorithms a comprehensive parallelization of the HyPer query engine was devised in [83]. The Adaptive Radix Tree ART was designed in [85].

HyPer has advanced SQL window functions for powerful decision support functions; Leis et al showed how these window functions are effectively parallelized on a multi-core server [87] . HyPer has an advanced query unnesting method that was described in [110]. The overall query optimizer of HyPer was evaluated in comparison to other market-leading systems in [84]. The multi-version concurrency control was described in [111]. The synchronization of data structures is supported by the recent hardware transactional memory features of Intel Haswell processors, as described in [86].

4.4 SAP HANA

4.4.1 Introduction

Main memory processing within SAP started years before HANA was born. The earliest direct predecessor of SAP HANA was the TREX text search engine (built in 2001) that stored meta data attributes of documents were in columnar format within main memory. Even at that time, keeping these data sets in main memory was not critical because of the small memory footprint of document attributes. The next step of the main memory technology in 2005 was the development of BWA, the SAP Business Warehouse accelerator. Using the BWA, selected data cubes of an SAP Business Warehouse (SAP BW) deployment could be replicated out of traditional disk-based database systems into the BWA for fast query processing especially to accelerate typical OLAP-style queries (e.g. aggregation operations along dimension hierarchies). Already in 2005 the price-benefit relation of memory cost versus performance was acceptable for many customers and BWA became very popular for SAP BW customers. As a logical next step, the idea was born to join forces within different product groups within SAP and start SAP HANA following a step-wise evolution approach from supporting data-mart scenarios extending SAP BW installations to adding more traditional database functionality: The column-based technology of TREX/BWA was used as the foundation for the column-store engine. P*TIME [28] – an in-memory database solution acquired by SAP in 2005 – contributed the row-store engine, the SQL subsystem as well as connection management. The MaxDB engine mainly contributed the persistency layer as well as the surrounding infrastructure/tools.

Hasso Plattner developed the vision to not only have one single system for an integrated data management platform, but also to completely rely on in-memory technology to exploit the capabilities of modern hardware. Driven by modern business application requirements to tightly couple operational as well as tactical and strategic business decisions, the overall goal was to satisfy OLTP as well as OLAP workloads tearing down the wall between operational ERP and traditional DWH-systems solely used to support decision-making processes. Looking back, the decision to focus on in-memory technology was the right decision: a standard ERP deployment exhibits an uncompressed database volume of roughly 5 TByte, a size accommodated by commodity server hardware. Even the largest SAP BW installation with a size of 100 TB can be accommodated with a reasonable hardware scenario using SAP HANA's compression schemes (see Section 4.4.2). Implementing a tight integration with data management infrastructure from SAP Sybase ranging from data stream engines (SAP Sybase ESP) to SAP Sybase IQ using a dynamic tiering approach, HANA supports enterprise-scale big data requirements and provides a solid foundation to satisfy modern business applications as well as data scientists working on the same data set [98].

Combination of OLTP and OLAP

Since the early days of the SAP HANA project, one its main design tenants was to allow the reunification of transactional and complex analytical workloads in the context of a single database management system [124]. While this seems to contradict the observation that one size does not fit all [141] at first glance, a second glance reveals that there are several scenarios where a system that performs reasonably well in one aspect (e.g., sustaining the OLTP workload of a large enterprise resource management (ERP) system [78]) and excels in another aspect (e.g. very fast OLAP performance) can add significant benefits for end users.

To deal with OLTP scenarios that require a very high throughput, SAP HANA incorporates an in-memory row store that is built on the foundation of the P*TIME system [28]. On an abstract level, the row store shares several key concepts and design choices (e.g. latch-free indexing) with the Hekaton system discussed in Chapter 4.1. However, to avoid redundant data storage in row and columnar format, the goal of the SAP HANA column store is to perform reasonably well for large-scale OLTP operations and excel in allowing complex analytical operations on fresh data without any propagation delays or data redundancy.

For the columnar data store in SAP HANA, the challenge was to design the system in a way that allows for handling the characteristic workloads of an OLTP system (e.g. primary-key based lookup operations, key/foreign key joins with high selectivity, in-place update operations, full record retrieval) without compromising OLAP performance [138]. To this end, several OLTP optimizations were introduced:

1. **Query plan generation.** In the initial design, the query compiler did not use the concept of prepared statements for the SAP HANA column store. While this concept works excellent for

complex OLAP workloads, OLTP queries are often very simple. Therefore, the query compilation overhead (of a few milliseconds) is prohibitively expensive, mandating the use of prepared statements and plan shortcuts for OLTP.

2. **Parallelization considered harmful.** The runtime system and query processing operators in SAP HANA make heavy use of parallelization on all possible levels as discussed in Section 4.4.5. While this is extremely beneficial to achieve good reporting performance for complex analytical queries, parallelization can be considered harmful for OLTP operations: For example, performing a simple key/foreign key join operation for a very small result set with parallelization enabled, easily increases the runtime by a factor of two due to the high overhead of job scheduling and context switches. Consequentially, the query compiler and runtime system in SAP HANA put a lot of effort in deciding on the right parallelization strategy depending on query complexity and system workload.

3. **Concurrent OLTP and OLAP scheduling.** The various design choices and optimizations discussed in this section enable the columnar store in SAP HANA to handle high-volume OLTP workloads. While the system does not excel at this task due to the drawbacks of columnar storage and dictionary compression for transactional workloads, this approach enables the execution of complex analytical queries without data redundancy, data replication or additional tuning (e.g. heavy indexing or usage of materialized views). However, additional care needs to be taken to ensure that complex analytical queries do not impact the (high amount of) concurrent OLTP operations: blocking or stalling them (e.g. by heavily parallelizing OLAP queries leading to resource unavailability) can easily cause queuing effects in the transactional workload and have negative impact on the overall system. Consequentially, the workload management components in HANA ensure that OLTP operations are never stalled by careful workload monitoring and class-based query scheduling,

i.e., transactional operations are a high-priority query class that is preferred over OLAP queries [125].

4.4.2 Data Organization

The in-memory data representation of SAP HANA is designed to fulfill two fundamental requirements: processing performance of complex, analytical queries needs to be maximized as well as the memory footprint should be minimized at the same time to allow for handling large data volumes with reasonable amounts of hardware.

Dictionary Compression. All data stored in columnar format uses dictionary compression. Besides reducing the memory footprint, dictionary compression also enables highly efficient scan operations as discussed in Section 4.4.5. Additionally, the dense domain coding can also be leveraged in other parts of the system and often provides interesting and non-obvious benefits (e.g. for keeping very compact histograms [101]). Consequentially, every column stored in SAP HANA consists of the two core data structures depicted in Figure 4.20, with the *dictionary* storing all the distinct values that occur in a particular column and the *index vector* indicating which value is stored in a particular row of this column. While Figure 4.20 depicts the positions together with the corresponding value/valueID, these positions are purely virtual and can be calculated from the offset in the data structure. Each column can optionally be extended with an index to facilitate processing highly selective queries (e.g. OLTP workloads as discussed in Section 4.4.1). Indexes leverage dictionary compression and are implemented as an inverted list, mapping valueIDs to the corresponding row numbers.

Delta/Main Concept. To achieve a maximum of query processing performance, SAP HANA keeps the dictionaries *sorted*. This enables the runtime system to perform comparisons directly on the dictionary-encoded values (e.g. whenever $a < b$ holds for a particular set of values, the same holds true for corresponding valueIDs). While this property is very beneficial for range queries, it imposes additional challenges for update processing: when adding a new literal to the dictionary, the valueID of the successor entries in the dictionary need to be incremented,

Index Vector		**Dictionary**	
position	valueID	position	value
1	3	1	Adam
2	2	2	Adriana
3	1	3	Alexa
4	3		
5	1		

Figure 4.20: Columnar Data Organization

potentially resulting in a large number of entries in the index vector that need to be updated. Typically, this makes update operations prohibitively expensive. To avoid this problem, all columnar tables in HANA are separated into two parts, the *main* and the *delta*.

The main part of a column consists of a sorted dictionary and an index vector. All insert operations into a table are handled by the delta part consisting of an index vector (similar to the main), an *unsorted* dictionary (to avoid the re-coding problem outlined above), as well as a tree-based index structure to provide fast access to the dictionary. Whenever a new entry is inserted into the delta, the index is used to check whether the corresponding literal is already stored in the dictionary. If so, the corresponding valueID is retrieved, otherwise a new entry is added to the tail of the dictionary structure and to the tree index. Updates are modeled as a sequence of deletes and inserts. Rows from the main part that need to be deleted are tracked by a separate bit vector.

While the delta part of a table avoids to re-encode large parts of the index vector in case of updates, query processing has to consult both data structures. Moreover, most of the core data access primitives on the delta are not as efficient as for the main part. Moreover, its additional index structure consumes main memory, thus conflicting with the goal of high data compression. To avoid these drawbacks, the system periodically consolidates the delta into the main in an operation called *delta merge*. The optimal point in time when to perform a delta merge results on a cost-based decision automatically taken by the system based on a cost function that (among others) considers the size of delta and main (in terms of main memory, number of records, and disk footprint), as well as the current system workload.

Compression Techniques

To keep the overhead of compression at a minimum, SAP HANA only uses lightweight compression techniques that allow efficient access both to individual values and to blocks of values. The basis for all of the following techniques is *domain coding* that is applied for each column independently [107]. For domain coding, each of the n distinct values of a column gets assigned an integer code between 0 and $n-1$ and each value of the column is replaced by its code word stored in the dictionary. The sequence of code words (index vector) is then stored as a bit stream where every code word just uses a fixed number of $b = \lceil log\ n \rceil$ bits. This index vector is then subject to further compression using one of several techniques.

Prefix coding. This is the simplest compression technique, where repetitions of the same value at the start of a column are deleted and replaced by one value and its frequency. If the most frequent value appears not only in the prefix but is also scattered among the other values, then good compression can be achieved with *sparse coding*. With sparse coding, all appearances of the most frequent value are deleted and their positions are stored in a bit vector with prefix coding applied.

Cluster coding. With this technique the data is blocked and only blocks with a single distinct value are compressed by storing only the single value. Additionally, a bit vector is needed to indicate which blocks are compressed, in order to be able to reconstruct the original column. If the data blocks contain more than one but still only few distinct values, *indirect coding* can be used.

Indirect coding. In this procedure, domain coding is applied to suitable blocks, which adds the indirection of requiring a separate mini-dictionary for each block. To reduce the number of dictionaries and hence the memory consumption, one dictionary can be used for successive blocks as long as any new entry in the dictionary does not increase the number of bits required to code the entries. A block is only compressed if and only if the dictionary and the references take less space than the original (domain coded) data.

Run length encoding. The final compression technique presented here is a slightly modified variant of *run-length encoding*, which compresses runs of repeated values to a single value for each run together with the number of repetitions in the run. The cumulative sum of the frequencies of the previous values yields the start position of each value in the run. As this imposes a big overhead SAP HANA slightly reduces the compression by storing only the start position and not the number of repetitions.

The column ordering problem. For maximum compression the presented techniques favor long ranges of the same values as a result of a sorted representation. Unfortunately, the columns cannot be sorted independently of each other, because the position of a value reflects the position of its row in the table. Decoupling the value positions from the row positions would require an expensive mapping of values to rows. The sequences of code words with the same value are called *remainder segments*. The column ordering problem consists in identifying the optimal global sort order to achieve largest possible remainder segments. Since the solution space grows exponentially with the number of columns, it is practically impossible to find an optimal solution to the column ordering problem. Therefore SAP HANA deploys several greedy heuristics that represent a compromise between reduced run time and reduced memory consumption. The simplest heuristic sorts the most frequent value in each column to the top, taking account of the dependencies between the columns. Using more advanced methods, the remainder segments can be further sorted without disrupting the existing ordering.

Data Aging

Business data is typically accessed frequently in the beginning of its lifecycle, when the corresponding business processes are still active. After some time, dependent on its status, the data is not accessed as part of the regular working set any more during normal operation. However, it may happen that old data becomes interesting again, for example when running analysis over multiple years, when accessing the order history of a customer, or during an audit. This observation is often characterized with the data temperature metaphor. Operationally relevant data is called ‘hot’, and data that is no longer accessed during normal operation is called ‘cold’. We use ‘current’ and ‘historical’ as synonyms. The data temperature can be used to horizontally partition the application data for optimizing resource consumption and performance. The process of moving data between the different partitions (i.e. from current to historical partitions) is called data **aging**.

The goal of aging is to both reduce the main memory footprint and speed up database queries by keeping only operationally relevant data in main memory. Historical data may be stored, loaded, and accessed differently, but remaining accessible via SQL. An aging concept has to match the needs and specifics of the application: applications store data differently, have different table schemes, and have different access patterns. For BW on SAP HANA, aging works almost transparently. BW partitions the data by range using a time dimension which is a primary key column, making aging an inherent part of the physical database design. Older data gets less frequently requested by users, thus corresponding partitions typically reside on disk.

More generally, business objects often span multiple tables, for example header and lineitem tables. They have statuses (like order fulfillment), but such a status might only be set in one of the tables. Using a status as partitioning column is therefore usually neither sufficient nor always possible as partitioning always relates to the columns of a single table where the tuple is stored in.

In order to maintain a brace for the tables that make up a business object and to be backwards compatible for applications, SAP HANA holds an artificial temperature column for aging-aware tables. The application actively sets values in this column to a date to indicate that the object is closed and the row shall be moved to the cold partition(s). It does so consistently for all related tables of the same business object, or even for a group of related business objects.

From an application perspective, a default restriction to “hot only” means that rows in cold storage are not present anymore, as if they were deleted. Therefore, the applications have to align on the aging criteria and finally pay attention which rows or business objects are moved to the cold partitions in order to always provide correct result sets. For data access, this means that the data is always consistent when it is being read in one of the two ways, either hot only or cold with a given date and everything newer than this date, including the hot partition.

For OLTP processing, the application typically uses a predicate to access hot data only. A generic SQL extension can be used to filter cold data while applying the semantic rules outlined above. Data of cold partitions may be read page-wise, reading only a minimal set of data from disk. This is opposed to the standard load behavior of columns that read everything into memory before processing it. Pages with cold data will be loaded into a page pool where an LRU mechanism is applied.

4.4.3 Indexing

In addition to the primary in-memory columnar store, SAP HANA also implements an in-memory row store based on P*Time [28]. Within P*Time, an optimized protocol called 'optimistic latch free index access protocol' (OLFIT) [27], is used to obtain good indexing performance and scalability in multi-core environments. The OLFIT protocol addresses significant disadvantages of traditional index concurrency control schemes in multi-core environments where frequent latching leads to a high number of cache coherence misses just because of concurrency control–even if there are only read operations and no updates are made. This was identified as a major factor limiting the scalability of index performance for in-memory multiprocessor systems. The OLFIT protocol addresses this problem with an optimistic control scheme that does not require setting latches for read operations. Instead only write operations set a latch, and only for those nodes that are actually changed. In addition, a version number is increased whenever a node is changed. Read operations use an optimistic approach to ensure consistency: they read the latch and if it is not set, they proceed and read the version number before and after the read operation. If the latch was set or if the version number was changed while reading, the read operation is repeated. With this optimization read performance is almost as high as without concurrency overhead and the overall index performance scales almost linearly with the number of processor cores.

4.4.4 Concurrency control

SAP HANA uses row level multi version concurrency control (MVCC) (see [155] for a textbook example) to ensure consistent read operations. MVCC is implemented by all SAP HANA data stores and is used for supporting both transaction level snapshot isolation and statement level snapshot isolation.

Consistent View and Transaction Timestamps For each database reader (a transaction or a statement, depending upon the desired isolation level) SAP HANA provides a consistent view of the database, which contains only the data the reader is allowed to see. To determine what data can be seen by a given reader, visibility information is stored with each data version. For committed versions, the visibility information is based on transaction timestamps. To avoid having to update the timestamp on a large number of data versions committed by a transaction, a level of indirection is used in which all versions committed by a transaction point to a common transaction block, which contains the commit timestamp. This indirection is subsequently and asynchronously removed by lazily updating the data versions info with the commit timestamp. Not yet committed versions are tagged with an identifier of the transaction that created them.

The transaction timestamps (TS) used for MVCC are integers from a commit counter, which is maintained by the transaction manager and incremented after each successful commit. The consistent view for a reader has an associated timestamp (CVTS) which is the value of the commit counter at the time the reader started. Several readers may have the same CVTS if no transaction was committed in between. For a given consistent view all data versions committed after the consistent view timestamp are not visible. Only those not yet committed versions are visible that were created by the reader's own transaction.

Consolidation of Data Versions. Old data versions no longer visible in any potential consistent view are periodically consolidated to free up memory. Version consolidation is done asynchronously as a regular scheduled background job, when triggered by specific system events, or manually by an administrator. The transaction manager maintains a system value MinReadTS, the timestamp of the oldest consistent view that must be kept because there is at least one active reader that needs to access it. Whenever a transaction ends, the transaction manager checks whether this was the last transaction with a consistent view timestamp equal to MinReadTS. If this is the case, MinReadTS can be advanced to the next larger CVTS of an active transaction. Any old data version may be removed if it has a committed successor version that is visible in the consistent view corresponding to MinReadTS, i.e., if the successor version has a timestamp TS $\leq$ MinReadTS.

Write Conflicts. MVCC ensures consistent read operations but does not prevent concurrent write operations on the same data version which can cause associated inconsistencies such as dirty write and lost updates. To prevent concurrent write operations on the same data versions, SAP HANA takes exclusive row-level write locks for each write access request. The transaction manager performs deadlock detection and avoidance by aborting victim transactions. With snapshot isolation, lost updates would occur if transactions were allowed to create new versions of records visible to them for which new versions were committed in the meanwhile by other transactions. SAP HANA detects such write conflicts and aborts the offending operations with a serialization error.

Snapshot Isolation and Uniqueness Constraints. SAP HANA supports uniqueness constraints (for primary key columns and other columns). Checks for violation of uniqueness constraints are not made against the consistent view seen by the write operation. Instead the checks include all existing versions. If a write operation creates a uniqueness conflict with versions committed since creation of the consistent view, the execution of the write statement will be aborted. If there is a constraint violation conflict with an uncommitted version, execution will block until the other transaction is finished or until the waiting write operation is aborted with a timeout.

4.4.5 Query Processing

The performance of the full table scan is critical for the overall query performance of a column-store database system. Compressing the underlying column data format is both an advantage and a challenge, because it reduces the data volume involved in a scan on the one hand but introduces the need for decompression during the scan on the other hand. As all columns in SAP HANA are dictionary encoded and the references to the dictionary entries are bit compressed, decompression speed during scan is a critical task. Vectorizing the scan with vector engines (available for example in all standard Intel processors) allows to execute SIMD (single instructions multiple data) instructions for significantly accelerating the performance. Intel's AVX2 implements vector-vector shift and gather instructions, which play a fundamental role for the performance of SAP HANA's core scan primitives. Since these SIMD instructions work on vectors of machine words such as 8, 16, 32, or 64bit integers, some efficiently implemented unpacking logic is requiring to de-compress column values [158].

The following vectorized steps are performed on 4 to 256 code words in parallel (with 256 bit registers): In a first step, the data is brought into a format that the scan predicate can work on. In the most general case, this means shuffling all bytes containing bits of the code word into the same machine word, cleaning the upper (unused) bits of the machine word, i.e., setting them to zero, aligning the code word to machine word boundaries, i.e. shifting the machine word to the right, and finally storing the result into a buffer. In a second step, the predicate is evaluated. For the range scan, this consists of two comparisons and can be done directly after cleaning by shifting the range once before the scan. Vectorized predicates can be evaluated directly after the alignment to machine word boundaries, while evaluation of other predicates is delayed until a buffer is filled in order to amortize virtual function calls etc. In a third step, the result of the predicate evaluation has to be extracted, be it a bit or an index indicating a match or the unpacked code words in case of an arbitrary predicate. The last step is to store the extracted result. In order to fully benefit from SIMD instructions, many non-trivial optimizations have to be applied to this scheme. Skipping the alignment for range predicates and tightly interweaving unpacking and evaluation of vectorized predicates are two important optimizations. Additionally, we separately optimize every single bit case, i.e., we have a different implementation of the above scheme for every value of b. In some bit cases, a single special SIMD instruction can perform the clean and align step at the same time, while in others, shuffle alone needs three instructions.

Grouping with aggregation is one of the most expensive relational database operators. The dominant cost of aggregation is–as with most relational operators–the movement of the data. In an in-memory database system, the challenge is to design an aggregation operator such that it uses the CPU caches efficiently to overcome the bottleneck to the significantly slower main memory. Traditionally, there are two opposite approaches to implement this operator: hashing and sorting. **Hash aggregation** inserts the input rows into a hash table, using the grouping attributes as key and aggregating the remaining attributes in-place. **Sort aggregation** first sorts the rows by the grouping attributes and then aggregates the consecutive rows of each group. The consensus is that a hash-based aggregation is better if the number of groups is small enough such that the output fits into the cache; the sort-based variant is more efficient if the number of groups is very large. SAP HANA provides a single aggregation operator combining the advantages of both worlds [108]. The overall mechanism is based on sorting by hash values allowing to combine hashing for early aggregation and state-of-the-art integer sorting routines depending on the locality of the data. We tune both the hashing and the sorting routine to modern hardware and devise a simple, yet effective criterion of locality to switch between the two.

Similarly to scan and aggregation operators, SAP HANA provides a rich set of further relational as well as non-relational operators, which are highly tuned for modern hardware, especially to leverage the cache hierarchies of modern CPU as well as consider NUMA architectures of large systems.

4.4.6 Durability and Recovery

Transaction durability in SAP HANA is realized using logging coupled with shadow paging. Any changes made by the transaction are logically logged into a write-ahead log. SAP HANA also writes out savepoints (a.k.a. checkpoints or snapshots in other database systems) at regular intervals. Thus, in order to recover a certain database state, the contents of the last savepoint plus the log since the last savepoint is sufficient. Although this concept is similar to traditional database systems, the complete persistency layer was developed from scratch and designed for distributed in-memory database requirements. The following examples highlight specific design decisions:

- **No traditional buffer cache:** SAP HANA employs a generic resource container, which manages loaded in-memory objects (such as data columns and dictionaries or even individual data pages) in a unified way. This resource container cooperates with the memory management subsystem, such that memory pressure will not result in paging, but rather releasing (hopefully) unneeded resources based on a weighted LRU algorithm.

- **Variable page size:** Instead of supporting just a single page size, the subsystem supports several page sizes (powers of 4, from 4KB to 16MB) to optimally use I/O bandwidth for consecutive ranges of columnar data, while at the same time keeping persistent space fragmentation very low. For example, a 20 MB object can be composed of one 16MB and one 4MB page. This also helps to keep the memory space needed for logical-to-physical block translation table minimal.

- **Shared nothing:** The persistent state of individual services in distributed settings is completely separated. Also savepoints and logs of individual services run unsynchronized, with the exception of consistent snapshot.

- **Consistent changes:** Operations on persistent state are grouped into low-level atomic operations called consistent changes. Consistent changes consist of writing REDO log entries, writing UNDO/CLEANUP entries, and modifying the persistent state, which may be an actual modification on pages loaded in resource container, or just an intent of modification by marking some pages dirty and letting materialization callback materialize them on savepoint. Consistent changes are guaranteed to be either persisted in the savepoint in their entirety, or not at all, i.e. the higher-level implementation just needs to provide REDO action to repeat any action executed online to create identical new logical state. UNDO/CLEANUP actions are executed automatically by the persistency subsystem either during transaction rollback (to undo the action) or during asynchronous garbage collection (to finally delete old states after any potential MVCC-based readers are gone).

Aside from failure recovery, the SAP HANA persistency subsystem is the basis for several advanced features, such as consistent on-line backup based on coordinated consistent snapshot across services (full and delta). This implies that the distributed database is recoverable from pure data backup even if the log is lost. The subsystems also provide point-in-time recovery as well as a high-availability solution with initial data shipping and continuous log shipping (near real-time takeover). The feature of efficient secure data deletion by coupling page encryption with targeted savepoints is another example of features required in enterprise database solutions.

4.5 Other Systems

The current main-memory database landscape by no means consists of the four primary systems we surveyed in previous chapters. The field is rich with both commercial and academic systems that feature novel and interesting architectures and techniques that achieve high performance on main-memory data. This section summarizes these systems.

4.5.1 solidDB

solidDB is a high performance database system originally developed by Solid Information Technology, a privately held company founded in 1992 in Helsinki Finland. solidDB was bought by IBM in 2007 and was subsequently sold to UNICOM global in 2014. The database engine continues to be developed and is used mainly as an embedded database in telecom and network software deployments.

Architecturally, solidDB is a hybrid database that consists of both a disk-based and main-memory optimized engine [93], where a single SQL statement may access data from either engine (this summary focuses on the main-memory engine). Like other main-memory engines, solidDB does not use page-oriented storage to avoid indirection overhead when accessing records.

solidDB indexes data using a Vtrie (for variable-length trie), a variant of the trie data structure that indexes leaf nodes structured similarly to B+-tree leaf nodes whose default size is a few cache lines. The Vtrie allows readers to proceed uncontested and latch-free by versioning index nodes, while writers use two-level locking to avoid conflict.

For concurrency control, solidDB uses pessimistic locking [139]; this is different than most current systems. For durability, solidDB uses snapshot-consistent checkpointing [91] to recover to a consistent state. If desired, users can disable transactional logging altogether to recover from the last durable snapshot. solidDB also provides high-availability using a hot standby approach that replicates the redo log.

4.5.2 Oracle TimesTen

TimesTen began as a research project at HP Labs named “Smallbase” and was later spun off into a separate company that was acquired by Oracle. TimesTen continues to be developed and serves as one of Oracle’s high-performance main-memory database solution. This overview covers the current features of Oracle TimesTen as of 2013 [74].

TimesTen is a memory-optimized database engine that can serve many purposes, such as a standalone engine, a transactional cache on top of an Oracle RDBMS, as well as an in-memory repository for interactive BI workloads. The in-memory portion consists of the in-memory database area, a temporary area for run-time allocations, and the in-memory log buffers. Database storage avoids page indirection overhead and uses direct pointers to records. TimesTen also implements several indexing methods including hash, bitmap, and range indexes supported by T-trees [79]. TimesTen also supports columnar compression using dictionary-based encoding. Users can specify to compress columns individually or together in groups.

An interesting feature of TimesTen is its deployment flexibility. While it can be used as a database of record, it is easily deployed as a cache on top of a disk-based Oracle RDBMS. Users can specify that a set of TimesTen tables act as caches against a set of corresponding tables in an Oracle database. Users can also specify how caches are loaded (pre-loaded or dynamically) and synchronized (read-only or updatable).

For transactional concurrency control, TimesTen uses a lockless multi-versioning approach to enable read-write concurrency. It uses row-level locking to handle write-write concurrency. Write-ahead logging along with period checkpointing provide durability in TimesTen. For performance, TimesTen offers a delayed-durability mode that acknowledges a transaction commit after writing a commit record to an in-memory log buffer (without waiting for durability of the commit record). In this mode, log flushes occur every 100 msec, which defines an upper bound on possible data loss.

4.5.3 Altibase

Altibase is an in-memory database provider based in South Korea that was founded in 1999. The company has a large customer base in the Asian market spanning telecommunications, financial, and manufacturing companies. HDB is Altibase's hybrid database system that consists of an engine optimized for memory-resident data and a disk-based engine that resembles a traditional RDBMS architecture [11], where transactions can span both engines. The rest of this section summarizes the features of the main-memory engine.

Altibase stores records on pages, where a memory-optimized table consists of one or more pages in contiguous memory. It seems Altibase uses a page-based organization for two reasons: checkpointing (checkpoints are written at page granularity) and compatibility with the disk-based engine. The engine supports several index types, including hash indexes, range indexes, and spatial indexes. Indexes are not stored in page format nor made durable and are rebuilt from scratch during recovery [11].

To support transactions, the Altibase in-memory engine uses a multi-version concurrency control method that creates a new version on each update (deletes create a tombstone). Each record points to its newest version (null for the most up-to-date record) forming a version chain. Durability is provided through the use of write-ahead-logging and fuzzy checkpointing. Altibase uses a page latch when copying page state to the transaction log buffer, but implements a latch-free checkpointing process when writing page data to the checkpoint file [11].

4.5.4 MemSQL

MemSQL [2, 136] is a hybrid database aimed at performing both transactions and analytics. Architecturally, MemSQL is designed to scale out on commodity hardware. The engine is organized into two tiers: (1) *aggregator nodes* interface with the client and perform query routing, parsing, and optimization while (2) *leaf nodes* provide the in-memory storage and query processing functionality. Internally, MemSQL avoids page-based indirection and uses lock-free skiplists to index data that contain direct memory pointers to records. MemSQL provides durability by flushing a redo-only transaction log. Like Hekaton, MemSQL uses the log to periodically create full snapshots of the database. The database recovers using the last valid checkpoint and is made consistent by replaying the redo-only transaction log [100].

4.5.5 Silo

Silo [151] is a high-performance main-memory database built atop the Masstree [97]. Silo supports transactions in the form of stored procedures. Architecturally, a Silo table consists of a set of in-memory records indexed on a primary key, along with any number of secondary indexes (possibly none). Indexes store direct record pointers, there is no page-based indirection.

Silo is designed to scale on large multi-core machines. The key to Silo's performance and scalability is that it reduces writes to "hotspots" in shared memory. Instead of assigning a transaction a unique timestamp (one example of a hotspot requiring an atomic fetch-and-add for every transaction), Silo uses an epoch-based approach where all transactions read a global epoch number E that increments every so often (40 ms by default). E occupies the high-order bits of each transactions id and defines the serial order of transactions.

Silo provides serializable transactions as follows. Transactions track their read sets and buffer their write sets. When committing, a transaction acquires all record locks in its write set (aborting if it cannot), generates its id by reading the current global epoch, validates its read set (aborting if validation fails), and finally installs its write set along with its transaction id and releasing the write locks. Phantom protection is provided by versioning index leaf nodes. During validation a transaction checks the leaf version(s) it read against the current version(s) in the index and aborts if there are discrepancies.

Silo exploits multi-core parallelism throughout its durability and recovery design [159]. Silo performs redo-only logging in parallel an multiple disks, ensuring that updates for an epoch are durable before updates from epochs with larger values. Silo performs database checkpoints in parallel by cooperatively scheduling checkpoint worker threads. Recovery is parallelized by replaying checkpoint files in parallel to rebuild in-memory state, followed by parallel log replay to bring the database to a consistent state.

5

Active Research and Future Directions

In the previous sections, we surveyed the architecture and implementation issues of modern main-memory database systems. Much of the technology we surveyed is either shipping in production systems, or is a mature research prototype that serves as the basis for ongoing research.

In this section, we summarize active areas of research in main-memory database systems that is more speculative in nature. We first cover "cold data management", which looks at how to manage records that are rarely accessed and might not need to remain in memory. Second, we cover the use of transactional memory, and how it might be used to ease the implementation of highly-concurrent main-memory system. We then summarize the use of non-volatile RAM in database systems: a more speculative research area exploring at how to adapt systems to efficiently use a new high performance, byte-addressable storage medium.

5.1 Cold Data Management

In a majority of OLTP workloads, some records are "hot" and accessed frequently (the working set), while others are "cold" and accessed infrequently. Good performance of main-memory database systems depends on the hot records residing in memory. Cold records, on the other hand, may not need to remain in memory. An active area of research is how to manage "cold" data in main-memory systems. The goal of this work is to efficiently identify and move cold records off the system's critical hot path. Research in this area is motivated by two key observations:

1. *Skew in OLTP workloads.* Real-life transactional workloads typically exhibit considerable access skew. For example, package tracking workloads for companies such as UPS or FedEx exhibit time-correlated skew. Records for a new package are frequently updated until delivery, then used for analysis for some time, and after that accessed again only on rare occasions. Another example is the natural skew found on large e-commerce sites such as Amazon, where some items are much more popular than others. Such preferences may change over time but typically not very rapidly.

2. *Economics.* It is significantly cheaper to store cold data on secondary storage rather than in DRAM. High-density server class DRAM comes at a large premium, making a medium like flash attractive for cold data. In fact, calculating the 5-minute rule proposed by Gray and Putzolu [56] using current hardware prices reveals that a 200 byte record should remain in memory if it is accessed at least every 60 minutes. This time window decreases as record size increases; a more complete analysis can be found in [88].

In the rest of this section we summarize three techniques for managing cold in modern main-memory systems. In general, work on cold-data management is still in the research phase; we do not know of a full scale implementation in production.

HyPeR Compression

HyPer's cold-data management scheme [48] separates cold transactional data from the hot data and compresses it in a read-optimized format for OLAP queries. This technique performs cold/hot data classification at the virtual memory page level, which is the granularity of its data organization. Classification piggybacks on the CPUs memory management unit setting of dirty page flags used for page frame relocation. HyPer pins pages in memory, so it is able to read and reset dirty flags to help classify cold and hot pages. After classification, chunks of cold records are compressed to reduce memory consumption. The cold chunks are stored on huge virtual memory pages and made immutable to allow for use in OLAP snapshots.

Data Blocks. Subsequent work in HyPer on cold data management produced Data Blocks: self-contained containers that store one or more attribute chunks in a byte-addressable compressed format [75]. The goal is to conserve memory while retaining the high OLTP and OLAP performance. By maintaining a flat structure without pointers, Data Blocks are also suitable for eviction to secondary storage, thereby reducing the DRAM footprint of HyPer. A Data Block contains all data required to reconstruct the stored attributes and our novel light-weight Positional SMA (PSMA) index structures, but no metadata such as schema information, as replicating this in each block would waste space. Although orthogonal to this work, Data Blocks have further been designed with upcoming secondary storage solutions in mind, including non-volatile RAM (NVRAM) and byte-addressable flash storage. Data stored in Data Blocks on such storage devices can directly be addressed and read without bringing the whole Data Block into DRAM.

Hekaton Siberia

Hekaton's Siberia approach splits the database into hot and cold storage [42]. This split is beneath the cursor interface of Hekaton, hiding the physical location of a record from higher software layers. Siberia employs a novel cold data classification techniques that logs a sample of record accesses, then analyzes the log in parallel to find the hot and cold records [88]. Siberia uses small approximate, but conservative, point and range filters [10] over the cold store. To prevent unnecessary accesses to cold storage during processing. Siberia also dovetails with

Hekaton's optimistic multi-version concurrency control scheme and is capable performing transactions that span both hot and cold stores, as well as migrating data to and from cold storage while the database is online and active.

Anti-Caching

Anti-caching [34] is a memory-oriented DBMS design built into H-Store that allows the system to manage databases that are larger than the amount of memory available without incurring the performance penalty of a disk-oriented system. When the amount of in-memory data exceeds an administrator-defined threshold, the DBMS moves data to disk to free up space for new data. The DBMS maintains in-memory "tombstones" for each evicted tuple. When a running transaction attempts to access an evicted tuple through its tombstone, the DBMS aborts that transaction and fetches that it needs from the anti-cache without blocking other transactions. Once the data that the transaction needs is moved back into memory, the transaction is restarted.

At runtime, H-Store monitors the amount of main memory used by the database. When the size of the database relative to the amount of available memory on the node exceeds some administrator-defined threshold, the DBMS "evicts" cold data to the anti-cache in order to make space for new data. To do this, the DBMS constructs a fixed-size *block* that contains the least recently used (LRU) tuples from the database and writes that block to the anti-cache. It then updates a memory-resident catalog that keeps track of every tuple that was evicted. When a transaction accesses one of these evicted tuples, H-Store switches that transaction into a "pre-pass" mode to learn about all of the tuples that the transaction needs. After this pre-pass is complete, H-Store then aborts that transaction (rolling back any changes that it may have made) and holds it while the system retrieves the tuples in the background. Once the data has been merged back into the in-memory tables, the transaction is released and restarted.

Anti-caching classifies cold data using the LRU replacement policy on a per-table basis. Statistics to identify LRU recency is stored in the in-memory tuple header. Anti-caching uses a block table for storing cold records, where blocks are similar in spirit to disk pages. An in-memory eviction table is used to map evicted tuples to the appropriate block on secondary storage. Anti-caching uses a special single-partition transaction to serialize and migrate cold tuples to block storage. Transactions that touch a cold record are aborted and restarted; before restarting the transaction goes into a pre-pass phase that tries to identify all cold records that will be accessed and migrates these records into memory. Once the pre-pass phase is complete the transaction restarts.

5.2 Transactional Memory

Transactional memory (TM) allows for atomic execution of arbitrary loads and stores within a critical section, relieving the programmer from thinking about fine-grained thread-level concurrency. Transactional memory research started in the late seventies [94] with the first practical implementation proposed by Herlihy and Moss in the early nineties [64]. However, use of TM to simplify the implementation of high performance database systems never took hold; this was primarily due to the significant performance hit due to transactions being implemented in software.

The recent release of hardware transactional memory (HTM) by Intel and IBM has brought about a resurgence in research exploring databases and transactional memory. While details differ, the basic idea is to piggyback the HTM implementation on existing CPU features. For instance, CPU caches are used to store transaction buffers and provide isolation. Also, the CPU cache coherency protocol can be used to detect transaction conflicts. All of this leads to a very low-overhead transactional memory implementation.

While HTM is efficient, it does have limitations in its current form that have been experimentally verified in a number of recent works [71, 86, 95]. A primary constraint is that the read and write set of a transaction must fit in cache in order for it to be executed. Thus many properties may limit a transaction's size including: cache capacity, cache set associativity, hyper-threading, TLB capacity and others. Another constraint is on transaction duration. Many hardware events, such as interrupts, context switches or page faults, will abort a transaction. Furthermore, conflict detection is usually done at the granularity of a cache line. This may lead to cases of false sharing where aborts occur due to threads accessing and modifying separate items on the same cache line. Given these constraints, a current open question is: *how does HTM fit reliably into high performance database systems?* The rest of this section summarizes recent research in this area.

5.2.1 Increasing Concurrency with HTM

The HyPeR team explored how to exploit HTM to achieve concurrency within main-memory database systems [86]. Their approach achieves thread-level parallelism by breaking a transaction into individual record accesses (read or update), where each access executes within a hardware transaction. This approach plays nicely with HTM limitations since single-record operations are short-lived and fairly predictable. To implement transactions, these HTM accesses are "glued together" using timestamp ordering concurrency control.

DBX [153] is a prototype OLTP system built to evaluate HTM as a general-purpose solution to implementing transactions. DBX uses an optimistic concurrency control approach that tracks its read set and buffers its writes. Before commit, transactions validate the read set and abort if any concurrent transaction has modified a record in the set. To apply writes, DBX uses a single HTM transaction to install its updates, thus synchronizing among concurrently executing threads. Unlike HyPeR, DBX attempts to batch all record updates within a single HTM transaction.

5.2.2 HTM and Indexing

Karnagel et al. explored using HTM *lock elision* as a way to improve thread-level concurrency in a B+-tree and in SAP Hana's delta storage index [71]. Programming with lock elision is similar to programming with locks (latches), except lock elision first attempts to execute a critical section transactionally, and only if the transaction aborts will it execute the critical section by acquiring the lock. The benefit of lock elision is that it provides optimistic concurrency for programs that use simple coarse grain locks. This work found that using HTM with a global elided lock leads to excellent performance for databases with small fixed-size keys but that care must be taken within the index implementation (in the case of the delta index) to avoid spurious HTM aborts due to false cache line sharing.

Makreshanski et al. explore the interplay between a high performance lock-free B+-tree and an HTM-based B+-tree that achieve parallelism through use of a global elided lock [95]. The work makes three main observations. First, HTM is an unattractive general-purpose solution for achieving concurrency within an index, since single read-only operations can abort when larger key and page sizes are used within an index (aborts due to transaction size and associativity conflict). Furthermore, fine-grained concurrency techniques like lock-crabbing cannot be implemented with HTM since a cacheline cannot be removed from a transaction. Second, there are fundamental performance advantages to a lock-free design since it never abort readers, whereas an HTM-based approach can abort readers on data conflict. Third, using HTM to implement a high performance multi-word compare-and-swap can greatly decrease the complexity of lock-free index design at a cost of roughly 10-15% overhead.

Recent work on DrTM explores building a scale-out transactional key-value store using HTM and RDMA [154]. The challenge in building DrTM involves avoiding the limited working set of HTM transactions as well as avoiding RDMA calls within an HTM transaction, which causes an automatic abort due to an interrupt. To avoid long transactions, DrTM uses transaction chopping to limit transaction size. To avoid RDMA calls within an HTM transaction, DrTM performs two-phase locking to "lock" remote records, read them into a local cache, update them, and write back local changes after commit.

5.3 Non-Volatile RAM

Non-volatile RAM (or NVRAM) refers to a broad class of technologies under development that promise a blend of the best properties from DRAM and hard disks, such as high performance, byte addressability, persistence after power failure, and energy efficiency. While there are a number of proposals for implementing NVRAM, including phase-change memory (PCM) [128], memristors [144], and STT-MRAM [40], vendors hope to hit a price/performance target below DRAM (in both price and performance) and above Flash/SSD, giving NVRAM a viable position in the storage hierarchy.

While NVRAM is not yet shipping[1], there has been speculative research looking at how to adapt database systems to use NVRAM effectively. The rest of this section classifies and summarizes this work.

5.3.1 Logging

One primary area of NVRAM research is in the context of "traditional" ARIES-style recovery techniques that explore placing the recovery log on NVRAM. Pelley et al. explored a group commit approach (similar to that introduced by Dewitt et al. [37]) that batches recovery log writes in Shore-MT [121]. The goal of this work is to reduce the number of write barriers necessary to ensure correct ordering of the log on NVRAM. Wang and Johnson [152] explore a distributed logging protocol in Shore-MT that equips each log with an NVRAM-based write buffer. Transactions are committed once writing a commit record to the NVRAM buffer, thereby avoiding a centralized logging bottleneck. A main problem addressed by this work is how to guarantee persistence of distributed log records given processor caches are volatile. The solution is to use a passive group commit protocol that tracks when all records required to ensure durability of a transaction are flushed from CPU cache to NVRAM. Fang et al. explore a similar approach using the solidDB log manager [45]. This work addresses the problem of detecting partial writes and the implications on database recovery.

MARS [31] rethinks ARIES-style logging and recovery to address the unique characteristics of NVRAM. MARS is a novel write-ahead logging scheme that performs redo-only logging. The key feature of MARS is the introduction of a multi-part atomic write primitive called

[1]Battery-backed RAM (or NVDIMM) can be considered a form of NVRAM and is available today. However, NVDIMM will follow price/performance trends similar to DRAM.

an *edible atomic write* (or EAW). An EAW is a set of per-object redo log entries that can be updated in place multiple times during transaction execution. On commit, the storage hardware copies the updates to their target locations; this copy is guaranteed to succeed in the event of a power failure.

5.3.2 Storage

Another primary area of research involves storage and programming models optimized for byte-addressable NVRAM. REWIND [29] is a lightweight user-mode library for implementing arbitrary persistent data structures in NVRAM. Users update REWIND data structures using a transaction interface. Underneath the hood, REWIND transparently manages a log of updates and manages commit and recovery of transactions.

SOFORT [115] is a storage engine designed for a two-level hierarchy of DRAM and NVRAM. The engine does not perform logging. It uses multi-version concurrency control to update NVRAM directly with new versions. The goal for SOFORT is to achieve nearly instant recovery; since updates are persisted in NVRAM, there is no need to "rebuild" the database from checkpoints and a recovery log. To this end, SOFORT intelligently chooses which part of the database engine is implemented in NVRAM as opposed to purely volatile DRAM in order to bound recovery time.

Arulraj et al. [14] experimentally evaluate three different storage engines on an Intel NVRAM hardware emulator. The storage approaches evaluated are: in-place updates, copy-on-write, and log-structured updates. This work also implements an NVRAM-optimized variant for each approach to address CPU overhead, storage footprint, and wear-leveling. The results from this evaluation suggest that an NVRAM-optimized in-place update storage approach is ideal in terms of overhead and device wear-leveling. Like SOFORT, this study observes that updating the database directly leads to almost instantaneous restart after a crash.

References

[1] Intel Xeon Processor E5-2699 v3 Specification Sheet. http://intel.ly/1xx6aZz.

[2] MemSQL. http://www.memsql.com.

[3] VoltDB. http://www.voltdb.com.

[4] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-Stores vs. Row-Stores: How Different are they Really? In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 967–980, 2008.

[5] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 23–34, 1995.

[6] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, 1987.

[7] Rakesh Agrawal and David J. DeWitt. Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation. *ACM Transactions on Database Systems, TODS*, 10(4):529–564, 1985.

[8] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. Dbmss on a modern processor: Where does time go? VLDB, pages 266–277, 1999.

[9] M.-C. Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *Proceedings of the VLDB Endowment, PVLDB*, 5(10):1064–1075, 2012.

[10] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *Proceedings of the VLDB Endowment, PVLDB*, 6(14):1714–1725, 2013.

[11] Altibase. *Altibase HDB Administrator's Manual*, 5 2015.

[12] Peter M. G. Apers, Martin L. Kersten, and Hans Oerlemans. PRISMA Database Machine: A Distributed, Main-Memory Approach. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, pages 590–593, 1988.

[13] Peter M. G. Apers, Carel A. van den Berg, Jan Flokstra, Paul W. P. J. Grefen, Martin L. Kersten, and Annita N. Wilschut. PRISMA/DB: A parallel main memory relational DBMS. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 4(6):541–554, 1992.

[14] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 707–722, 2015.

[15] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 362–373, 2013.

[16] Jerry Baulier, Philip Bohannon, S. Gogate, C. Gupta, S. Haldar, S. Joshi, A. Khivesera, Henry F. Korth, Peter McIlroy, J. Miller, P. P. S. Narayan, M. Nemeth, Rajeev Rastogi, S. Seshadri, Abraham Silberschatz, S. Sudarshan, M. Wilder, and C. Wei. DataBlitz Storage Manager: Main Memory Database Performance for Critical Applications. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 519–520, 1999.

[17] Philip A. Bernstein and Nathan Goodman. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 285–300, 1980.

[18] Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[19] Philip A. Bernstein and Nathan Goodman. A sophisticate's introduction to distributed concurrency control (invited paper). In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 62–76, 1982.

[20] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[21] Dina Bitton, Maria Hanrahan, and Carolyn Turbyfill. Performance of Complex Queries in Main Memory Database Systems. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 72–81, 1987.

[22] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 37–48, 2011.

[23] Philip Bohannon, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, S. Seshadri, and S. Sudarshan. The Architecture of the Dalí Main-Memory Storage Manager. *Multimedia Tools Appl.*, 4(2):115–151, 1997.

[24] Michael J. Carey and Miron Livny. Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication. Proceedings of the International Conference on Very Large Data Bases, VLDB, pages 13–25, 1988.

[25] Michael J. Carey and Michael Stonebraker. The Performance of Concurrency Control Algorithms for Database Management Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 107–118, 1984.

[26] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Record*, 39:12–27, 2011.

[27] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 181–190, 2001.

[28] Sang Kyun Cha and Changbin Song. P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 1033–1044, 2004.

[29] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *Proceedings of the VLDB Endowment, PVLDB*, 8(5):497–508, 2015.

[30] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 235–246, 2001.

[31] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction Support for Next-Generation, Solid-State Drives. In *Proceedings of the ACM Symposium on Operating Systems Principles, SOSP*, pages 197–212, 2013.

[32] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of the USENIX Annual Technical Conference, ATC*, pages 21–34, June 2012.

[33] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 3:48–57, 2010.

[34] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-caching: A New Approach to Database Management System Architecture. *Proceedings of the VLDB Endowment, PVLDB*, 6(14):1942–1953, 2013.

[35] Kalen Delaney, Paul Randal, Kimberley Tripp, Conor Cunningham, Adam Machanic, and Ben Navarez. *SQL Server 2008 Internals.* Microsoft, 2009.

[36] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 228–237, 1986.

[37] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation Techniques for Main Memory Database Systems. *SIGMOD Record*, 14(2):1–8, 1984.

[38] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 1243–1254, 2013.

[39] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, NSDI*, pages 401–414, 2014.

[40] Alexander Driskill-Smith. Latest Advances and Future Prospects of STT-RAM. In *Non-Volatile Memories Workshop*, 2010.

[41] Margaret H. Eich. A Classification and Comparison of Main Memory Database Recovery Techniques. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 332–339, 1987.

[42] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *Proceedings of the VLDB Endowment, PVLDB*, 7(11):931–942, 2014.

[43] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment, PVLDB*, 8(11):1190–1201, 2015.

[44] Jose M. Faleiro and Daniel J. Abadi. Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold? In *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2017.

[45] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. High Performance Database Logging using Storage Class Memory. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 1221–1231, 2011.

[46] Peter A. Franaszek and John T. Robinson. Limitations of concurrency in transaction processing. *ACM Transactions on Database Systems, TODS*, 10(1):1–28, 1985.

[47] Craig Freedman, Erik Ismert, and Per-Åke Larson. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Engineering Bulletin*, 37(1):22–30, 2014.

[48] Florian Funke, Alfons Kemper, and Thomas Neumann. Compacting Transactional Data in Hybrid OLTP & OLAP Databases. *Proceedings of the VLDB Endowment, PVLDB*, 5(11):1424–1435, 2012.

[49] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 4(6):509–516, 1992.

[50] Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Data Engineering Bulletin*, 8(2):3–10, 1985.

[51] Shahram Ghandeharizadeh, David J. DeWitt, and Waheed Qureshi. A Performance Analysis of Alternative Multi-Attribute Declustering Strategies. *Proceedings of the ACM Conference on Management of Data, SIGMOD*, 21(2):29–38, 1992.

[52] Vibby Gottemukkala and Tobin J. Lehman. Locking and Latching in a Memory-Resident Database System. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 533–544, 1992.

[53] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[54] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 209–218, 1993.

[55] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of Locks in a Large Shared Data Base. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 428–451, 1975.

[56] Jim Gray and Gianfranco R. Putzolu. The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 395–398, 1987.

[57] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[58] Le Gruenwald and Margaret H. Eich. MMDB Reload Algorithms. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 397–405, 1991.

[59] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[60] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[61] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phil Garrett, and Andreas Reuter. Group Commit Timers and High Volume Transaction Systems. In *Proceedings of the International Workshop on High Performance Transaction Systems, HPTS*, pages 301–329, 1989.

[62] Joseph M. Hellerstein and Michael Stonebraker. Readings in Database Systems. chapter Transaction Management, pages 238–243. 4th edition, 1998.

[63] Joseph M. Hellerstein, Michael Stonebraker, and James R. Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[64] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the International Symposium on Computer Architecture, ISCA*, pages 289–300, 1993.

[65] M. Heytens, S. Listgarten, M-A. Neimat, and K. Wilkinson. Smallbase: A main-memory dbms for high-performance applications. Technical report, Hewlett-Packard Laboratories, 1995.

[66] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 469–477, 1995.

[67] H. V. Jagadish, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S. Sudarshan. Dalí: A High Performance Main Memory Storage Manager. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 48–59, 1994.

[68] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: A Scalable Approach to Logging. *Proceedings of the VLDB Endowment, PVLDB*, 3(1-2):681–692, 2010.

[69] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision Locks. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 143–147, 1981.

[70] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment, PVLDB*, 1(2):1496–1499, 2008.

[71] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehne. Improving In-Memory Database Index Performance with Intel Transactional Synchronization Extensions. In *Proceedings of the IEEE Symposium on High Performance Computer Architecture, HPCA*, pages 476–487, 2014.

[72] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 195–206, 2011.

[73] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. Generating Code for Holistic Query Evaluation. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 613–624, 2010.

[74] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Engineering Bulletin*, 36(2):6–13, 2013.

[75] Harald Land, Tobias Mühlbauer, Florian Funke, Peter Boncz, Thomas Neumann, and Alfons Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 311–326, 2016.

[76] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. Massively Parallel NUMA-aware Hash Joins. In *Proceedings of the International Workshop on In-Memory Data Management, IMDM*, 2013.

[77] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proceedings of the VLDB Endowment, PVLDB*, 5(4):298–309, 2011.

[78] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krüger, and Martin Grund. High-Performance Transaction Processing in SAP HANA. *IEEE Data Engineering Bulletin*, 36(2):28–33, 2013.

[79] Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 294–303, 1986.

[80] Tobin J. Lehman and Michael J. Carey. Query Processing in Main Memory Database Management Systems. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 239–250, 1986.

[81] Tobin J. Lehman and Michael J. Carey. A Recovery Algorithm for a High-Performance Memory-Resident Database System. Proceedings of the ACM Conference on Management of Data, SIGMOD, pages 104–117, 1987.

[82] Tobin J. Lehman, Eugene J. Shekita, and Luis-Felipe Cabrera. An Evaluation of Starburst's Memory Resident Storage Component. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 4(6):555–566, 1992.

[83] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-core Age. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 743–754, 2014.

[84] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment, PVLDB*, 9(3):204–215, 2015.

[85] Viktor Leis, Alfons Kemper, and Thomas Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2013.

[86] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 580–591, 2014.

[87] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. Efficient Processing of Window Functions in Analytical SQL Queries. *Proceedings of the VLDB Endowment, PVLDB*, 8(10):1058–1069, 2015.

[88] Justin Levandoski, Per-Åke Larson, and Radu Stoica. Identifying Hot and Cold Data in Main-Memory Databases. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 26–37, 2013.

[89] Justin Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 302–313, 2013.

[90] Kai Li and Jeffrey F. Naughton. Multiprocessor Main Memory Transaction Processing. *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, DPDS*, pages 177–187, 1988.

[91] Antti-Pekka Liedes and Antoni Wolski. SIREN: A Memory-Conserving, Snapshot-Consistent Checkpoint Algorithm for in-Memory Databases. In *Proceedings of the International Conference on Data Engineering, ICDE*, page 99, 2006.

[92] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, NSDI*, pages 429–444, 2014.

[93] Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, and Katriina Vakkila. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *IEEE Data Engineering Bulletin*, 36(2):14–20, 2013.

[94] David B. Lomet. Process Structuring, Synchronization, and Recovery Using Atomic Actions. In *Proceedings of the ACM Conference on Language Design for Reliable Software*, pages 128–137, 1977.

[95] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. *Proceedings of the VLDB Endowment, PVLDB*, 8(11):1298–1309, 2015.

[96] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory OLTP recovery. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 604–615, 2014.

[97] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the European Conference on Computer Systems, EuroSys*, pages 183–196, 2012.

[98] Norman May, Wolfgang Lehner, Shahul Hameed P., Nitesh Maheshwari, Carsten Müller, Sudipto Chowdhuri, and Anil K. Goel. SAP HANA - From Relational OLAP Database to Big Data Infrastructure. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, pages 581–592, 2015.

[99] Memcached. `http://memcached.org/`.

[100] MemSQL Durability and Recovery. `http://bit.ly/1nqM2qK`.

[101] Guido Moerkotte, David DeHaan, Norman May, Anisoara Nica, and Alexander Böhm. Exploiting Ordered Dictionaries to Efficiently Construct Histograms with Q-error Guarantees in SAP HANA. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 361–372, 2014.

[102] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging. *ACM Transactions on Database Systems, TODS*, 17(1):94–162, 1992.

[103] C. Mohan, B. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems, TODS*, 11(4):378–396, 1986.

[104] Henrik Mühe, Alfons Kemper, and Thomas Neumann. How to Efficiently Snapshot Transactional Data: Hardware or Software Controlled? In *Proceedings of the International Workshop on Data Management on New Hardware, DaMoN*, pages 17–26, 2011.

[105] Henrik Mühe, Alfons Kemper, and Thomas Neumann. The Mainframe Strikes Back: Elastic Multi-Tenancy using Main Memory Database Systems on a Many-core Server. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, pages 578–581, 2012.

[106] Tobias Mühlbauer, Wolf Rödiger, Angelika Reiser, Alfons Kemper, and Thomas Neumann. ScyPer: Elastic OLAP Throughput on Transactional Data. In *Proceedings of the ACM SIGMOD Workshop on Data Analytics in the Cloud, DanaC*, 2013.

[107] Ingo Müller, Cornelius Ratsch, and Franz Färber. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, pages 283–294, 2014.

[108] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-Efficient Aggregation: Hashing Is Sorting. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 1123–1136, 2015.

[109] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment, PVLDB*, 4(9):539–550, 2011.

[110] Thomas Neumann and Alfons Kemper. Unnesting Arbitrary Queries. In *Datenbanksysteme für Business, Technologie und Web, BTW*, pages 383–402, 2015.

[111] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 677–689, 2015.

[112] Christos Nikolaou, Manolis Marazakis, and G. Georgiannakis. Transaction Routing for Distributed OLTP Systems: Survey and Recent Results. *Inf. Sci.*, 97:45–82, 1997.

[113] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. Alphasort: A RISC Machine Sort. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 233–242, 1994.

[114] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John K. Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the ACM Symposium on Operating Systems Principles, SOSP*, pages 29–41, 2011.

[115] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. SOFORT: a Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *Proceedings of the International Workshop on Data Management on New Hardware, DaMoN*, pages 1–7, 2014.

[116] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *Operating Systems Review*, 43(4):92–105, 2009.

[117] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMCloud. *Communications of the ACM*, 54(7):121–130, 2011.

[118] Sriram Padmanabhan. *Data Placement in Shared-Nothing Parallel Database Systems*. PhD thesis, University of Michigan, 1992.

[119] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. PLP: Page Latch-free Shared-everything OLTP. *Proc. VLDB Endow.*, 4(10):610–621, July 2011.

[120] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-Aware Automatic Data Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, 2012.

[121] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage Management in the NVRAM Era. *Proceedings of the VLDB Endowment, PVLDB*, 7(2):121–132, 2013.

[122] Slawomir Pilarski and Tiko Kameda. Checkpointing for Distributed Databases: Starting from the Basics. *IEEE Transactions on Parallel Distributed Systems, TPDS*, 3(5):602–610, 1992.

[123] Holger Pirk, Florian Funke, Martin Grund, Thomas Neumann, Ulf Leser, Stefan Manegold, Alfons Kemper, and Martin L. Kersten. CPU and Cache Efficient Management of Memory-Resident Databases. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 14–25, 2013.

[124] Hasso Plattner. A Common Database Approach for OLTP and OLAP using an In-Memory Column Database. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 1–2, 2009.

[125] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling. In *Proceedings of the Technology Conference on Performance Evaluation and Benchmarking, TPCTC*, pages 97–112, 2014.

[126] Erhard Rahm. A Framework for Workload Allocation in Distributed Transaction Processing Systems. *Journal on Systems Software*, 18:171–190, May 1992.

[127] Jun Rao and Kenneth A. Ross. Making b$^+$-trees cache conscious in main memory. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 475–486, 2000.

[128] Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Yi-Chou Chen, Robert M. Shelby, Martin Salinga, Daniel Krebs, S-H Chen, Hsiang-Lan Lung, and Charles Lam. Phase-Change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[129] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 1539–1551, 2016.

[130] Kun Ren, Alexander Thomson, and Daniel J. Abadi. Lightweight Locking for Main Memory Database Systems. *PVLDB*, 6(2):145–156, 2012.

[131] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. High-Speed Query Processing over High-Speed Networks. *Proceedings of the VLDB Endowment, PVLDB*, 9(4):228–239, 2015.

[132] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Locality-Sensitive Operators for Parallel Main-memory Database Clusters. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 592–603, 2014.

[133] Stephen M. Rumble, Ankita Kejriwal, and John K. Ousterhout. Log-Structured Memory for DRAM-based Storage. In *Proceedings of the USENIX conference on File and Storage Technologies, FAST*, pages 1–16, 2014.

[134] Kenneth Salem and Hector Garcia-Molina. Checkpointing Memory-Resident Databases. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 452–462, 1989.

[135] Kenneth Salem and Hector Garcia-Molina. System M: A Transaction Processing Testbed for Memory Resident Data. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 2(1):161–172, 1990.

[136] Rajkumar Sen, Jack Chen, and Nika Jimsheleishvilli. Query Optimization Time: The New Bottleneck in Real-time Analytics. In *Proceedings of the International Workshop on In-Memory Data Management, IMDM*, pages 1–6, 2015.

[137] Vishal Sikka, Franz Färber, Anil K. Goel, and Wolfgang Lehner. SAP HANA: The Evolution from a Modern Main-Memory Data Platform to an Enterprise Application Platform. *Proceedings of the VLDB Endowment, PVLDB*, 6(11):1184–1185, 2013.

[138] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 731–742, 2012.

[139] IBM Knowledge Center: Pessimistic vs. Optimistic Concurrency Control. `http://ibm.co/1PCPlI3`.

[140] Michael Stonebraker. The Case for Shared Nothing. *IEEE Data Engineering Bulletin*, 9:4–9, 1986.

[141] Michael Stonebraker and Ugur Çetintemel. “One Size Fits All”: An Idea Whose Time Has Come and Gone. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 2–11, 2005.

[142] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era: (It’s Time for a Complete Rewrite). In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 1150–1160, 2007.

[143] Michael Stonebraker and Ariel Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin*, 36(2):21–27, 2013.

[144] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The Missing Memristor Found. *Nature*, 453(7191):80–83, 2008.

[145] Y. C. Tay, Nathan Goodman, and Rajan Suri. Locking performance in centralized databases. *ACM Transactions on Database Systems, TODS*, 10(4):415–462, 1985.

[146] Times-Ten Team. In-Memory Data Management for Consumer Transactions The Times-Ten Approach. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 528–529, 1999.

[147] Times-Ten Team. In-Memory Data Management in the Application Tier. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 637–641, 2000.

[148] Times-Ten Team. Mid-tier Caching: The TimesTen Approach. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 588–593, 2002.

[149] Robert H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems, TODS*, 4(2):180–209, June 1979.

[150] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the ACM Conference on Management of Data, SIGMOD*, pages 1–12, 2012.

[151] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the ACM Symposium on Operating Systems Principles, SOSP*, pages 18–32, 2013.

[152] Tianzheng Wang and Ryan Johnson. Scalable Logging through Emerging Non-Volatile Memory. *Proceedings of the VLDB Endowment, PVLDB*, 7(10):865–876, 2014.

[153] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. Using Restricted Transactional Memory to Build a Scalable In-Memory Database. In *Proceedings of the European Conference on Computer Systems, EuroSys*, pages 1–26, 2014.

[154] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-Memory Transaction Processing using RDMA and HTM. In *Proceedings of the ACM Symposium on Operating Systems Principles, SOSP*, pages 87–104, 2015.

[155] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

[156] Kyu-Young Whang and Ravi Krishnamurthy. Query Optimization in a Memory-resident Domain Relational Calculus Database System. *ACM Transactions on Database Systems, TODS*, 15(1):67–95, March 1990.

[157] Authur Whitney, Dennis Shasha, and Stevan Apter. High Volume Transaction Processing without Concurrency Control, Two Phase Commit, SQL or C++. In *Proceedings of the International Workshop on High Performance Transaction Systems, HPTS*, 1997.

[158] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. Vectorizing Database Column Scans with Complex Predicates. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, ADMS*, pages 1–12, 2013.

[159] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *Proceedings of the USENIX Symposium on Operation System Design and Implementation, OSDI*, pages 465–477, 2014.

[160] Daniel C. Zilio. *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems.* PhD thesis, University of Toronto, 1998.