In-Memory Big Data Management and Processing: A Survey

Hao Zhang, Gang Chen, *Member, IEEE*, Beng Chin Ooi, *Fellow, IEEE*, Kian-Lee Tan, *Member, IEEE*, and Meihui Zhang, *Member, IEEE*

Abstract—Growing main memory capacity has fueled the development of in-memory big data management and processing. By eliminating disk I/O bottleneck, it is now possible to support interactive data analytics. However, in-memory systems are much more sensitive to other sources of overhead that do not matter in traditional I/O-bounded disk-based systems. Some issues such as fault-tolerance and consistency are also more challenging to handle in in-memory environment. We are witnessing a revolution in the design of database systems that exploits main memory as its data storage layer. Many of these researches have focused along several dimensions: modern CPU and memory hierarchy utilization, time/space efficiency, parallelism, and concurrency control. In this survey, we aim to provide a thorough review of a wide range of in-memory data management and processing proposals and systems, including both data storage systems and data processing frameworks. We also give a comprehensive presentation of important technology in memory management, and some key factors that need to be considered in order to achieve efficient in-memory data management and processing.

Index Terms—Primary memory, DRAM, relational databases, distributed databases, query processing

1 Introduction

The explosion of Big Data has prompted much research to develop systems to support ultra-low latency service and real-time data analytics. Existing disk-based systems can no longer offer timely response due to the high access latency to hard disks. The unacceptable performance was initially encountered by Internet companies such as Amazon, Google, Facebook and Twitter, but is now also becoming an obstacle for other companies/organizations which desire to provide a meaningful real-time service (e.g., real-time bidding, advertising, social gaming). For instance, trading companies need to detect a sudden change in the trading prices and react instantly (in several milliseconds), which is impossible to achieve using traditional disk-based processing/storage systems. To meet the strict real-time requirements for analyzing mass amounts of data and servicing requests within milliseconds, an in-memory system/database that keeps the data in the random access memory (RAM) all the time is necessary.

Jim Gray's insight that "Memory is the new disk, disk is the new tape" is becoming true today [1]—we are witnessing a trend where memory will eventually replace disk and the role of disks must inevitably become more archival. In the last decade, multi-core processors and the availability of large amounts of main memory at plummeting cost are creating new breakthroughs, making it viable to build in-memory systems where a significant part, if not the entirety, of the database fits in memory. For example, memory storage capacity and bandwidth have been doubling roughly every three years, while its price has been dropping by a factor of 10 every five years. Similarly, there have been significant advances in non-volatile memory (NVM) such as SSD and the impending launch of various NVMs such as phase change memory (PCM). The number of I/O operations per second in such devices is far greater than hard disks. Modern high-end servers usually have multiple sockets, each of which can have tens or hundreds of gigabytes of DRAM, and tens of cores, and in total, a server may have several terabytes of DRAM and hundreds of cores. Moreover, in a distributed environment, it is possible to aggregate the memories from a large number of server nodes to the extent that the aggregated memory is able to keep all the data for a variety of large-scale applications (e.g., Facebook [2]).

Database systems have been evolving over the last few decades, mainly driven by advances in hardware, availability of a large amount of data, collection of data at an unprecedented rate, emerging applications and so on. The landscape of data management systems is increasingly fragmented based on application domains (i.e., applications relying on relational data, graph-based data, stream data). Fig. 1 shows state-of-the-art commercial and academic systems for disk-based and in-memory operations. In this survey, we focus on in-memory systems; readers are referred to [3] for a survey on disk-based systems.

In business operations, speed is not an option, but a must. Hence every avenue is exploited to further improve performance, including reducing dependency on the hard disk, adding more memory to make more data resident in

- *H. Zhang, B.C. Ooi, and K.-L. Tan are with the School of Computing, National University of Singapore, Singapore 117417. E-mail: {zhangh, ooibc, tankl}@comp.nus.edu.sg.*
- *G. Chen is with the College of Computer Science, Zhejiang University, Hangzhou 310027, China. E-mail: cg@cs.zju.edu.cn.*
- *M. Zhang is with the Information Systems Technology and Design Pillar, Singapore University of Technology and Design, Singapore 487372. E-mail: meihui_zhang@sutd.edu.sg.*

Manuscript received 16 Jan. 2015; revised 22 Apr. 2015; accepted 25 Apr. 2015. Date of publication 28 Apr. 2015; date of current version 1 June 2015. Recommended for acceptance by J. Pei.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2015.2427795

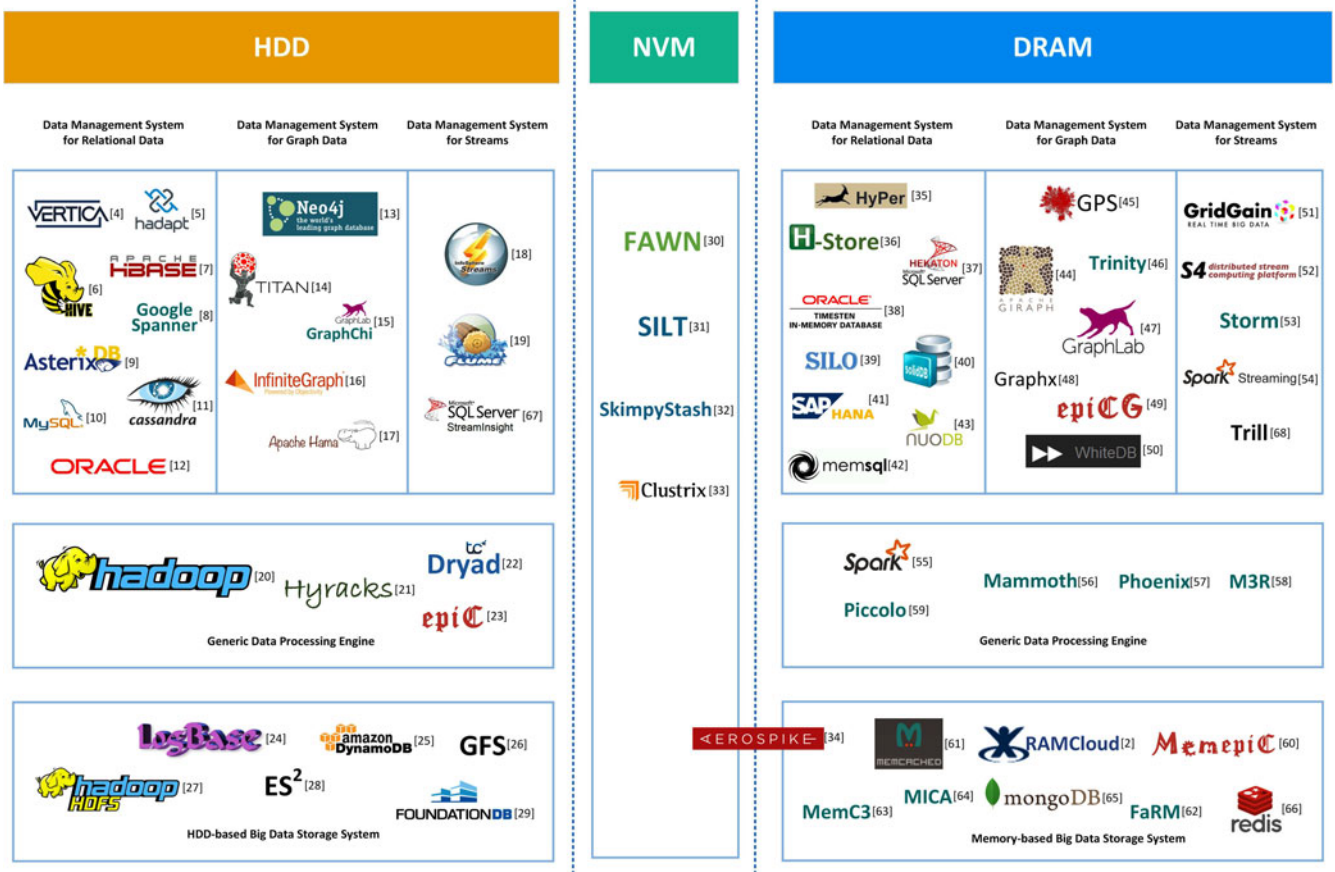

Fig. 1. The (Partial) landscape of disk-based and in-memory data management systems.

the memory, and even deploying an in-memory system where all data can be kept in memory. In-memory database systems have been studied in the past, as early as the 1980s [69], [70], [71], [72], [73]. However, recent advances in hardware technology have invalidated many of the earlier works and re-generated interests in hosting the whole database in memory in order to provide faster accesses and real-time analytics [35], [36], [55], [74], [75], [76]. Most commercial database vendors have recently introduced in-memory database processing to support large-scale applications completely in memory [37], [38], [40], [77]. Efficient in-memory data management is a necessity for various applications [78], [79]. Nevertheless, in-memory data management is still at its infancy, and is likely to evolve over the next few years.

In general, as summarized in Table 1, research in in-memory data management and processing focus on the following aspects for efficiency or enforcing ACID properties:

- Indexes. Although in-memory data access is extremely fast compared to disk access, an efficient index is still required for supporting point queries in order to avoid memory-intensive scan. Indexes designed for in-memory databases are quite different from traditional indexes designed for disk-based databases such as the B^+-tree, because traditional indexes mainly care about the I/O efficiency instead of memory and cache utilization. Hash-based indexes are commonly used in key-value stores, e.g., Memcached [61], Redis [66], RAMCloud [75], and can be further optimized for better cache utilization by reducing pointer chasing [63]. However hash-based indexes do not support range queries, which are crucial for data analytics and thus, tree-based indexes have also been proposed, such as T-Tree [80], Cache Sensitive Search Trees (CSS-Trees) [81], Cache Sensitive B^+-Trees (CSB^+-Trees) [82], Δ-Tree [83], BD-Tree [84], Fast Architecture Sensitive Tree (FAST) [85], Bw-tree [86] and Adaptive Radix Tree (ART) [87], some of which also consider pointer reduction.
- Data layouts. In-memory data layouts have a significant impact on the memory usage and cache utilization. Columnar layout of relational table facilitates scan-like queries/analytics as it can achieve good cache locality [41], [88], and can achieve better data compression [89], but is not optimal for OLTP queries that need to operate on the row level [74], [90]. It is also possible to have a hybrid of row and column layouts, such as PAX which organizes data by columns only within a page [91], and SAP HANA with multi-layer stores consisting of several delta row/column stores and a main column store, which are merged periodically [74]. In addition, there are also proposals on handling the memory fragmentation problem, such as the slab-based allocator in Memcached [61], and log-structured data organization with periodical cleaning in RAMCloud [75], and better utilization of some hardware features (e.g., bit-level parallelism, SIMD), such as BitWeaving [92] and ByteSlice [93].
- Parallelism. In general, there are three levels of parallelism, i.e., data-level parallelism (e.g., bit-level

TABLE 1
Optimization Aspects on In-Memory Data Management and Processing

Aspects	Concerns	Related Work
Index	cache consciousness, time/space efficiency	T-Tree [80], CSS-Trees [81], CSB$^+$-Trees [82], Δ-Tree [83], BD-Tree [84], FAST [85], ART [87]
Data Layout	cache consciousness, space efficiency	PAX [91], columnar layout [41], [88], HANA Hybrid Store [74], slab allocator [61], log-structure [75]
Parallelism	linear scaling, partitioning	BitWeaving [92], bit-parallel aggregation [94], SIMD sorting [95], SIMD scanning [96], [97], multi-core join [98], distributed computing [2], [55], [99], [100]
Concurrency Control/ Transaction Management	overhead, correctness	virtual snapshot [35], lock-eliding [101], transactional memory [102], [103], PALM [104], LIL [105], VLL [106], OCC [39], [107], MVCC [108], [109], DGCC [110]
Query Processing	code locality, register temporal locality, time efficiency	stored procedure [111], JIT compilation [112], [113], join [98], [114], [115], [116], [117], [118], [119], [120], [121], [122], sort [95], [123], [124]
Fault Tolerance	durability, correlated failures, availability	Copyset [125], fast recovery [126], group commit and log coalescing [37], [127], NVM [128], [129], [130], command logging [131], adaptive logging [132], remote logging [2], [40]
Data Overflow	locality, paging strategy, hot/cold classification	Anti-caching [133], Hekaton Siberia [134], data compression [74], [89], [135], virtual memory management [136], pointer swizzling [137], UVMM [138]

parallelism, SIMD),[1] shared-memory scale-up parallelism (thread/process),[2] and shared-nothing scale-out parallelism (distributed computation). All three levels of parallelism can be exploited at the same time, as shown in Fig. 2. The bit-parallel algorithms fully unleash the intra-cycle parallelism of modern CPUs, by packing multiple data values into one CPU word, which can be processed in one single cycle [92], [94]. Intra-cycle parallelism performance can be proportional to the packing ratio, since it does not require any concurrency control (CC) protocol. SIMD instructions can improve vector-style computations greatly, which are extensively used in high-performance computing, and also in the database systems [95], [96], [97]. Scale-up parallelism can take advantage of the multi-core architecture of super-computers or even commodity computers [36], [98], while scale-out parallelism is highly utilized in cloud/distributed computing [2], [55], [99]. Both scale-up and scale-out parallelisms require a good data partitioning strategy in order to achieve load balancing and minimize cross-partition coordination [100], [139], [140], [141].

- Concurrency control/transaction management. Concurrency control/transaction management becomes an extremely important performance issue in in-memory data management with the many-core systems. Heavy-weight mechanisms based on lock/semaphore greatly degrade the performance, due to its blocking-style scheme and the overhead caused by centralized lock manager and deadlock detection [142], [143]. Lightweight Intent Lock (LIL) [105] was proposed to maintain a set of lightweight counters in a global lock table instead of lock queues for intent locks. Very Lightweight Locking (VLL) [106] further simplifies the data structure by compressing all the lock states of one record into a pair of integers for partitioned databases. Another class of concurrency control is based on timestamp, where a predefined order is used to guarantee transactions' serializability [144], such as optimistic concurrency control (OCC) [39], [107] and multi-version concurrency control (MVCC) [108], [109]. Furthermore, H-Store [36], [101], seeks to eliminate concurrency control in single-partition transactions by partitioning the database beforehand based on a priori workload and providing one thread for each partition. HyPer [35] isolates OLTP and OLAP by *fork*-ing a child process (via *fork()* system call) for OLAP jobs based on the hardware-assisted virtual snapshot, which will never be modified. DGCC [110] is proposed to reduce the overhead of concurrency control by separating concurrency control from execution based on a dependency graph. Hekaton [104], [107] utilizes optimistic MVCC and lock-free data structures to achieve high concurrency efficiently. Besides, hardware transactional memory (HTM) [102], [103] is being increasingly exploited in concurrency control for OLTP.
- Query processing. Query processing is going through an evolution in in-memory databases. While the traditional Iterator-/Volcano-style model [145] facilitates easy combination of arbitrary operators, it generates a huge number of function calls (e.g., *next()*) which results in evicting the register contents. The poor code locality and frequent instruction miss-predictions further add to the overhead [112], [113]. Coarse-grained stored procedures (e.g., transaction-level) can be used to alleviate the problem [111], and dynamic compiling (Just-in-Time) is another approach to achieve better code and data locality [112], [113]. Performance gain can also be achieved by optimizing specific query operation

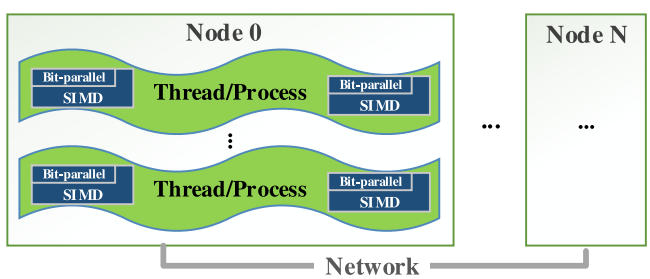

Fig. 2. Three levels of parallelism.

1. Here data-level parallelism includes both bit-level parallelism achieved by data packing, and word-level parallelism achieved by SIMD.

2. Accelerators such as GPGPU and Xeon Phi are also considered as shared-memory scale-up parallelism.

such as join [98], [114], [115], [116], [117], [118], [119], [120], [121], [122], and sort [95], [123], [124].

- Fault tolerance. DRAM is volatile, and fault-tolerance mechanisms are thus crucial to guarantee the data durability to avoid data loss and to ensure transactional consistency when there is a failure (e.g., power, software or hardware failure). Traditional write-ahead logging (WAL) is also the de facto approach used in in-memory database systems [35], [36], [37]. But the data volatility of the in-memory storage makes it unnecessary to apply any persistent undo logging [37], [131] or completely disables it in some scenarios [111]. To eliminate the potential I/O bottleneck caused by logging, group commit and log coalescing [37], [127], and remote logging [2], [40] are adopted to optimize the logging efficiency. New hardware technologies such as SSD and PCM are utilized to increase the I/O performance [128], [129], [130]. Recent studies proposed to use *command logging* [131], which logs only operations instead of the updated data, which is used in traditional ARIES logging [146]. [132] studies how to alternate between these two strategies adaptively. To speed up the recovery process, a consistent snapshot has to be checkpointed periodically [37], [147], and replicas should be dispersed in anticipation of correlated failures [125]. High availability is usually achieved by maintaining multiple replicas and stand-by servers [37], [148], [149], [150], or relying on fast recovery upon failure [49], [126]. Data can be further backuped onto a more stable storage such as GPFS [151], HDFS [27] and NAS [152] to further secure the data.
- Data overflow. In spite of significant increase in memory size and sharp drop in its price, it still cannot keep pace with the rapid growth of data in the Big Data era, which makes it essential to deal with data overflow where the size of the data exceeds the size of main memory. With the advancement of hardware, hybrid systems which incorporate non-volatile memories (NVMs) (e.g., SCM, PCM, SSD, Flash memory) [30], [31], [32], [118], [127], [153], [154], [155], [156] become a natural solution for achieving the speed. Alternatively, as in the traditional database systems, effective eviction mechanisms could be adopted to replace the in-memory data when the main memory is not sufficient. The authors of [133], [134], [157] propose to move cold data to disks, and [136] re-organizes the data in memory and relies on OS to do the paging, while [137] introduces pointer swizzling in database buffer pool management to alleviate the overhead caused by traditional databases in order to compete with the completely re-designed in-memory databases. UVMM [138] taps onto a hybrid of hardware-assisted and semantics-aware access tracking, and non-blocking kernel I/O scheduler, to facilitate efficient memory management. Data compression has also been used to alleviate the memory usage pressure [74], [89], [135].

The focus of the survey is on large-scale in-memory data management and processing strategies, which can be broadly grouped into two categories, i.e., in-memory data storage systems and in-memory data processing systems. Accordingly, the remaining sections are organized as follows. Section 2 presents some background on in-memory data management. We elaborate in-memory data storage systems, including relational databases and NoSQL databases in Section 3, and in-memory data processing systems, including in-memory batch processing and real-time stream processing in Section 4. As a summary, we present a qualitative comparison of the in-memory data management systems covered in this survey in Section 5. Finally, we discuss some research opportunities in Section 6, and conclude in Section 7.

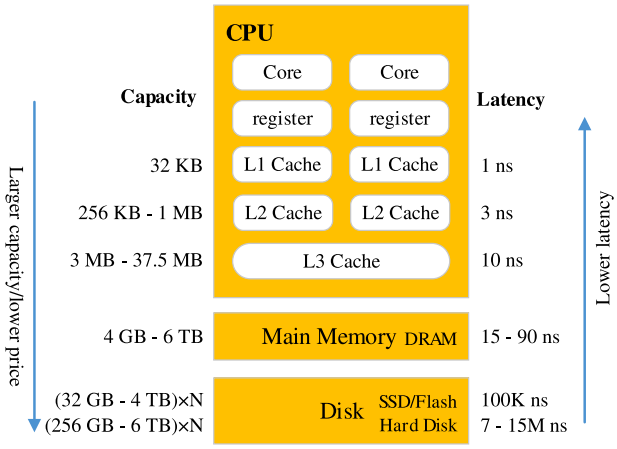

Fig. 3. Memory hierarchy.

2 Core Technologies for In-Memory Systems

In this section, we shall introduce some concepts and techniques that are important for efficient in-memory data management, including memory hierarchy, non-uniform memory access (NUMA), transactional memory, and non-volatile random access memory (NVRAM). These are the basics on which the performance of in-memory data management systems heavily rely.

2.1 Memory Hierarchy

The memory hierarchy is defined in terms of access latency and the logical distance to the CPU. In general, it consists of registers, caches (typically containing L1 cache, L2 cache and L3 cache), main memory (i.e., RAM) and disks (e.g., hard disk, flash memory, SSD) from the highest performance to the lowest. Fig. 3 depicts the memory hierarchy, and respective component's capacity and access latency [158], [159], [160], [161], [162], [163], [164], [165], [166]. It shows that data access to the higher layers is much faster than to the lower layers, and each of these layers will be introduced in this section.

In modern architectures, data cannot be processed by CPU unless it is put in the registers. Thus, data that is about to be processed has to be transmitted through each of the memory layers until it reaches the registers. Consequently, each upper layer serves as a cache for the underlying lower layer to reduce the latency for repetitive data accesses. The performance of a data-intensive program highly depends on the

utilization of the memory hierarchy. How to achieve both good spatial and temporal locality is usually what matters the most in the efficiency optimization. In particular, spatial locality assumes that the adjacent data is more likely to be accessed together, whereas temporal locality refers to the observation that it is likely that an item will be accessed again in the near future. We will introduce some important efficiency-related properties of different memory layers respectively.

2.1.1 Register

A processor register is a small amount of storage within a CPU, on which machine instructions can manipulate directly. In a normal instruction, data is first loaded from the lower memory layers into registers where it is used for arithmetic or test operation, and the result is put back into another register, which is then often stored back into main memory, either by the same instruction or a subsequent one. The length of a register is usually equal to the word length of a CPU, but there also exist double-word, and even wider registers (e.g., 256 bits wide YMMX registers in Intel Sandy Bridge CPU micro architecture), which can be used for single instruction multiple data (SIMD) operations. While the number of registers depends on the architecture, the total capacity of registers is much smaller than that of the lower layers such as cache or memory. However, accessing data from registers is very much faster.

2.1.2 Cache

Registers play the role as the storage containers that CPU uses to carry out instructions, while caches act as the bridge between the registers and main memory due to the high transmission delay between the registers and main memory. Cache is made of high-speed static RAM (SRAM) instead of slower and cheaper dynamic RAM (DRAM) that usually forms the main memory. In general, there are three levels of caches, i.e., L1 cache, L2 cache and L3 cache (also called last level cache—LLC), with increasing latency and capacity. L1 cache is further divided into data cache (i.e., L1-dcache) and instruction cache (i.e., L1-icache) to avoid any interference between data access and instruction access. We call it a cache hit if the requested data is in the cache; otherwise it is called a cache miss.

Cache is typically subdivided into fixed-size logical cache lines, which are the atomic units for transmitting data between different levels of caches and between the last level cache and main memory. In modern architectures, a cache line is usually 64 bytes long. By filling the caches per cache line, spatial locality can be exploited to improve performance. The mapping between the main memory and the cache is determined by several strategies, i.e., direct mapping, N-way set associative, and fully associative. With direct mapping, each entry (a cache line) in the memory can only be put in one place in the cache, which makes addressing faster. Under fully associative strategy, each entry can be put in any place, which offers fewer cache misses. The N-way associative strategy is a compromise between direct mapping and fully associative—it allows each entry in the memory to be in any of N places in the cache, which is called a cache set. N-way associative is often used in practice, and the mapping is deterministic in terms of cache sets.

In addition, most architectures usually adopt a least-recently-used (LRU) replacement strategy to evict a cache line when there is not enough room. Such a scheme essentially utilizes temporal locality for enhancing performance. As shown in Fig. 3, the latency to access cache is much shorter than the latency to access main memory. In order to gain good CPU performance, we have to guarantee high cache hit rate so that high-latency memory accesses are reduced. In designing an in-memory management system, it is important to exploit the properties of spatial and temporal locality of caches. For examples, it would be faster to access memory sequentially than randomly, and it would also be better to keep a frequently-accessed object resident in the cache. The advantage of sequential memory access is reinforced by the prefetching strategies of modern CPUs.

2.1.3 Main Memory and Disks

Main memory is also called internal memory, which can be directly addressed and possibly accessed by the CPU, in contrast to external devices such as disks. Main memory is usually made of volatile DRAM, which incurs equivalent latency for random accesses without the effect of caches, but will lose data when power is turned off. Recently, DRAM becomes inexpensive and large enough to make an in-memory database viable.

Even though memory becomes the new disk [1], the volatility of DRAM makes it a common case that disks[3] are still needed to backup data. Data transmission between main memory and disks is conducted in units of pages, which makes use of data spatial locality on the one hand and minimizes the performance degradation caused by the high-latency of disk seek on the other hand. A page is usually a multiple of disk sectors[4] which is the minimum transmission unit for hard disk. In modern architectures, OS usually keeps a buffer which is part of the main memory to make the communication between the memory and disk faster.[5] The buffer is mainly used to bridge the performance gap between the CPU and the disk. It increases the disk I/O performance by buffering the writes to eliminate the costly disk seek time for every write operation, and buffering the reads for fast answer to subsequent reads to the same data. In a sense, the buffer is to the disk as the cache is to the memory. And it also exposes both spatial and temporal locality, which is an important factor in handling the disk I/O efficiently.

2.2 Memory Hierarchy Utilization

This section reviews related works from three perspective—register-conscious optimization, cache-conscious optimization and disk I/O optimization.

2.2.1 Register-Conscious Optimization

Register-related optimization usually matters in compiler and assembly language programming, which requires

3. Here we refer to hard disks.

4. A page in the modern file system is usually 4 KB. Each disk sector of hard disks is traditionally 512 bytes.

5. The kernel buffer is also used to buffer data from other block I/O devices that transmit data in fixed-size blocks.

utilizing the limited number of registers efficiently. There have been some criticisms on the traditional iterator-style query processing mechanisms for in-memory databases recently as it usually results in poor code and data locality [36], [112], [167]. HyPer uses low level virtual machine (LLVM) compiler framework [167] to translate a query into machine code dynamically, which achieves good code and data locality by avoiding recursive function calls as much as possible and trying to keep the data in the registers as long as possible [112].

SIMD is available in superscalar processors, which exploits data-level parallelism with the help of wide registers (e.g., 256 bits). SIMD can improve the performance significantly especially for vector-style computation, which is very common in Big Data analytics jobs [95], [96], [97], [112], [168].

2.2.2 Cache-Conscious Optimization

Cache utilization is becoming increasingly important in modern architectures. Several workload characterization studies provide detailed analysis of the time breakdown in the execution of DBMSs on a modern processor, and report that DBMSs suffer from high memory-related processor stalls when running on modern architectures. This is caused by a huge amount of data cache misses [169], which account for 50-70 percent for OLTP workloads [91], [170] to 90 percent for DSS workloads [91], [171], of the total memory-related stall. In a distributed database, instruction cache misses are another main source of performance degradation due to a large number of TCP network I/Os [172].

To utilize the cache more efficiently, some works focus on re-organizing the data layout by grouping together all values of each attribute in an N-ary Storage Model (NSM) page [91] or using a Decomposition Storage Model (DSM) [173] or completely organizing the records in a column store [41], [90], [174], [175]. This kind of optimization favors OLAP workload which typically only needs a few columns, but has a negative impact on intra-tuple cache locality [176]. There are also other techniques to optimize cache utilization for the primary data structure, such as compression [177] and coloring [178].

In addition, for memory-resident data structures, various cache-conscious indexes have been proposed such as Cache Sensitive Search Trees [81], Cache Sensitive B^+-Trees [82], Fast Architecture Sensitive Trees [85], and Adaptive Radix Trees [87]. Cache-conscious algorithms have also been proposed for basic operations such as sorting (e.g., burst sort) [179] and joining [98], [114].

In summary, to optimize cache utilization, the following important factors should be taken into consideration:

- Cache line length. This characteristic exposes spatial locality, meaning that it would be more efficient to access adjacent data.
- Cache size. It would also be more efficient to keep frequently-used data within at least L3 cache size.
- Cache replacement policy. One of the most popular replacement policies is LRU, which replaces the least recently used cache line upon a cache miss. The temporal locality should be exploited in order to get high performance.

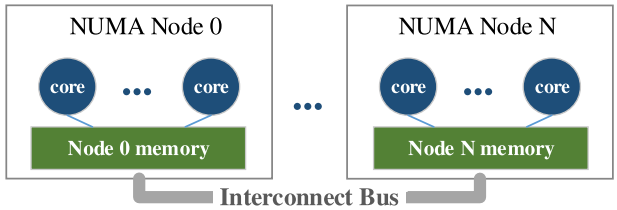

Fig. 4. NUMA topology.

2.3 Non-Uniform Memory Access

Non-uniform memory access is an architecture of the main memory subsystem where the latency of a memory operation depends on the relative location of the processor that is performing memory operations. Broadly, each processor in a NUMA system has a *local memory* that can be accessed with minimal latency, but can also access at least one *remote memory* with longer latency, which is illustrated in Fig. 4.

The main reason for employing NUMA architecture is to improve the main memory bandwidth and total memory size that can be deployed in a server node. NUMA allows the clustering of several memory controllers into a single server node, creating several *memory domains*. Although NUMA systems were deployed as early as 1980s in specialized systems [185], since 2008 all Intel and AMD processors incorporate one memory controller. Thus, most contemporary multi-processor systems are NUMA; therefore, NUMA-awareness is becoming a mainstream challenge.

In the context of data management systems, current research directions on NUMA-awareness can be broadly classified into three categories:

- partitioning the data such that memory accesses to remote NUMA domains are minimized [115], [186], [187], [188], [189];
- managing NUMA effects on latency-sensitive workloads such as OLTP transactions [190], [191];
- efficient data shuffling across NUMA domains [192].

2.3.1 Data Partitioning

Partitioning the working set of a database has long been used to minimize data transfers across different data domains, both within a compute node and across compute nodes. Bubba [186] is an example of an earlier parallel database system that uses a shared-nothing architecture to scale to hundreds of compute nodes. It partitions the data using a hash- or range-based approach and always performs the analytics operations only in the nodes that contain the relevant partitions. Gamma [187] is another example that was designed to operate on a complex architecture with an Intel iPSC/2 hypercube with 32 processors and 32 disk drives. Like Bubba, Gamma partitions the data across multiple disk drives and uses a hash-based approach to implement join and aggregate operations on top of the partitioned data. The partitioned data and execution provide the partitioned parallelism [193]. With NUMA systems becoming mainstream, many research efforts have started to address NUMA issues explicitly, rather than just relying on data partitioning. Furthermore, modern systems

TABLE 2
Comparison between STM and HTM

	Performance Penalty	Hardware Support	Transaction Size	Implementations
STM	Much	Atomic Operation	Large	TinySTM [180], Clojure [181], Haskell [182]
HTM	No or Little	Cache, Bus Protocol	Small	Intel TSX [183], AMD ASF [184]

have increasingly larger number of cores. The recent change in memory topology and processing power have indeed attracted interest in re-examining traditional processing methods in the context of NUMA.

A new sort-merge technique for partitioning the join operation was proposed in [115] to take advantage of NUMA systems with high memory bandwidth and many cores. In contrast to hash join and classical sort-merge join, the parallel sort-merge strategy parallelizes also the final merge step, and naturally operates on local memory partitions. This is to take advantage of both the multi-core architecture and the large local memory bandwidth that most NUMA systems have.

Partitioning the database index was proposed for the Buzzard system [188]. Index operations typically incur frequent pointer chasing during the traversal of a tree-based index. In a NUMA system, these pointer operations might end up swinging from one memory domain to another. To address this problem, Buzzard proposes a NUMA-aware index that partitions different parts of a prefix tree-based index across different NUMA domains. Furthermore, Buzzard uses a set of dedicated worker threads to access each memory domain. This guarantees that threads only access their local memory during index traversal and further improves the performance by using only local comparison and swapping operations instead of expensive locking.

Partitioning both the input data and query execution was proposed in [189]. In contrast to plan-driven query execution, a fine-grained runtime task scheduling, termed "morsel query execution" was proposed. The morsel-driven query processing strategy dynamically assembles small pieces of input data, and executes them in a pipelined fashion by assigning them to a pool of worker threads. Due to this fine-grained control over the parallelism and the input data, morsel-driven execution is aware of the data locality of each operator, and can schedule their execution on local memory domains.

2.3.2 OLTP Latency

Since NUMA systems have heterogeneous access latency, they pose a challenging problem to OLTP transactions which are typically very sensitive to latency. The performance of NUMA-unaware OLTP deployments on NUMA systems is profiled in [190], where many of these systems are deemed to have achieved suboptimal and unpredictable performance. To address the needs for a NUMA-aware OLTP system, the paper proposes "hardware islands", in which it treats each memory domain as a logical node, and uses UNIX domain sockets to communicate among the NUMA memory domains of a physical node. The recently proposed ATraPos [191] is an adaptive transaction processing system that has been built based on this principle.

2.3.3 Data Shuffling

Data shuffling in NUMA systems aims to transfer the data across the NUMA domains as efficiently as possible, by saturating the transfer bandwidth of the NUMA interconnect network. A NUMA-aware coordinated ring-shuffling method was proposed in [192]. To shuffle the data across NUMA domains as efficiently as possible, the proposed approach forces the threads across NUMA domains to communicate in a coordinated manner. It divides the threads into an inner ring and an outer ring and performs communication in a series of rounds, during which the inner ring remains fixed while the outer ring rotates. This rotation guarantees that all threads across the memory domains will be paired based on a predictable pattern, and thus all the memory channels of the NUMA interconnect are always busy. Compared to the naive method of shuffling the data around the domains, this method improves the transfer bandwidth by a factor of four, when using a NUMA system with four domains.

2.4 Transactional Memory

Transactional memory is a concurrency control mechanism for shared memory access, which is analogous to atomic database transactions. The two types of transactional memory, i.e., software transactional memory (STM) and hardware transactional memory (HTM), are compared in Table 2. STM causes a significant slowdown during execution and thus has limited practical application [194], while HTM has attracted new attention for its efficient hardware-assisted atomic operations/transactions, since Intel introduced it in its mainstream Haswell microarchitecture CPU [102], [103]. Haswell HTM is implemented based on cache coherency protocol. In particular, L1 cache is used as a local buffer to mark all transactional read/write on the granularity of cache lines. The propagation of changes to other caches or main memory is postponed until the transaction commits, and write/write and read/write conflicts are detected using the cache coherency protocol [195]. This HTM design incurs almost no overhead for transaction execution, but has the following drawbacks, which make HTM only suitable for small and short transactions.

- The transaction size is limited to the size of L1 data cache, which is usually 32 KB. Thus it is not possible to simply execute a database transaction as one monolithic HTM transaction.
- Cache associativity makes it more prone to false conflicts, because some cache lines are likely to go to the same cache set, and an eviction of a cache line leads to abort of the transaction, which cannot be resolved by restarting the transaction due to the determinism of the cache mapping strategy (refer to Section 2.1.2).
- HTM transactions may be aborted due to interrupt events, which limits the maximum duration of HTM transactions.

There are two instruction sets for Haswell HTM in Transactional Synchronization Extensions (TSX),[6] i.e., Hardware Lock Ellison (HLE) and Restricted Transactional Memory (RTM). HLE allows optimistic execution of a transaction by eliding the lock so that the lock is free to other threads, and restarting it if the transaction failed due to data race, which mostly incurs no locking overhead, and also provides backward compatibility with processors without TSX. RTM is a new instruction set that provides the flexibility to specify a fallback code path after a transaction aborts. The author [102] exploits HTM based on HLE, by dividing a database transaction into a set of relatively small HTM transactions with timestamp ordering (TSO) concurrency control and minimizing the false abort probability via data/index segmentation. RTM is utilized in [103], which uses a three-phase optimistic concurrency control to coordinate a whole database transaction, and protects single data read (to guarantee consistency of sequence numbers) and validate/write phases using RTM transactions.

2.5 NVRAM

Newly-emerging non-volatile memory raises the prospect of persistent high-speed memory with large capacity. Examples of NVM include both NAND/NOR flash memory with block-granularity addressability, and non-volatile random access memory with byte-granularity addressability.[7] Flash memory/SSD has been widely used in practice, and attracted a significant amount of attention in both academia and industry [32], [33], but its block-granularity interface, and expensive "erase" operation make it only suitable to act as the lower-level storage, such as replacement of hard disk [30], [32], or disk cache [198]. Thus, in this survey, we only focus on NVRAMs that have byte addressability and comparable performance with DRAM, and can be brought to the main memory layer or even the CPU cache layer.

Advanced NVRAM technologies, such as phase change memory [199], Spin-Transfer Torque Magnetic RAM (STT-MRAM) [200], and Memristors [201], can provide orders of magnitude better performance than either conventional hard disk or flash memory, and deliver excellent performance on the same order of magnitude as DRAM, but with persistent writes [202]. The read latency of PCM is only two-five times slower than DRAM, and STT-MRAM and Memristor could even achieve lower access latency than DRAM [118], [128], [129], [203]. With proper caching, carefully architected PCM could also match DRAM performance [159]. Besides, NVRAM is speculated to have much higher storage density than DRAM, and consume much less power [204]. Although NVRAM is currently only available in small sizes, and the cost per bit is much higher than that of hard disk or flash memory or even DRAM, it is estimated that by the next decade, we may have a single PCM with 1 TB and Memristor with 100 TB, at price close to the enterprise hard disk [128], [129]. The advent of NVRAM offers an intriguing opportunity to revolutionize the data management and re-think the system design.

It has been shown that simply replacing disk with NVRAM is not optimal, due to the high overhead from the cumbersome file system interface (e.g., file system cache and costly system calls), block-granularity access and high economic cost, etc. [127], [130], [205]. Instead, NVRAM has been proposed to be placed side-by-side with DRAM on the memory bus, available to ordinary CPU `loads` and `stores`, such that the physical address space can be divided between volatile and non-volatile memory [205], or be constituted completely by non-volatile memory [155], [206], [207], equipped with fine-tuned OS support [208], [209]. Compared to DRAM, NVRAM exhibits its distinct characteristics, such as limited endurance, write/read asymmetry, uncertainty of ordering and atomicity [128], [205]. For example, the write latency of PCM is more than one order of magnitude slower than its read latency [118]. Besides, there is no standard mechanisms/protocols to guarantee the ordering and atomicity of NVRAM writes [128], [205]. The endurance problem can be solved by wear-leveling techniques in the hardware or middleware levels [206], [207], [210], which can be easily hidden from the software design, while the read/write asymmetry, and ordering and atomicity of writes, must be taken into consideration in the system/algorithm design [204].

Promisingly, NVRAM can be architected as the main memory in general-purpose systems with well-designed architecture [155], [206], [207]. In particular, longer write latency of PCM can be solved by data comparison writes [211], partial writes [207], or specialized algorithms/structures that trade writes for reads [118], [204], [212], which can also alleviate the endurance problem. And current solutions to the write ordering and atomicity problems are either relying on some newly-proposed hardware primitives, such as atomic 8-byte writes and epoch barriers [129], [205], [212], or leveraging existing hardware primitives, such as cache modes (e.g., write-back, write-combining), memory barriers (e.g., `mfence`), cache line flush (e.g., `clflush`) [128], [130], [213], which, however, may incur non-trivial overhead. General libraries and programming interfaces are proposed to expose NVRAM as a persistent heap, enabling NVRAM adoption in an easy-to-use manner, such as NV-heaps [214], Mnemosyne [215], NVMalloc [216], and recovery and durable structures [213], [217]. In addition, file system support enables a transparent utilization of NVRAM as a persistent storage, such as Intel's PMFS [218], BPFS [205], FRASH [219], ConquestFS [220], SCMFS [221], which also take advantage of NVRAM's byte addressability.

Besides, specific data structures widely used in databases, such as B-Tree [212], [217], and some common query processing operators, such as sort and join [118], are starting to adapt to and take advantage of NVRAM properties. Actually, the favorite goodies brought to databases by NVRAM is its non-volatility property, which facilitates a more efficient logging and fault tolerance mechanisms [127], [128], [129], [130]. But write atomicity and deterministic orderings should be guaranteed and achieved efficiently via carefully designed algorithms, such as group

6. Intel disabled its TSX feature on Haswell, Haswell-E, Haswell-EP and early Broadwell CPUs in August 2014 due to a bug. Currently Intel only provides TSX on Intel Core M CPU with Broadwell architecture, and the newly-released Xeon E7 v3 CPU with Haswell-EX architecture [196], [197].

7. NVM and NVRAM usually can be used exchangeably without much distinction. NVRAM is also referred to as Storage-Class Memory (SCM), Persistent Memory (PM) or Non-Volatile Byte-addressable Memory (NVBM).

commit [127], passive group commit [128], two-step logging (i.e., populating the log entry in DRAM first and then flushing it to NVRAM) [130]. Also the centralized logging bottleneck should be eliminated, e.g., via distributed logging [128], decentralized logging [130]. Otherwise the high performance brought by NVRAM would be degraded by the legacy software overhead (e.g., centention for the centralized log).

3 In-Memory Data Storage Systems

In this section, we introduce some in-memory databases, including both relational and NoSQL databases. We also cover a special category of in-memory storage system, i.e., cache system, which is used as a cache between the application server and the underlying database. In most relational databases, both OLTP and OLAP workloads are supported inherently. The lack of data analytics operations in NoSQL databases results in an inevitable data transmission cost for data analytics jobs [172].

3.1 In-Memory Relational Databases

Relational databases have been developed and enhanced since 1970s, and the relational model has been dominating in almost all large-scale data processing applications since early 1990s. Some widely used relational databases include Oracle, IBM DB2, MySQL and PostgreSQL. In relational databases, data is organized into tables/relations, and ACID properties are guaranteed. More recently, a new type of relational databases, called NewSQL (e.g., Google Spanner [8], H-Store [36]) has emerged. These systems seek to provide the same scalability as NoSQL databases for OLTP while still maintaining the ACID guarantees of traditional relational database systems.

In this section, we focus on in-memory relational databases, which have been studied since 1980s [73]. However, there has been a surge in interests in recent years [222]. Examples of commercial in-memory relational databases include SAP HANA [77], VoltDB [150], Oracle TimesTen [38], SolidDB [40], IBM DB2 with BLU Acceleration [223], [224], Microsoft Hekaton [37], NuoDB [43], eXtremeDB [225], Pivotal SQLFire [226], and MemSQL [42]. There are also well known research/open-source projects such as H-Store [36], HyPer [35], Silo [39], Crescando [227], HYRISE [176], and MySQL Cluster NDB [228].

3.1.1 H-Store / VoltDB

H-Store [36], [229] or its commercial version VoltDB [150] is a distributed row-based in-memory relational database targeted for high-performance OLTP processing. It is motivated by two observations: first, certain operations in traditional disk-based databases, such as logging, latching, locking, B-tree and buffer management operations, incur substantial amount of the processing time (more than 90 percent) [222] when ported to in-memory databases; second, it is possible to re-design in-memory database processing so that these components become unnecessary. In H-Store, most of these "heavy" components are removed or optimized, in order to achieve high-performance transaction processing.

Transaction execution in H-Store is based on the assumption that all (at least most of) the templates of transactions are known in advance, which are represented as a set of compiled stored procedures inside the database. This reduces the overhead of transaction parsing at runtime, and also enables pre-optimizations on the database design and light-weight logging strategy [131]. In particular, the database can be more easily partitioned to avoid multi-partition transactions [140], and each partition is maintained by a *site*, which is single-threaded daemon that processes transactions serially and independently without the need for heavy-weight concurrency control (e.g., lock) in most cases [101]. Next, we will elaborate on its transaction processing, data overflow and fault-tolerance strategies.

Transaction processing. Transaction processing in H-Store is conducted on the partition/site basis. A site is an independent transaction processing unit that executes transactions sequentially, which makes it feasible only if a majority of the transactions are single-sited. This is because if a transaction involves multiple partitions, all these sites are sequentialized to process this distributed transaction in collaboration (usually 2PC), and thus cannot process transactions independently in parallel. H-Store designs a skew-aware partitioning model—*Horticulture* [140]—to automatically partition the database based on the database schema, stored procedures and a sample transaction workload, in order to minimize the number of multi-partition transactions and meanwhile mitigate the effects of temporal skew in the workload. *Horticulture* employs the large-neighborhood search (LNS) approach to explore potential partitions in a guided manner, in which it also considers read-only table replication to reduce the transmission cost of frequent remote access, secondary index replication to avoid broadcasting, and stored procedure routing attributes to allow an efficient routing mechanism for requests.

The *Horticulture* partitioning model can reduce the number of multi-partition transactions substantially, but not entirely. The concurrency control scheme must therefore be able to differentiate single partition transactions from multi-partition transactions, such that it does not incur high overhead where it is not needed (i.e., when there are only single-partition transactions). H-Store designs two low overhead concurrency control schemes, i.e., light-weight locking and speculative concurrency control [101]. Light-weight locking scheme reduces the overhead of acquiring locks and detecting deadlock by allowing single-partition transactions to execute without locks when there are no active multi-partition transactions. And speculative concurrency control scheme can proceed to execute queued transactions speculatively while waiting for 2PC to finish (precisely after the last fragment of a multi-partition transaction has been executed), which outperforms the locking scheme as long as there are few aborts or few multi-partition transactions that involve multiple rounds of communication.

In addition, based on the partitioning and concurrency control strategies, H-Store utilizes a set of optimizations on transaction processing, especially for workload with interleaving of single- and multi-transactions. In particular, to process an incoming transaction (a stored procedure with concrete parameter values), H-Store uses a Markov model-

based approach [111] to determine the necessary optimizations by predicting the most possible execution path and the set of partitions that it may access. Based on these predictions, it applies four major optimizations accordingly, namely (1) execute the transaction at the node with the partition that it will access the most; (2) lock only the partitions that the transaction accesses; (3) disable undo logging for non-aborting transactions; (4) speculatively commit the transaction at partitions that it no longer needs to access.

Data overflow. While H-Store is an in-memory database, it also utilizes a technique, called *anti-caching* [133], to allow data bigger than the memory size to be stored in the database, without much sacrifice of performance, by moving cold data to disk in a transactionally-safe manner, on the tuple-level, in contrast to the page-level for OS virtual memory management. In particular, to evict cold data to disk, it pops the least recently used tuples from the database to a set of block buffers that will be written out to disks, updates the evicted table that keeps track of the evicted tuples and all the indexes, via a special eviction transaction. Besides, non-blocking fetching is achieved by simply aborting the transaction that accesses evicted data and then restarting it at a later point once the data is retrieved from disks, which is further optimized by executing a pre-pass phase before aborting to determine all the evicted data that the transaction needs so that it can be retrieved in one go without multiple aborts.

Fault tolerance. H-Store uses a hybrid of fault-tolerance strategies, i.e., it utilizes a replica set to achieve high availability [36], [150], and both checkpointing and logging for recovery in case that all the replicas are lost [131]. In particular, every partition is replicated to k sites, to guarantee k-safety, i.e., it still provides availability in case of simultaneous failure of k sites. In addition, H-Store periodically checkpoints all the committed database states to disks via a distributed transaction that puts all the sites into a copy-on-write mode, where updates/deletes cause the rows to be copied to a shadow table. Between the interval of two checkpointings, command logging scheme [131] is used to guarantee the durability by logging the commands (i.e., transaction/stored procedure identifier and parameter values), in contrast to logging each operation (insert/delete/update) performed by the transaction as the traditional ARIES physiological logging does [146]. Besides, memory-resident undo log can be used to support rollback for some abort-able transactions. It is obvious that command logging has a much lower runtime overhead than physiological logging as it does less work at runtime and writes less data to disk, however, at the cost of an increased recovery time. Therefore, command logging scheme is more suitable for short transactions where node failures are not frequent.

3.1.2 Hekaton

Hekaton [37] is a memory-optimized OLTP engine fully integrated into Microsoft SQL server, where Hekaton tables[8] and regular SQL server tables can be accessed at the same time, thereby providing much flexibility to users. It is designed for high-concurrency OLTP, with utilization of lock-free or latch-free data structures (e.g., latch-free hash and range indexes) [86], [230], [231], and an optimistic MVCC technique [107]. It also incorporates a framework, called Siberia [134], [232], [233], to manage hot and cold data differently, equipping it with the capacity to handle Big Data both economically and efficiently. Furthermore, to relieve the overhead caused by interpreter-based query processing mechanism in traditional databases, Hekaton adopts the compile-once-and-execute-many-times strategy, by compiling SQL statements and stored procedures into C code first, which will then be converted into native machine code [37]. Specifically, an entire query plan is collapsed into a single function using *label*s and *goto*s for code sharing, thus avoiding the costly argument passing between functions and expensive function calls, with the fewest number of instructions in the final compiled binary. In addition, durability is ensured in Hekaton by using incremental checkpoints, and transaction logs with log merging and group commit optimizations, and availability is achieved by maintaining highly available replicas [37]. We shall next elaborate on its concurrency control, indexing and hot/cold data management.

Multi-version concurrency control. Hekaton adopts optimistic MVCC to provide transaction isolation without locking and blocking [107]. Basically, a transaction is divided into two phases, i.e., normal processing phase where the transaction never blocks to avoid expensive context switching, and validation phase where the visibility of the read set and phantoms are checked,[9] and then outstanding commit dependencies are resolved and logging is enforced. Specifically, updates will create a new version of record rather than updating the existing one in place, and only records whose valid time (i.e., a time range denoted by start and end timestamps) overlaps the logical read time of the transaction are visible. The uncommitted records are allowed to be speculatively read/ignored/updated if those records have reached the validation phase, in order to advance the processing, and not to block during the normal processing phase. But speculative processing enforces commit dependencies, which may cause cascaded abort and must be resolved before committing. It utilizes atomic operations for updating on the valid time of records, visibility checking and conflict detection, rather than locking. Finally, a version of a record is garbage-collected (GC) if it is no longer visible to any active transaction, in a cooperative and parallel manner. That is, the worker threads running the transaction workload can remove the garbage when encountering it, which also naturally provides a parallel GC mechanism. Garbage in the never-accessed area will be collected by a dedicated GC process.

Latch-free Bw-Tree. Hekaton proposes a latch-free B-tree index, called Bw-tree [86], [230], which uses delta updates to make state changes, based on atomic compare-and-swap (CAS) instructions and an elastic virtual page[10] management subsystem—LLAMA [231]. LLAMA provides a

8. Hekaton tables are declared as "memory optimized" in SQL server, to distinguish with normal tables.

9. Some of validation checks are not necessary, depending on the isolation levels. For example, no validation is required for *read committed* and *snapshot isolation*, and only read set visibility check is needed for *repeatable read*. Both checks are required only for serializable isolation.

10. The virtual page here does not mean that used by OS. There is no hard limit on the page size, and pages grow by prepending "delta pages" to the base page.

virtual page interface, on top of which logical page IDs (PIDs) are used by Bw-tree instead of pointers, which can be translated into physical address based on a mapping table. This allows the physical address of a Bw-tree node to change on every update, without requiring the address change to be propagated to the root of the tree.

In particular, delta updates are performed by prepending the update delta page to the prior page and atomically updating the mapping table, thus avoiding update-in-place which may result in costly cache invalidation especially on multi-socket environment, and preventing the in-use data from being updated simultaneously, enabling latch-free access. The delta update strategy applies to both leaf node update achieved by simply prepending a delta page to the page containing the prior leaf node, and structure modification operations (SMO) (e.g., node split and merge) by a series of non-blocking cooperative and atomic delta updates, which are participated by any worker thread encountering the uncompleted SMO [86]. Delta pages and base page are consolidated in a later pointer, in order to relieve the search efficiency degradation caused by the long chain of delta pages. Replaced pages are reclaimed by the epoch mechanism [234], to protect data potentially used by other threads, from being freed too early.

Siberia in Hekaton. Project Siberia [134], [232], [233] aims to enable Hekaton to automatically and transparently maintain cold data on the cheaper secondary storage, allowing more data fit in Hekaton than the available memory. Instead of maintaining an LRU list like H-Store Anti-Caching [133], Siberia performs offline classification of hot and cold data by logging tuple accesses first, and then analyzing them offline to predict the top K hot tuples with the highest estimated access frequencies, using an efficient parallel classification algorithm based on exponential smoothing [232]. The record access logging method incurs less overhead than an LRU list in terms of both memory and CPU usage. In addition, to relieve the memory overhead caused by the evicted tuples, Siberia does not store any additional information in memory about the evicted tuples (e.g., keys in the index, evicted table) other than the multiple variable-size Bloom filters [235] and adaptive range filters [233] that are used to filter the access to disk. Besides, in order to make it transactional even when a transaction accesses both hot and cold data, it transactionally coordinates between hot and cold stores so as to guarantee consistency, by using a durable update memo to temporarily record notices that specify the current status of cold records [134].

3.1.3 HyPer/ScyPer

HyPer [35], [236], [237] or its distributed version ScyPer [149] is designed as a hybrid OLTP and OLAP high performance in-memory database with utmost utilization of modern hardware features. OLTP transactions are executed sequentially in a lock-less style which is first advocated in [222] and parallelism is achieved by logically partitioning the database and admitting multiple partition-constrained transactions in parallel. It can yield an unprecedentedly high transaction rate, as high as 100,000 per second [35]. The superior performance is attributed to the low latency of data access in in-memory databases, the effectiveness of the space-efficient Adaptive Radix Tree [87] and the use of stored transaction procedures. OLAP queries are conducted on a consistent snapshot achieved by the virtual memory snapshot mechanism based on hardware-supported shadow pages, which is an efficient concurrency control model with low maintenance overhead. In addition, HyPer adopts a dynamic query compilation scheme, i.e., the SQL queries are first compiled into assembly code [112], which can then be executed directly using an optimizing Just-in-Time (JIT) compiler provided by LLVM [167]. This query evaluation follows a data-centric paradigm by applying as many operations on a data object as possible, thus keeping data in the registers as long as possible to achieve register-locality.

The distributed version of HyPer, i.e., ScyPer [149], adopts a primary-secondary architecture, where the primary node is responsible for all the OLTP requests and also acts as the entry point for OLAP queries, while secondary nodes are only used to execute the OLAP queries. To synchronize the updates from the primary node to the secondary nodes, the logical redo log is multicast to all secondary nodes using Pragmatic General Multicast protocol (PGM), where the redo log is replayed to catch up with the primary. Further, the secondary nodes can subscribe to specific partitions, thus allowing the provisioning of secondary nodes for specific partitions and enabling a more flexible multi-tenancy model. In the current version of ScyPer, there is only one primary node, which holds all the data in memory, thus bounding the database size or the transaction processing power to one server. Next, we will elaborate on HyPer's snapshot mechanism, register-conscious compilation scheme and the ART indexing.

Snapshot in HyPer. HyPer constructs a consistent snapshot by *fork*-ing a child process (via *fork()* system call) with its own copied virtual memory space [35], [147], which involves no software concurrency control mechanism but the hardware-assisted virtual memory management with little maintenance overhead. By *fork*-ing a child process, all the data in the parent process is virtually "copied" to the child process. It is however quite light-weight as the copy-on-write mechanism will trigger the real copying only when some process is trying to modify a page, which is achieved by the OS and the memory management unit (MMU). As reported in [236], the page replication is efficient as it can be done in 2 μs. Consequently, a consistent snapshot can be constructed efficiently for the OLAP queries without heavy synchronization cost.

In [147], four snapshot mechanisms were benchmarked: software-based Tuple Shadowing which generates a new version when a tuple is modified, software-based Twin Tuple which always keeps two versions of each tuple, hardware-based Page Shadowing used by HyPer, and HotCold Shadowing which combines Tuple Shadowing and hardware-supported Page Shadowing by clustering update-intensive objects. The study shows that Page Shadowing is superior in terms of OLTP performance, OLAP query response time and memory consumption. The most time-consuming task in the creation of a snapshot in the Page Shadowing mechanism is the copying of a process's page table, which can be reduced by using huge page (2 MB per page on x86) for cold data [135]. The hot or cold data is monitored and clustered with a