Knowledge and Information Systems (2019) 61:1209–1240
https://doi.org/10.1007/s10115-019-01340-7

SURVEY PAPER

In-memory transaction processing: efficiency and scalability considerations

Huiqi Hu[1] · Xuan Zhou[1] · Tao Zhu[1] · Weining Qian[1] · Aoying Zhou[1]

Received: 30 March 2018 / Revised: 20 January 2019 / Accepted: 23 January 2019 /
Published online: 13 February 2019

Abstract
Traditional disk-resident OLTP systems were mainly designed for computers with relatively small memory. Driven by the advance of hardware, OLTP systems need to be redesigned for larger memory and multi-core environments. Compared to disk-resident systems, in-memory systems have significant performance advantages, from the perspectives of both transaction throughput and query latency. Their performance is no longer limited by disk I/Os. Instead, the efficiency and scalability over multi-core CPUs become more important. In this paper, we survey and summarize a wide spectrum of design and implementation considerations that may affect the efficiency or scalability of an in-memory OLTP system. These considerations are concerned with most of the main components of databases, including concurrency control, logging, indexing and transaction compilation. For each of the components, we provide some in-depth analysis based on recent research works. This survey also aims to provide some guidance for designing or implementing high-performance in-memory OLTP systems.

Keywords Database system · Transaction processing · In-memory database · Concurrency control

✉ Huiqi Hu
hqhu@dase.ecnu.edu.cn

Xuan Zhou
xzhou@dase.ecnu.edu.cn

Tao Zhu
zhutao@stu.ecnu.edu.cn

Weining Qian
wnqian@dase.ecnu.edu.cn

Aoying Zhou
ayzhou@dase.ecnu.edu.cn

[1] School of Data Science and Engineering, East China Normal University, Shanghai, China

1 Introduction

Online transaction processing (OLTP) systems have been around, as a dominant tool that supports automated business, for decades. Traditional disk-resident OLTP database systems were mainly designed for computers with relatively small memory and a small number of CPUs. Such a system is typically composed of several functional components, including a buffer manager, a centralized locking infrastructure adopting a two-phase locking mechanism and a ARIES log manager [1]. These classic structures and mechanisms were quite successful in processing transactions of traditional enterprises, where the amounts of users and requests were usually limited. Today, Web applications are faced with much higher requirements, as they have to serve millions of users at the same time. Alibaba, the largest Chinese e-commerce platform, is a typical example. According to its reports, when launching a sales promotion, its back-end has to handle millions of concurrent data accesses per second, originated from users' browsing and purchasing requests [2]. Traditional OLTP systems can barely meet such needs.

Fortunately, OLTP systems are evolving rapidly, by taking advantage of the advance of hardware technologies over the past decades. Today, a low-end server can be equipped with 256 GB of RAM and 24 physical CPU cores, which yields a cluster of four servers with an aggregate 1 TB of RAM and a hundred cores. The architecture and functional mechanisms of traditional OLTP systems have undergone significant improvement, to make the best of the in-memory and multi-core environment. For instances, we can significantly enlarge the buffer and let data reside in memory to support much faster data access. Meanwhile, we can adopt a more lightweight concurrency control mechanism to overcome the performance hurdles of centralized locking. Moreover, memory conscious indexes can be used to further improve the performance. Such new approaches enable in-memory systems to outperform traditional disk-resident systems by several orders of magnitude. Recently, a number of in-memory OLTP database products have emerged, such as SAP HANA [3, 4], VoltDB [5], Hyper [6]. Reports show that they perform much better than traditional databases on standard benchmarks [5].

As the hardware environment has changed dramatically, a simple replacement of the disk storage with a memory storage cannot utilize the new environment effectively. New memory-oriented technologies need to be introduced. A significant amount of research works have been devoted to promote or analyze the new technologies. As the system performance is no longer constrained by disk I/Os, the focus of optimization has shifted to code efficiency and multi-core scalability. Code efficiency mainly refers to the reduction in the CPU cycles for transaction execution by optimizing its critical code path. Multi-core scalability refers the ability of a system to increase throughput by using more CPUs cores. In this paper, we survey the recent studies of OLTP from these two perspectives.

From the point of view of architecture, the back-end of an in-memory OLTP system mainly contains four components as illustrated in Fig. 1 (the frontend, e.g., network manager and query parser, is not the focus of our discussion). All of them need to be efficient and scalable. This paper will review and summarize the technical considerations on these main components and the possible solutions. The components include:

Concurrency control. Transaction management is the key function of an OLTP system. The ACID properties guarantee that a transaction is executed atomically without violating the database's integrity, consistency and durability. Concurrency control is meant to enforce isolated execution of concurrent transactions. Many efforts have been made

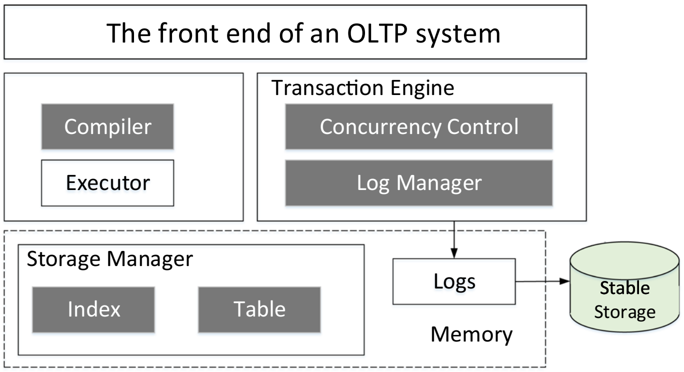

Fig. 1 The main components of an in-memory OLTP system

to improve its efficiency and scalability, by reducing the contention points of concurrency control protocols. We summarize the related works on four common CC mechanisms: (i) optimistic concurrency control (OCC); (ii) lock-based concurrency control; (iii) deterministic transaction execution, which sequences transactions prior to their execution; (iv) multi-version concurrency control. This part of the survey is presented in Sect. 2.

Logging and recovery. To preserve atomicity and durability of transactions in face of system failures, a logging and recovery mechanism is required. We split log management into three stages: (i) log generation, (ii) log persistence and (iii) log replay for recovery. There are several works aiming to improve the degree of parallelism of log generation and persistence, since its I/O latency can be a serious performance issue to in-memory systems. Some works even propose to replace disk with non-volatile memory to eliminate the I/O bottleneck. The related works are summarized in Sect. 3.

– *Index and data management*. Adopting an in-memory storage can reduce the complexity of data management, as data migration between disk and memory is no longer a concern. However, there are still efficiency- and scalability-related problems. Existing works studied memory optimized indexes which aim to maximize the proportion of cache-friendly data access. They also investigate how to effectively manage the limited memory space. We summarize them in Sect. 4.

– *Transaction compilation*. Efficient CPU utilization is critical for an in-memory database system. Transaction compilation (or query compilation) improves CPU utilization by converting query plans into efficient binary codes. It also exploits transaction semantics to generate better concurrent schedules. This part of works is summarized in Sect. 5.

In addition to the above technologies, distributed transaction management is also a major concern of today's transactional systems. By partitioning data over multiple nodes, we can further enlarge the memory capacity. We discuss the related works on distributed transactions in Sect. 6. Due to the architectural shift of database systems, their behavioral characteristics have also changed. Some studies have analyzed and compared the characteristics of a number of new in-memory systems. We review those studies in Sect. 7. Besides, we provide a brief comparison of some popular in-memory OLTP systems in Sect. 8. Furthermore, we identified some important issues of in-memory OLTP that have not been well recognized or widely discussed. We introduce them in Sect. 9.

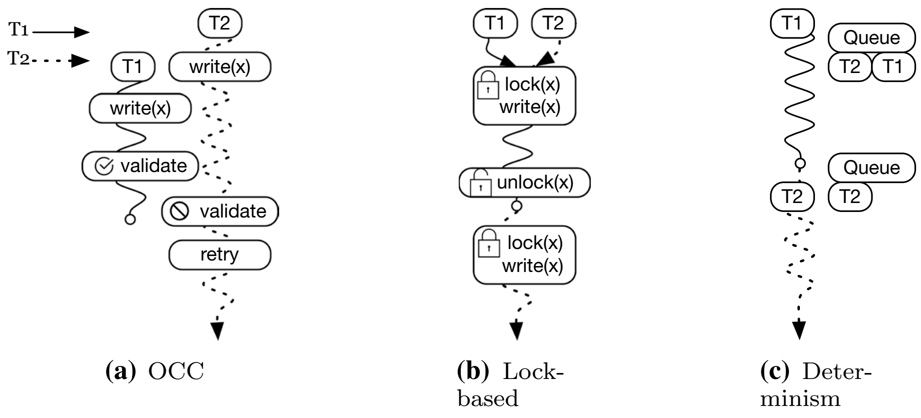

Fig. 2 Three concurrency control protocols

There have been several recent literature reviews on in-memory database systems, including the book authored by Faerber et al. [7] and the survey by Zhang et al. [8]. They mainly present some contemporary memory OLTP systems and their enabling technologies. While there is some overlap between this article and those reviews, there is more difference. The existing surveys discuss the design of database components from a relatively high-level perspective. In contrast, our survey discusses a much wider range of related research works and provide more details. We made the design and implementation considerations more explicit, so that it can be a guidance for developing future in-memory OLTP systems. Some of the considerations have been discussed explicitly in the literature, while the others are based on our own interpretation and conclusion. Taking the optimistic concurrency control in Sect. 2 as an example, we identify three design considerations: the efficiency of validation, decentralization of timestamp allocation and the cost of validation. There is no explicit mention of these three in the literature [7, 8], though there are works addressing them individually. The purpose of this paper is to identify the critical problems facing in-memory OLTP system, which the developers of future systems should pay close attention to.

2 Concurrency control

Concurrency control is an essential component in database systems, and various techniques have been proposed to improve its efficiency. We classify the techniques into four types of mechanisms. One is Optimistic Concurrency Control (OCC) which speculates conflicts through validation (Fig. 2a). One is lock-based concurrency control which pessimistically utilizes lock to prevent conflicts (Fig. 2b). We summarize the adoption of these two approaches in in-memory database in Sects. 2.1 and 2.2. OCC and lock-based methods are non-deterministic, i.e., the order of transactions is not determined until the execution. An alternative method is to sequence the transactions before execution (Fig. 2c). This is known as deterministic approaches. We summarize these types of techniques in Sect. 2.3. In practice, many database systems implements multi-version techniques to increase the degree of concurrency. The related works are discussed in Sect. 2.4.

2.1 Optimistic concurrency control

OCC was first proposed by Kung et al. [9]. In OCC, transactions are serialized through an assisted validation phase. During the phase, each transaction compares its read/write set against the concurrent ones to detect conflicts, and the transaction restarts if its validation fails. After a transaction is successfully validated, it is allocated with a transaction number that reflects its serial order. The primary argument of OCC is that conflicts rarely occur under the read-intensive workload. Its main advantage is that it sidesteps the overhead of maintaining a locking mechanism and the related movements to handle deadlocks.

It is worth noting that OCC is rarely used in disk-based databases. This is because OCC is not suitable for workloads containing a lot of conflicts. In a disk-based DBMS, transaction processing is often blocked by disk I/O. This produces two effects. Firstly, each transaction lasts longer due to the latency of I/O. Secondly, the system tends to serve more concurrent requests, so as to saturate the CPUs. As a result, the probability of conflicting data access increases vastly. A higher degree of contention implies a higher rate of validation failures and thus higher overheads. However, OCC is widely preferred by in-memory databases. As transactions no longer access the disk, they become much shorter, which leads to a lower degree of contention. Meanwhile, retrying a transaction is no longer expensive, as it does not involve I/Os. Thus, abortion and retry are not a big concern to an in-memory database.

Many studies have attempted to optimize OCC under the in-memory environment [6, 10–14]. We classify the existing works according to their goals of optimization:

Efficiency Consideration-I: Optimizing the validation phase [10–12, 15]. The very first OCC protocol [9] is not always efficient, since it requires a lot of works to validate the read set of a transaction against the write sets of all the concurrent transactions. To this end, Larson et al. [11, 12] adopted a self-validation mechanism for in-memory database. Instead of comparing the read/write set of concurrent transactions, it simply checks whether any record in the read set of a transaction has changed during its conduction. As such a validation method performs well in most cases, it was widely adopted by many in-memory systems (e.g., [10, 16]). In spite of this, self-validation is not always optimal in all cases. For instance, Neumann et al. [15] found that it is expensive for processing read-intensive transactions, because it has to track every single read access and the resulting read set can be too large. Hence, they proposed precision locking [17] (PL). Instead of maintaining a detailed read set, PL induces the predicate spaces of the accessed records (e.g., the records $x = \{1, 2, 3\}$ can be covered by the predicate space $1 \leq x \leq 3$). It then verifies the write set of a committing transaction against the predicate spaces of its concurrent transactions. The two aforementioned validation methods both have their own application scopes. When designing an in-memory OLTP system, one should identify a sweet spot of the existing approaches.

Scalability Consideration-I: Improving multi-core scalability by avoiding centralized transaction number generation [10, 13]. OCC usually requires a global counter to allocate transaction sequence numbers (or commit timestamps), which is a centralized contention point that limits the multi-core scalability, as identified in [10]. There are some works trying to avoid such centralized allocation. In [10], Tu et al. leveraged an epoch-based method in their system named *Silo*. Its major contribution is a new serializable commit protocol optimized for multi-cores. Different from the common OCC protocols, its assignment of transaction id is completely distributed without global critical section.

The transaction id is assigned collectively by a periodically updated global counter and a thread-local counter. The former only allocates a single number (named as an epoch, similar to a group id) for transactions committed in the same group. Within each thread, a local counter generates monotonically increasing numbers for its own transactions. This method benefits scalability. However, it complicates the durability mechanism, since the transaction ids do not represent the serial order of transactions. Silo treats epoches as the durability units, since only epoch numbers strictly reflect the commit order. This somehow increases the transaction latency of Silo. Meanwhile, since the transaction ids do not reflect the write-after-read dependency between transactions, Silo can only rely on value logging instead of command logging for recovery [18]. Yu et al. [13] proposed a data-driven timestamp management protocol. Instead of assigning timestamps to transactions, this protocol assigns read and write timestamps to accessed data items and uses them to compute a valid commit timestamp for each transaction. Thus, no centralized timestamp allocator is required. Whenever two transactions conflict, the generated commit timestamps always correctly reflect the serialization order.

Scalability Consideration-II: Efficient algorithm for retrying failed transactions [14]. OCC consumes extra CPU resources as failed transactions have to be aborted and retried repeatedly, especially under highly contended workloads. The retrying procedures will impose a serious impact on scalability. To reduce the overheads, Wu et al. [14] proposed transaction healing to better schedule a failed transaction. This approach needs to exploit the programming semantics and analyze the dependency among operations within a transaction. After validation fails, a healing phase is invoked to judiciously restore non-serializable operations. It saves CPU cost because only non-serializable operations are retried instead of all operations.

2.2 Lock-based concurrency control

Lock-based concurrency control, typically two-phase locking (2PL) [19], is a widely used concurrency control mechanism in conventional database systems. In a database system, locks are held to protect shared database contents (e.g., records, indexes, tables or data partitions) from uncontrolled accesses. Disk-based systems implement two-phase locking through a centralized lock table. In essence, the lock table's structure is a hash table, where conflicting lock requests are placed in the same hash bucket with a linked list. Accessing the lock table requires looking up the hash table and iterating over a linked lists, which incurs extra CPU overheads. Besides, it relies on latches to protect the internal structure of the lock table. In general, a latch is meant to schedule the accesses to an internal data structure from concurrent operations.[1] The lock table uses latches to schedule concurrent lock acquisition and releasing operations. Thus, when multiple cores try to acquire the same latch, it results in race conditions and impairs the multi-core scalability. Such an implementation has already been proven to be problematic [20, 21] even for the disk-based systems. The case is worse for the in-memory systems [22], as multi-core parallelism is more important to their performance. In recent years, a set of new designs are proposed to improve lock-based concurrency control schemes.

Efficiency Consideration-I: Reducing CPU instructions spent on acquiring/releasing locks [12, 23]. Maintaining a centralized lock manager is costly [22]. Considering

[1] Which is not visible at the transactional level.

processing a lock acquisition request, the lock manager should look up the hash table, iterate over a linked list and check whether any granted locking entries conflict with the new locking request. Compared to a centralized lock table, placing locks on tuples (row locking) seems more efficient to in-memory databases. Ren et al. [23] proposed a lightweight locking mechanism for in-memory database. Two lock fields are reserved for each record and directly stored in the header of the record. Obtaining the lock and retrieving the record itself can be serviced by a single memory access. Larson et al. [12] introduced a new way to implement lock-based concurrency control for Hekaton. Similarly, the method preserves 64 bits in the header of each record for storing the locking status. Locking and releasing the lock can be performed by modifying the 64-bit-sized variable with a simple compare-and-swap instruction. A side effect, however, is that row locking makes lock inheritance difficult.

Scalability Consideration-I: Reducing latch contention during lock acquisition/releasing [20, 21]. A centralized lock table usually relies on latches to protect its data structures from concurrent accesses incurred by lock acquisition and releasing. Latch contention on the lock manager severely constrains the multi-core scalability. Some in-memory databases replace the centralized locking scheme with some lightweight decentralized mechanisms, e.g., by stores locking information in the header of each record [12]. Hence, lock acquisition and releasing do not access any centralized data structure concurrently. In addition, latch-free implementation has been adopted to minimize the overheads of contention.

There are also works aiming at improving the multi-core scalability of a centralized lock table [20, 21]. While these techniques were primarily designed for disk-based systems, they work for main-memory systems. Jung et al. [21] proposed a new way to implement the lock manager, which takes advantage of compare-and-swap instructions and barriers to protect data structures instead of using latches. On the other hand, Johnson et al. [20] proposed speculative lock inheritance, where frequently acquired locks are passed directly from one transaction to another, to reduce the number of calls (release and acquire) to the lock manager. DORA [24] breaks the database into segments and let each segment be serviced by a single thread. Each part uses a thread-local lock table. This helps alleviate the contention introduced by a centralized lock manager. In addition, a transaction can parallelize its execution if there is no data dependence among the segments.

Scalability Consideration-II: Extracting parallelism by utilizing procedural logics [25–27]. A lock-based scheduler may under-utilize multi-core CPUs when scheduling workloads of high contention [28]. If too many requests are competing for the same lock, a lock holder blocks all the others from making process. As a result, the number of active transactions is reduced [29, 30]. Some work try to extract more concurrency for the lock-based mechanism [25, 26]. Xie et al. [25] claimed that it is expensive to use a single mechanism to ensure ACID properties uniformly across all transactions. They presented modular concurrency control, which partitions transactions into multiple groups and customizes concurrency control for each group based on the workload characteristic. Two-phase locking will isolate concurrent data accesses coming from different groups. For highly contending workload, aggressive mechanisms (e.g., transaction chopping [27]) are applied to increase concurrency. Narula et al. [26] introduced a new way named phase reconciliation to handle conflicting data accesses. When contention happens, the algorithm switches to a split phase to let each transaction operate on a copy of the record in parallel. A reconciliation phase then merges these copies into the global one. Such optimization works when operations on the contended record satisfy the commutative property (e.g., max, min, add). That is to say, changing the order of these operands does not change the final results. Overall, the above methods attempt to solve the problem of scheduling transactions with high contention. They extract more parallelism to access the contending tuples. However, their use somehow requires that we make the procedural logics of transactions explicit to the system.

2.3 Deterministic transaction execution

In a disk-based system, a transaction can be blocked by the slow disk access. Thus, concurrency control protocol is used to interleave execution of multiple transactions to improve the CPU utilization. In an in-memory system, slow disk access is no longer a problem. Stonebraker et al. [31] considered that it is possible to run multiple transactions one-by-one without interleaving their execution. In other words, instead of scheduling multiple transactions and interleaving their execution on the fly, it is possible to determine their serialization order before execution. Then, each transaction can be processed entirely without being interrupted. This is known as *deterministic execution.*

Efficiency Consideration-I: Eliminating concurrency control itself [5, 32]. The major advantage of deterministic execution is the complete elimination of concurrency control's overhead. `VoltDB`/`H-store` [5, 32] firstly proposes the deterministic execution strategy which uses a single-thread execution engine to process transactions. In the system, incoming transactions are queued and processed one-by-one by a single thread. As a result, the system neither uses concurrency control to isolate transactions nor maintains latches to protect the inner data structures.

Efficiency Consideration-II: Improving the performance by lazy execution. Almost all database systems should have a transaction fully executed before making responses to the clients. Faleiro et al. [33] proposed a lazy transaction execution engine for the deterministic database, which returns a commit/abort promise to clients without real transaction execution. A transaction is processed only when its results would be accessed by others. During the execution phase, a transaction can only be aborted by its client-defined logic. Thus, the technique determines whether one transaction is committable by checking all constraints during its initiation phase. The method increases the overall cache locality, as a repeatedly accessed record will be brought into CPU cache only once. This also helps to shorten the critical section, which improves the system's performance in processing high-contention workload.

Scalability Consideration-I: Parallelization of deterministic transaction processing [5, 32, 34, 35]. Running all transactions on a single thread cannot utilize the parallelism of multi-core CPUs. To parallelize, we can follow two patterns, known as "data-wise" and "transaction-wise" parallelism, respectively. In terms of data-wise parallelism, data are horizontally partitioned and transactions accessing different partitions are executed in parallel. For transaction-wise parallelism, transactions are assigned to a number of execution threads, which access a shared-data storage concurrently (i.e., each execution thread is allowed to access all the data). To leverage parallelism for deterministic transaction processing, H-store [5] exploits data-wise parallelism and Calvin [34] leverages both.

Basically, H-store partitions the database and assigns each partition to a single core. Given two transactions A and B that access different partitions, A and B can be executed concurrently, even though the deterministic order of A is in front of B. Calvin [34] achieves data-wise parallelism by partitioning the database over multiple nodes. More than that, on each node it implements transaction-wise parallelism. That is, a sequencer on each node places concurrent transactions into a global queue. Then, the transactions in the queue are processed by a pool of execution threads. The transaction executor utilizes a deterministic locking scheme to guarantee that the serial order specified by the sequencer is followed by the transactions. The deterministic locking scheme assumes the transactions' read and write sets are known in advance. In particular, it utilizes two-phase locking, while transactions have to obtain locks in the order given by the sequencer. A transaction begins its processing only after it acquires all needed locks. Therefore, multiple transactions can be processed in parallel only when they have no lock in common.

Scalability Consideration-II: Improving the performance of cross-partition transactions. As mentioned above, deterministic database systems conduct data partitioning to implement data-wise parallelization. Processing a transaction spanning multiple partitions will be a problem, as it may block other transactions from advancing. Realizing the limitation, Evan Jones et al. [36] proposed two methods to unlock more concurrency: (i) when a cross-partition transaction has completed on one partition p_1 and is still waiting for other partitions to complete, it allows single-partition transactions to be speculatively executed on p_1; (ii) it only allows deterministic execution when no cross-partition transactions exists, and uses 2PL to isolate data access. Pavlo et al. [37] proposed to utilize run-time behaviors of transactions to optimize the execution. For example, if a transaction is known to access only a single partition, it can be scheduled by determinism execution; otherwise, it is routed to the partition containing most records to be executed. Distributed commit protocol is required to coordinate cross-partition transactions [5, 32], which can be expensive. Calvin et al. [34] make use of determinism to avoid two-phase commit protocols. This will be investigated in Sect. 6.

2.4 Multi-version concurrency control

OCC and lock-based methods are treated as two basic methods for concurrency control. In practice, many database systems integrate a multi-version mechanism into them to enhance performance. In a multi-version database, updating a record creates a new version instead of overwriting the existing one. The main advantage of MVCC is that it potentially allows for greater concurrency by permitting parallel accesses on different versions. For instance, in a multi-version database, transactions are usually running under snapshot isolation. A transaction reads the latest version of records created before the start of the transaction. This design improves the degree of concurrency by isolating reading from writing. As a result, many in-memory databases are in favor of MVCC (e.g., SAP HANA, Hekaton, MemSQL). They utilize mechanisms commonly known as MVOCC or MV2PL [38, 39].

Efficiency Consideration-I: Improving the efficiency of MVCC. There is no "standard" implementation of MVCC despite the fact that many systems utilize it. To compare the performance of different implementations, Wei et al. [38] made a comprehensive empirical evaluation. They examined the design considerations of four key components: concurrency control, version storage, garbage collection and index management. Each component has multiple design or implementation choices. For example, they classify the storage structures of versions into three types, known as append-only storage, time-travel storage and delta storage. The limitations of the different designs and implementation choices are identified in this work.

They also evaluated some state-of-the-art implementations of MVCC. For instance, they studied the cooperative garbage collection mechanism used in Hekaton. Garbage collection is important for MVCC as the system will soon run out of space without it. Many DBMSs use background threads (called GC threads) that periodically scan the chains of records to expire out-of-date versions. The method is not very efficient since the system has to put its major resources into transaction execution and the number of GC threads is usually limited. To improve the efficiency, Hekaton [40] use a cooperative mechanism to clean expired version. In addition to background GC threads, the worker threads running transactions are also used to identify and remove the expired versions, which naturally balances the workload of garbage collection. The implementation of MVCC in an in-memory database is a fundamental problem, and [38] provides many insights into it.

In addition to the details of implementing MVCC, many works focus on providing serializable isolation for MVCC. Lowering the isolation levels can benefit the performance, because it avoids many potential conflicts. For instance, in snapshot isolation (SI), a transaction does not need to consider whether its read set has been updated. As a result, the cost of both detecting and handling conflicts can be saved. However, weaker isolation may fail to prevent some anomalies (e.g., the write skew anomaly [41]). Therefore, many works attempted to enable serializable isolation based on the implementation of SI.

Efficiency Consideration-II: Achieving serializable snapshot isolation (SSI) by identifying non-serializable patterns. To support serializable snapshot isolation [42–47], it is critical to prevent anomaly such as write skew from happening. To achieve that, Fekete et al. [42, 43] proposed to use dependency graph to identify non-serializable patterns, and modify the business logic to achieve serializability by inserting extra update SQLs into transactions. They created the algorithm called serializable snapshot isolation [44], which automatically detects and prevents anomalies. Following the theory in [48], it defines the "dangerous structure" that can cause anomalies. For each transaction T, it maintains two flags: (i) an in-conflict flag is used to indicate a write-after-read dependency (also known as the anti-dependency) from T' to T; (ii) an out-conflict flag is used to represent a write-after-read dependency from T to a third transaction T''. Both flags must be set, if a cycle occurs in the dependency graph [48]. Thus, SSI aborts the transaction T if both its flags are set. Obviously, this design is coarse-grained and conservative. It sometimes aborts transactions unnecessarily. For instance, the method is not suitable for processing transactions containing a lot of reads, because they are likely to be aborted [49]. The inventors of SSI further improved the algorithm by providing a more precise criterion to perform abortion [45, 47], and implement the method into a prototype system [49], which is suitable for processing read-mostly workload.

Scalability Consideration-I: Implementing scalable serializable snapshot isolation. Han et al. [50] evaluated the implementation of the SSI mechanisms proposed in [44, 45, 47]. They observed poor scalability in a multi-core environment and identified that the implementation of SSI introduces intensive latch contention when maintaining the dependencies [51]. Therefore, they propose a new multi-core scalable implementation of SSI by reducing its latch usage. It detects write-after-read dependency among concurrent transactions at run-time, without placing latches on the transaction manager's internal structures.

3 Logging and recovery

Durability is a critical requirement for transaction processing. It ensures that data loss does not happen when systems crash. To achieve that, most existing database systems adopt Write-Ahead Logging [1] (WAL). It maintains undo and/or redo log entries for each transaction. Undo entries are used to erase effects made by uncommitted transactions, while redo entries are used to restore modifications created by committed transactions. Overall, the implementation of the WAL can be decomposed into three parts: log generation, log

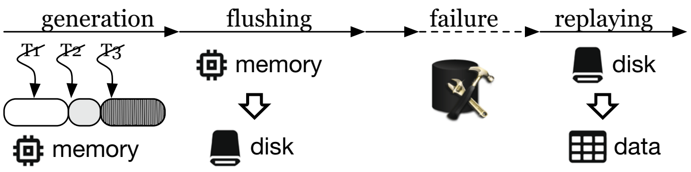

Fig. 3 Log management

persistence and log replaying. Figure 3 illustrates the whole procedure. Firstly, log generation creates log entries for each transaction and buffers them in the main memory. Secondly, log persistence forces generated undo/redo entries into the disk. Lastly, log replaying is invoked after system crashes, and a recovery procedure would replay log entries to restore the database. All these parts should be implemented carefully, to ensure efficiency or scalability. For instance, traditional disk-based databases usually keep all log entries in a centralized buffer before flushing them into the disk. The buffer could become a scalability bottleneck if too many threads try to fill log entries simultaneously. In the following, we summarize the recent efforts in improving the performance of these three parts.

3.1 Log generation

Efficiency Consideration-I: Reducing the amount of log entries generated by each transaction [11, 52, 53]. Logging that generates both redo and undo log entries for each transaction is time-consuming [22]. In an in-memory system, undo log may not be necessary, because the system does not need to flush uncommitted data into the durable storage. Thus, recovery does not need to erase any uncommitted data. In principle, redo log is sufficient to guarantee recoverability [11]. Malviya et al. [52] proposed a more aggressive *command logging* method. It stores neither the undo log nor redo log for transactions. Instead, it only stores the identity of the invoked transactions and their parameters. In comparison, these fields consume much less storage space than undo/redo entries. The limitation of command logging is that it does not support ad-hoc transactions. Besides, only transactions under serializable scheduling can perform command logging, since all the transactions must be re-executed in a serial order for recovery. This also increases the cost of recovery.

Scalability Consideration-I: Scaling log generation by reducing latch contention [54, 55]. Since the disk device only supports limited input/output operations per second, it is costly to run a disk write per log entry. Hence, an important technique named group commit [56] is used to improve disk utility. With group commit, a centralized log buffer will collect log entries from multiple transactions. A dedicated thread then periodically flushes the content in the buffer into the disk. Since multiple threads append log entries into the same buffer, latches are required to isolate concurrent data accesses. As a result, the technique results in extra latch contention.

The method proposed by Johnson et al. [54] can reduce log buffer contention by shortening the critical section on the centralized log buffer. Each transaction releases the mutex immediately after it occupies a buffer region and transactions which have acquired their buffer spaces can fill their log records asynchronously. Meanwhile, to guarantee correctness, transactions must release their buffer regions in order, to make sure that there is no gap in the log. Kim et al. [49] presented a latch-free centralized logging method. Before appending any entries into the log buffer, each transaction acquires an ordered LSN and buffer space using a single atomic instruction(`fetch-and-add`). To ensure that log records persist in sequence, the central log buffer is divided into multiple segments and buffer regions are released based on the order of the segments. June et al. also utilized atomic operations for buffer management. To deal with log gaps (which is called LSN holes) [57] , a thread is dedicated to maintain LSN (SBL). SBL represents the maximum LSN number which has no LSN hole before it. It indicates the safe point for log persistence. Two mechanisms, named hopping and crawling, were introduced to advance SBL. Wang et al. [55] proposed a decentralized log manager built on non-volatile memory (NVM). A transaction is considered committed once it writes its log entries into a NVM buffer. By using multiple NVM buffers in parallel, it eliminates the contention on a centralized buffer.

3.2 Log persistence

Efficiency Consideration-I: Reducing persistence latency by using advanced durable device [58, 59]. Persisting log entries in hard disk introduce high I/O latency, making log persistence the most costly stage for in-memory transaction processing. Some works try to reduce this part of latency by using more advanced devices. Non-Volatile Memory (NVM) or Storage Class Memory (SCM) [60] is persistent and byte-addressable. Its access latency is close to that of DRAM. Therefore, it is a promising hardware for improving the efficiency of logging and recovery.

In 2011, Fang et al. [58] introduced the first NVM-based logging approach. They showed that NVM-based logging can simplify the logging process, which is beneficial to both concurrency and latency. However, persistent write on NVM also raises new problems in handling system crashes. For instance, undo/redo log entries can be partially written into NVM before the system crashes. A recovery algorithm should identify the unused memory space and partially written log entries correctly. Huang et al. [59] exploited a variety of ways to adopt NVM in a database system. Its NV-Logging mechanism stores only log entries in the NVM, while its NV-disk mechanism uses NVM to store both data and log. They showed that NV-logging is much more cost-effective than the NV-disk approach.

Scalability Consideration-I: Utilizing multiple log devices [10]. Tu et al. [10] proposed an epoch-based commit protocol to accomplish parallel log persistence in their high-speed transaction system *Silo*, where the concept of the epoch is borrowed from their concurrency control protocol (see Sect. 2.1). In their approach, multiple log files are maintained and each log file is bound with a logging thread and a separate disk. Each logging thread is responsible for servicing a disjoint subset of working threads. Such a design leverages multiple log devices to improve transaction throughput.

3.3 Log replaying

Efficiency Consideration-I: Parallel recovery for command logging. Command logging requires less storage space and has a much lower run-time overhead. It improves the performance of transaction execution at the cost of recovery efficiency, as all the transactions must be re-executed to restore the database. To reach a middle ground, Yao et al. [53] proposed an adaptive logging approach by combining value logging and command logging, which can achieve faster recovery while preserving a comparable transaction throughput. It generates a dependency graph based on the parameters of command logging in the log file and organizes transactions in groups based on their dependency relationship. Therefore, groups without inter-dependency can perform their recovery in parallel. However, if the dependencies among transactions are complex, it may hinder the process of parallel recovery. As a remedy, the approach identifies highly dependent transactions based on a cost model and applies value logging to them. This further improves the degree of parallelism. Wu et al. [61] proposed another parallel recovery mechanism for command logging, called PACMAN. It first analyzes the stored procedures. Then, it detects the dependencies of SQLs both within and across transactions and decomposes them into multiple conflict-free units. Thus, units that are not mutually conflicting can be recovered in parallel. To further increase the parallelism for each unit, it exploits the run-time parameters of stored procedures to generate a more fine-grained parallel execution schedule at the recovery time.

Efficiency Consideration-II: Speeding up recovery with write-behind logging. Arulraj et al. [62] used NVM as data storage and proposed a new protocol named write-behind logging. In the logging protocol, updated data are flushed to the NVM before recording them in the log. After system crashes, the updates of all the committed transactions are guaranteed to be persistent (as NVM is non-volatile). Thus, data can be restored immediately. As the changes of uncommitted transactions may be persistent too, the system also persists the ids of active transactions in NVM. Then, it can simply discard the updates of uncommitted transactions during the recovery procedure. As discarding updates can be performed much faster than replaying updates, write-behind logging enables instant recovery. Moreover, since the contents of its logs are more lightweight, write-behind logging does not incur extra overheads than traditional WAL.

Scalability Consideration-I: Replaying log entries in a scalable way [18]. Apart from command logging, Zheng et al. [18] built *SiloR* to improve the scalability of recovery. SiloR relies on both redo logs and checkpoints to restore a database—correct database state can be generated by replaying the value logs on the last created checkpoint file. It performs parallel logging and checkpointing by using multiple threads, each of which writes a disjoint subset of records into a distinct disk. Then, logs on different disks can be replayed in parallel. In particular, SiloR allows checkpointing of an inconsistent snapshot of a database, that is, the effects of a committed transaction can be partially included in a snapshot. This enables SiloR to avoid synchronization among parallel checkpointing threads.

4 Index and data management

Traditional database indexes are designed to accelerate query on disk-based databases. In an in-memory database, the index resides in memory. Many efforts have been made to construct index in memory and to provide high-performance data access. Most of the efforts focus on tree-structured indexes (typically B-trees [63] and its variants). Compared to other types of indexes, such as hash, tree-structured indexes are more commonly used, due to its support for range query. We summarize the existing works as follows.

Efficiency Consideration-I: Reducing cache miss of index access. As indexes are maintained in the main memory, cache efficiency becomes more important. The basic unit of caching is known as cache line, holding recently referenced instructions and data. Modern CPUs typically have multiple levels of caches with different access speeds and capacities. The CPU starts to execute instruction only if both the required instruction and data reside in the cache (called cache hit). Otherwise, the penalty of cache miss comes into play and the CPU has to wait until they are fetched in the cache. From the perspective of cache efficiency, conventional B-tree styled index is not *cache-conscious* [64], as their design does not consider cache efficiency. Thus, a number of existing works focus on improving the cache efficiency of index [64–68].

(i) B-tree uses a large portion of index nodes to store child pointers [64]. When traversing from the root to a leaf node, each node access may incur a cache miss. In total, there can be n cache misses for one lookup operation, where n denotes the depth of the index. Clearly, n increases when the capacity of a node decreases. As an optimization, CSB^+-tree [64] and *CSS*-Tree [65] tried to eliminate unnecessary child pointers within a node. This enables a cache line to store more children and thus reduce the height of the tree. For instance, CSB^+-tree keeps only one child pointer per node and the rest of the children can be found by adding an offset to that pointer.

(ii) For conventional B-trees, the procedure of index traversal may not be propitious to exert instruction pipelines. This can cause another performance issue. Because the result of a comparison within each index node cannot be predicted easily, instruction cache misses can be common, which causes additional latency [66, 67]. The problem can be tackled by leveraging modern hardware architectures. For instance, the approaches of [66, 67] utilize SIMD to perform multiple comparisons simultaneously. Recently, Kraska et al. [69] put forward an idea that indexing can be seen as a machine learning model to map a key to the position (address) of a record. From this perspective, if a learned index can predict the position of target correctly, B-tree traversal is no longer necessary.

(iii) Mao et al. developed Masstree [68], which is a trie-concatenated B^+-tree that supports variable-length keys. Masstree partitions the keys into 8-byte slices and stores each key slice in a single B^+-tree node. Thus, all keys shorter than $8h+8$ bytes are stored at a B^+-tree of level h. Note that the 8-byte key slice is represented as a integer type (defined as $uint64\_t$). Under this design, Masstree can translate the expensive key comparison of arbitrary data types (e.g., strings) into fast integer comparison. This is the main reason of its high performance. Besides, design also allows it to prefetch the cache lines of nodes to improve cache efficiency.

Efficiency Consideration-II: Designing fast and memory efficient radix index. Some works focus on memory-based radix index [68, 70]. Radix tree (also known as trie, prefix tree) [71] is widely used for indexing character strings. It is composed of a set of strings, each containing m characters. An m-ary tree is used to connect the strings, such that each string is identified by a unique path from the root to a leaf node. There is also generalized radix [72] tree that can support arbitrary type of records by encoding bits into strings. Then, each inner node keeps a 2^s-array of pointers to its child, where s is the number of encoded bits. As a consequence, all operations have $O(k / s)$ complexity where k is the length of the key. One perceived advantage of radix tree is its search complexity, which only depends on the length of the keys instead of the number of records in the index. If s is large, the index will be efficient. This is attractive for indexing extremely large datasets. Some recent works [70, 73, 74] have indicated that the radix index outperforms B^+-trees in many cases. Nevertheless, a radix index also has a number of disadvantages. One downside is that it cannot customize key orders by specifying arbitrary comparators, which can limit its scope of application. Another disadvantage is that its space cost can be high, especially when s is large and most of its child pointers are null. To address the problem, Leis et al. [70] developed the adaptive radix tree (ART), which was adopted in the Hyper system [6]. By utilizing adaptive fanout, it can keep the space usage in control without scarifying the efficiency of search. In [75], the authors developed a radix index called HOT, featured by its ability to control the tree height. Reducing the tree height can further improve the efficiency of tree traversal. The method proposed in [75] further optimizes the node representation of HOT for better cache efficiency and SIMD utilization. Zhang et al. [76] presents a space-efficient radix index called fast succinct trie. It supports exact-match filtering (i.e., point query to identify the existence of a record) and range-filtering at the same time. It utilizes a LOUDS-based encoding method [77] to organize the index. Thus, the space consumption is somehow minimized from the perspective of information theory, while its performance is comparable to standard uncompressed indexes.

Efficiency Consideration-III: Improving space-efficiency by relocating cold data. Despite the rapid growth of memory capacity, main memory is still expensive in comparison with disk and SSD. A method to reduce the consumption of memory is to identify and relocate cold data. In particular, the patterns of most OLTP workloads are somehow skewed, and the distinction between hot (frequently accessed) and cold (infrequently or never accessed) data is somehow clear. Thus, it is usually beneficial to manage hot and cold record differently. The problem can be further divided into two issues.

(i) *Efficient cold data identification.* The first step is to identify cold data [78–80]. The simplest method is to monitor the read/write frequency of data records in each worker thread. This method obviously has negative impact on performance, due to its overhead. Thus, several studies have been conducted in attempt to move the monitoring task away from the critical path of transaction processing. Stoica et al. [78, 80] proposed to perform sampling online and identify hot data by analyzing transaction logs offline. They [81] also introduced a set of algorithms to estimate access frequencies, which are more accurate than traditional LRU-k and ARC algorithms [82]. Funke et al. [79] proposed to use a more efficient hardware-assistant approach, which monitors data accesses by reading the reset-flags within the CPU's memory management unit.

(ii) *Data migration and compression.* After cold data are identified, it will be either compacted or migrated into a secondary storage [78, 79, 83, 84].

Some works provide methods to migrate cold data. Stoica et al. [78] utilizes the default paging mechanism of the operating system for this purpose. As all the data are managed in a unified virtual memory space, it can simply out the cold data swap. In contrast to the page-level approach [78], the approaches of [80, 83] perform fine-grained migration at the tuple-level. They execute special transactions that select tuples and relocate them to the disk. When a transaction requires to use the cold data, they perform a pre-scan execution to identify the needed tuples and move them into the memory.

Cold data can also be compressed to save storage space [79, 84]. Funke et al. [79] proposed to compress immutable data using the common run-length and dictionary compression algorithm. Lang et al. [84] proposed a compression format called DATA BLOCKS. This advantage is that it does not introduce overhead to data access. It provides efficient point access and range scan algorithms.

Index entries also differ in their accessing frequencies [73, 85]. Zhang et al. [73] argued that memory indexes can be bigger than data itself. They presented a dual-stage architecture and a transformation procedure to reduce the space consumption of index. The basic approach is to leverage the skewed distribution of data accesses. Its dynamic stage resides in memory, and all the new entries are added to the dynamic stage. Thus, it is most likely that queries to the most recent entries will be evaluated in memory. The static stage resides in the durable devices (e.g., disk). As the size of the dynamic stage grows, it periodically moves the older entries to the static stage. Most popular indexes, such as B+tree, ART [70] and Masstree [68], can benefit from this design.

Scalability Consideration-I: Reducing latch usage on indexes. Latches are usually used to protect critical sections of indexes from concurrent threads. There have been a number of methods for dealing with concurrency control on B-trees [86, 87]. Early works rely on latch coupling, which perform latching heavily. The index traversal proceeds from one node to its child by holding the latch on the node while requesting the latch on a child. Latching coupling may seriously degrade the scalability due to its inherent cost and the coherence problem [88]—in a multi-core environment, different processors have to fetch index nodes into their caches frequently; as the states of latches keep changing, it is well possible that the states in the caches of different processors are inconsistent.

Several works aim to reduce the usage of latching [68, 85, 88, 89]. Some aim to eliminate latching from read operations. Cha et al. [88] proposed an optimistic concurrency control scheme that supports latch-free index traversal (read) for B^+-tree and its variants. For each node, it maintains a latch and a version number. Every write operation first obtains the node latch, and updates the node content. Then, it increases its version number, and releases the latch. To read a node, it adopts an optimistic strategy by first reading the version number of the node before accessing the data. At the end, it verifies whether the node latch is free and the current version number is equal to the previous one. If both conditions are true, the read operation succeeds. Masstree [68] uses a similar optimistic strategy to eliminate latching from read operations. It also uses a fine-grained latching scheme to enable concurrent write operations on different parts of the tree. The approaches in [68, 88] can eliminate latching from lookup operations. To reduce the usage of latch in write operations, Bw-tree [74, 85] abandons in-place updates and adopts delta updates. It is completely latch free. Each update on Bw-tree produces a new address in the form of a delta record attached to an existing page. Updates of records are concurrent, while only one winner can be installed on the index, which is achieved through an atomic compare-and-swap operation. Moreover, its delta update mechanism is more cache efficient than the in-place update mechanism, since it can reduce CPU cache invalidation.[2] In [74], further optimizations were conducted to improve the performance of Bw-tree.

The authors of [89] proposed in-memory B^+-trees that perform batch operations. Each batch can contain multiple writes and read operations. Special scheduling is performed within each batch to avoid concurrency control—all write operations are not executed until all read operations complete, and conflicting write operations are assembled into different rounds which are executed sequentially.

5 Transaction compilation

Transaction compilation (or query compilation) aims at generating better execution codes (or plans) for incoming requests. Transaction compilation is possible when UDF is used. UDF allows database systems to obtain the transaction logics in advance and avoid

[2] For in-place updates, the state of data is likely to be inconsistent in the caches of multiple processors.

client–server interaction during transaction processing. Transaction compilation increases CPU utilization in two ways: (1) optimized compilation can reduce extra instructions used in interpreted execution; (2) exploitation of application logics can generate better schedules with improved concurrency. We summarize them as follows.

Efficiency Consideration-I: Improving code efficiency through just-in-time compilation [11, 90]. The query plan of an SQL statement is an iterator tree, where each node corresponds to a physical operator. In the volcano model [91], records flow from the leaves to the root. Each operator exposes three interfaces: *open*, *next*, *close*, through which a parent node pulls records from its children. The overheads of such a query execution model are tremendous, and span several stages, such as data type interpretation, expression evaluation, function invocation, etc [11, 90]. Data interpretation and expression evaluation necessitate a significant amount extra instructions. Function invocation leads to poor instruction and data locality and more cache misses.

In systems such as Hekaton [11], through UDF query plans are converted into *C* codes as a single function. The function calls (i.e., open, next, close) between tree nodes are translated into *goto* and *label* statements. Then, Hekaton leverages the compiler and linker of Visual C to convert C code to efficient machine code. A C/C++ compiler usually takes several seconds to compile a query. To save the compiling time, Hyper [90] adopts a different design. It uses a Low-Level Virtual Machine (LLVM) compiler code to translate a query plan into portable assembler code. Later, a LLVM virtual machine can translate these codes into machine code. This approach takes only milliseconds to compile a query. Moreover, Hyper tries to compile multiple operators of a query plan into loop fragments. Each loop can perform multiple operations before moving each tuple out of CPU registers. Such a design significantly improves the data and code locality.

Scalability Consideration-I: Extracting more parallelism by exploiting application logics. By analyzing the transaction logic in advance, it is possible to generate specialized schedules and extract more parallelism from concurrency control.

Several works have exploited program semantics to specialize concurrency control protocols [14, 92]. For systems adopting 2PL, Yan et al. [92] proposed to automatically reorder the operations within a transaction, so as to postpone the execution of operations with high conflicting probability. This can shorten the critical sections of the transactions that compete for “hot” locks. This is achievable if we know the procedural logic of the application in advance. For systems adopting OCC, Wu et al. [14] proposed to capture dependency among operations within a transaction. Two operations are dependent if the output of the preceding one is the input of the subsequent one. This enables OCC to retry transactions through a healing phase, as described in Sect. 2.

A second way to improve currency is transaction chopping. The work of [27] proposed a SC-graph model for performing transaction chopping. In the model, an S-edge connects two sibling pieces of code from the same transaction, and a C-edge connects two pieces of codes of different transactions that may potentially conflict. An SC-cycle is a cycle containing both S-edge and C-edge. If no SC-cycle exists, each piece can be treated as an independent transaction without violating serializability. The approach in [93] utilizes the SC-graph model. It automatically splits transactions into fine-grained pieces, which allows for appearance of SC-cycles, and perform special scheduling on these pieces. IC3 [94] tracks the dependencies of the pieces online and blocks a piece from processing if its execution can potentially violate serializability. Compared with [27], these methods can extract more parallelism by partitioning transactions into smaller pieces. ROCOCO [95] further extends transaction chopping to distributed concurrency control. Servers firstly track dependencies among concurrent transactions without actually executing them. With the dependency information, during the actual execution, the servers know how to schedule the different pieces cleverly to achieve a serializable order. In general, transaction chopping only works well for long transactions. It requires a step to analyze program semantics, by either assuming the read/write set of transactions are known in advance or tracking the dependencies among transactions on-the-fly at the price of extra overhead.

6 Distributed transaction management

As the memory size of a single machine is limited, many systems partition data over multiple nodes as a way to increase memory capacity [5, 35]. It significantly improves scalability for transactions that access single partition but puts the execution of multi-partition transactions into a dilemma due to the cost of distributed transaction management. Note that distributed transaction management is a very broad research topic. The content is beyond the theme of this survey. Thus, we only discuss several studies investigating in-memory distributed systems, which include two aspects: (1) data partitioning approaches that aim to reduce the number of distributed transactions and skew accesses ; (2) new commit methods to replace the conventional two-phase commit protocol.

Scalability Consideration-I: Reducing distributed transactions and skew accesses with fine-grained partition. A fine-grained partition can reduce the number of distributed transactions, and hence reduce the cost of distributed commit. Several approaches have been studied for partitioning the data effectively [96–98]. Curino et al. [96] proposed a graph-based algorithm to minimize the number of distributed transactions when the workload is known in advance. The graph regards records as nodes and transactions as edges, which connect records used within the same transaction. Then, it applies a graph-partitioning algorithm to find balanced partitions that minimize the weight of cut edges. Thus, it also minimizes the number of distributed transactions.

The skewness of data accesses is another problem for data partitioning as the system performance decreases when the load is unbalanced among different nodes. Andrew et al. [97] took it into consideration and developed a tool to generate data partitions for deterministic OLTP systems. It selects the best data layout that minimizes the number of distributed transactions while also reducing the skewness of temporal data accesses. A cost model is proposed to evaluate the performance of DBMS for different schemes of partitions. The cost is computed based on a sample workload trace without actually deploying the database. An approximate algorithm is then used to compare the potential partitioning solutions and pick the one with the minimum cost. The works of [96, 97] assume that workload is aware and static. Taft et al. [98] proposed a reconfiguration system called E-Store which adjusts data partitioning dynamically for a shared-nothing DBMS. To identify hot records and their skewed access distribution, E-Store utilizes a two-stage monitoring tool. First, it identifies workload using a lightweight method which collects the OS-level statistics of CPU utilization for each partition. Once the first phase detects imbalanced workload, it starts the second phase of tuple-level monitoring over the entire system for a short period. Then, it uses the collected information to generate a new partitioning scheme, which may increase or decrease the number of nodes and reorganize the placement of the data records.

Scalability Consideration-II: Removing the expensive two-phase commit protocol. Many systems employ an agreement protocol (typically the two-phase commit protocol [99]) among all participating partitions to ensure transaction atomicity and durability. It requires multiple network round-trips among participant nodes at the commit time. Therefore, a distributed transaction is much more costly than a local transaction. To deal with the overhead raised from the agreement protocol, several alternatives to the two-phase commit protocol were proposed [35, 100].

Lin et al. [100] proposed a scheme to avoid two-phase commit by converting a distributed transaction into a local transaction. It is motivated as OLTP queries involve only a few data records. Thus, it is not more expensive to migrate records than to send sub-transactions to remote nodes. Therefore, before processing a transaction, it figures out which nodes the required records reside by performing a key-value lookup on a record-owner table. Next, it performs a migration process to move all the required records to a single machine and modifies the data ownership in the record-owner table. After the migration, the transaction is executed locally, and no distributed transactions will be performed. The method is less friendly when many concurrent partition transactions access the same record as the migration progress need to be scheduled.

Abadi et al. [35] optimized the commit protocol for Calvin, the partitioned database system employing a deterministic transaction execution policy. Calvin maintains replicas for each node and performs deterministic execution, where multiple replicas reach an agreement on the serial order of transactions in prior to their execution. As a result, a node failure will not block a transaction from committing, because a replica node is performing the same sequence of transactions. Thus, the distributed commit protocol of Calvin does not need to worry about node failures. Each node involved in a transaction only waits for single-round committing messages from the other nodes and commits once all are received. The method only works for system executing deterministic transactions on replicas with the same scheduling order.

7 System analysis and test

As the database architectures differ dramatically, their system behaviors are also different. A lot of efforts have been made to analyze the bottlenecks that affect the performance of the new systems.

CPU-cost analysis. Some studies made breakdown analysis, while the others go deeper to investigate the micro-behavior.

(i) *Experiment study.* Stonebraker et al. [22] are the first to realize that the database architecture has changed due to the advance of hardware. They removed each feature from a database system one at a time and identified the major components that affect system throughput. Ren et al. [101] compared deterministic and non-deterministic transaction execution. They conducted an experimental study to evaluate the advantages and disadvantages of determinism. Results show that a deterministic system produces extra latency, but it can scale to a higher throughput than non-deterministic systems.

(ii) *Micro-behavior analysis.* Some works analyze the in-memory database from the aspects of their micro-behavior. Ailamaki et al. [102] is the first paper to analyze the breakdown of execution time in a in-memory database system. It shows that a half of the execution time is spent in stalls due to L2 data cache misses and L1 instruction cache misses. Sirin et al. [103] compare the micro-architectural behavior (i.e., the number of instructions retired per cycle and the number of the misses from each level of cache) of in-memory database systems with the behavior of disk-based systems when running the same OLTP workload. They found that in-memory systems also under-utilize the micro-architectural features, just as disk-based systems do. Both of them spend more than half of the CPU cycles in instruction and data stalls, which is a result of poor L1 cache locality. The study of [104] further analyzes the leading cause of instruction and data misses. It identifies that the index probe operation is the leading cause of cache misses.

Scalability test. Regarding system scalability, Yu et al. [28] regards concurrency control as a potential bottleneck that limits the scalability of the system under multi-core environment. It provides an evaluation of the scalability of seven concurrency control schemes, by scaling up the number of cores to 1024 through a CPU simulator called Graphite [105]. The results show that all seven concurrency control schemes fail to scale to a large number of cores, but for different reasons. The work of [106] evaluates the scalability of several widely used database systems on multi-core architectures. Results demonstrate that contention over mutex causes the database performance to drop. As the number of cores increases, the performance decreases instead of increasing in spite of improved computing capacity. Appuswamy et al. [107] evaluated the impact of system architecture on the scalability of OLTP engines for contended workloads. They designed a testbed of main-memory database engine that implemented four system architectures. Then, they conducted analysis to characterize the interaction between system architectures and concurrency control protocols under contended workload.

8 In-memory OLTP system

In this section, we briefly review some existing in-memory database systems. There are increasing interests from both academia and industry in designing and developing in-memory database systems. In academia, there are many active research projects such as *H-store* [32], *HyPer/ScyPer* [6, 108] and *Silo* [10]. Many of them have successfully evolved into commercial systems(e.g., H-store and Hyper). In industry, many practical systems have come into the market, such as *SAP HANA* [4], *Microsoft Hekaton* [11], *Oracle TimesTen* [109], IBM *solidDB* [110], *MemSQL* [111].

Roughly, we can classify these systems into two categories based on their scale: non-distributed and distributed systems. Some early systems keep all the data in a single machine equipped with large memory. They are known as non-distributed databases (Sect. 8.1). However, their performance can be hindered by the memory capacity, the number of processors or other hardware resources. Distributed systems connect multiple servers to obtain more hardware resources (Sect. 8.2). We summarize the features of those systems in Table 1.

8.1 Non-distributed system

Hekaton. Hekaton [11] is an in-memory transaction processing engine fully built in Microsoft SQL Server. Transaction requests are encoded as stored procedures to make the best of the concurrency control and compilation mechanisms. Hekaton optimizes performance

Table 1 Comparison of in-memory OLTP systems

System	Index, storage	Concurrency control	Recovery	Memory underutilized	Query compilation and transaction API
Hekaton [11]	Bw-tree [85]	MVCC, OCC, 2PL [74]	WAL	Siberia [80, 81]	Compiled SQL [112]
Silo [10]	Masstree [68]	MVCC, OCC	WAL [18], epoch-based protocol [10]	–	Stored procedure
Hyper [6]	Hashing, balanced search tree, ART [70]	Determinism, virtual snapshot	WAL	Compressed columnar storage [84]	LLVM [113], compiled SQL [90]
solidDB [110]	Vtrie [114]	MVCC, OCC, 2PL	WAL, Log-based replication [115]	Disk-based engine	SQL, direct link
TimesTen [109]	Hashing, bitmap [116]	MVCC, 2PL	WAL, log-based replication	Column compression	SQL, direct link, JDBC/ODBC
Altibase [118]	Hashing, range index	MVCC, 2PL	WAL, log-based replication	Column compression	SQL, JDBC/ODBC
Peloton [119, 120]	Bw-tree [74]	MVCC, timestamp ordering [13]	WAL, write-behind logging [62]	–	SQL, JDBC
HANA [3]	Multi-stage storage [121], CSB$^+$-tree [64], etc	MVCC, 2PL	WAL	Column-store, compression	SQL Script, SQL, etc, JDBC/ODBC
VoltDB [5]	B+-tree, hashing, binary tree	Determinism [36]	Command logging [52]	Anti-caching [83]	Stored procedure, reflection
MemSQL [111]	Hashing, skip lists	MVCC, 2PL	WAL	Column-store, compression, disk storage	LLVM [113], compiled SQL

by using latch-free data structures (e.g., Bw-trees [85]) and a multi-version optimistic concurrency control scheme [12]. It also compiles stored procedure into efficient machine code [112]. Later, the Siberia component [81] is used to classify records into cold and hot ones based on access frequencies. It controls memory usage by shifting cold ones into disk [80]. Durability and recovery are ensured by write-ahead logging and checkpointing.

Silo. Silo [10] is an in-memory database prototype. It supports transactions in the form of stored procedures and is designed to scale on large multi-core machines. Silo stores records with Masstree [68] indexes. The key to its good performance is its concurrency control method. As described in Sect. 2, it combines OCC with an epoch-based commit protocol. Silo relies on both redo logging and checkpointing to restore the database. Both logging and checkpointing are parallelized [18]. Checkpoints and logs are written into different disks of a machine. During recovery, checkpoint files in different disks are first rebuilt in the memory, followed by log replaying to bring the database to a consistent state.

Hyper. Hyper [6] is the main-memory system designed for hybrid transactional and analytical workload. For transaction processing, Hyper executes transaction sequentially following the design of determinism database [5, 34] and logically partition the database to utilize multi-core parallelism. On the other hand, analytical queries execute on a read-only virtual snapshot generated by the hardware-level copy-on-write mechanism [6]. A virtual memory snapshot is created by a child process of the OLTP process via the *fork*() system call. The snapshot stays in precisely the state that the fork() takes place. To improve query efficiency, it converts user-input queries into assembly codes using the LLVM [113]. For data storage, Hyper uses the adaptive radix tree(ART) [70] to achieve both space and time efficiencies. To reduce memory usage, cold data are stored in a compressed columnar format, named *Data Blocks* [84].

solidDB. IBM solidDB [110] supports in-memory storage as well as disk-based storage for high-performance transaction processing. MVCC and 2PL are used to isolate conflicting data accesses in the memory storage. It uses the index of Vtrie [114] tree to accelerate data access. To provide durability and availability, solidDB performs both checkpointing and replication. A failed instance can recover database from either the checkpoint file or its hot-standby [115]. SolidDB can replicate data to both another solidDB instance and other data servers. Between solidDB instances, it uses statement-based replication. Between solidDB and other data servers, log-based replication is used. For query processing, a client application can link to the database server routines directly.

TimesTen. ORACLE TimesTen [109] is an in-memory database, which is designed for both OLTP and OLAP workloads. Its primary database instance is responsible for OLTP workloads. Data are synchronized into a standby instance handling OLAP workloads. For concurrency control, it uses a fine-grained locking scheme for update-intensive workloads and multi-versioning for read-only workloads. It supports hash index, bitmap indexes [116] and range indexes. The storage engine also offers columnar compression to reduce the memory usage. It uses write-ahead logging to ensure durability. A multi-partition log buffer is designed to avoid centralized log generation. For high availability, it uses log-based replication and provides the 2-safe replication mode [117]. For query processing, TimesTen supports ODBC and JDBC. It also allows a client to link into the server directly.

Altibase. Altibase is a database provided by the Altibase Corporation [118]. It uses a hybrid architecture where the high-performance data are stored in an in-memory engine and the voluminous data are kept in a disk-based engine. Altibase manages concurrency using MVCC. A lock-based method is utilized over the MVCC to guarantee the ACID. For data residing in memory, records are organized in pages. Indexes, such as hash indexes, range indexes, and spatial indexes are supported. Altibase uses fuzzy and ping-pong checkpointing methods to back up the most recent state of the database and uses write-ahead logging for recovery [118]. It also employs log-based replication for availability. It offers ODBC/JDBC and C++ procedure interface to clients.

Peloton. Peloton is an open-source in-memory database system developed by Carnegie Mellon University [119, 120]. A main feature of the system is that it is autonomous, as the system can tune itself to optimize performance using artificial intelligence techniques. Peloton re-implements the latch-free Bw-tree for in-memory data access [74]. For concurrency control, Wu et al. [38] have evaluated both OCC and 2PL combined with the MVCC in the system. It further provides a data-driven timestamp ordering protocol to serialize transactions [13]. Peloton leverages write-ahead logging when data are stored in memory. It also supports byte-addressable NVM storage, where data are persistent on NVM devices. For deployment on NVM, write-behind logging is utilized [62]. For accessing, it offers JDBC interface.

8.2 Distributed system

HANA. SAP HANA [3, 4] is a distributed, in-memory database system designed for hybrid OLTP and OLAP workloads. It is able to process relational data, semi-structured and unstructured data. To achieve that, it implements a multi-tier in-memory storage engine [3]. The main store is a highly compressed column-orientated storage, which reduces memory usage and helps in complex query processing. To efficiently support OLTP workloads, HANA adopts a multi-tier [121] storage. Transactions writes are firstly kept in `L1-delta`, a write-optimized row-store; and then propagated into `L2-delta` in the format of column-store. At last, the L2-delta structure is periodically merged into the main store. For concurrency control, it implements MVCC and distributed locking. An optimized two-phase commit protocol [4] was designed to ensure atomicity and reduce the overhead of distributed transaction processing. Durability is implemented through write-ahead logging and checkpointing. In query processing, applications can use SQL (via JDBC/ODBC) or SQL Script to operate on the system. HANA performs horizontal partitioning to scale out. It offers an interface for a DBA to directly assign partitions to individual HANA nodes. Therefore, the DBA can assign all "transactionally hot" partitions on a single server to avoid distributed transactions.

VoltDB. VoltDB [5] is a distributed, in-memory, deterministic relational database system. The database is horizontally partitioned. Each partition binds to a site, where transactions are serially executed by a single thread. Durability is ensured by command logging [52] and checkpointing. VoltDB improves system availability by maintaining replicas for each data partition on different nodes. Each transaction is sent to all replicas and processed in parallel. As its concurrency control is deterministic, multiple replicas do not get diverged by replicating transaction input. When handling a database larger than the memory provided, an anti-caching mechanism [83] is used to evict cold records into the disk. For query processing, transaction requests are expressed in Java stored procedures. The server invokes stored procedures using the Java reflection mechanism. VoltDB originates from the H-Store project [32].[3] H-Store made innovation on concurrency control [36], distributed transaction processing [37] and auto partitioning algorithms [97].

[3] The project is ended in 2016.

MemSQL. MemSQL [111] is a distributed, in-memory, shared-nothing relational database system. It was designed for OLTP as well as OLAP workload. MemSQL uses MVCC for read requests and performs locking for handling write–write conflicts. Records are organized by latch-free skip lists to support fast random access. Durability is supported by persisting write-ahead logs and checkpoints into the disk. For query processing, MemSQL adopts Just-In-Time compilation to convert SQL into bytecode and compile them into machine codes through LLVM [113]. For multi-node scaling, it adopts a two-tiered clustered architecture with two types of nodes. The *leaf* node acts as data storage node responsible for processing queries. The *aggregator* node routes queries to proper leaf nodes, aggregates results and responds to clients. Tables are horizontally partitioned over leaf nodes. For high availability, each data partition has a master and a slave versions on two leaf nodes. Data are written into the master version and synchronized to the slave using log-based replication.

Summary Table 1 summarizes these systems and their technologies. As we can see, different main-memory systems have adopted quite different index strategies, while all traditional disk-based systems tend to use B-tree (B^+-tree). For concurrency control, MVCC and lock-based protocol are widely adopted by many systems. For recovery, write-ahead logging is commonly implemented to avoid data loss in case of system crashes. Only VoltDB uses a different method, known as command logging. To save memory space, systems usually rely on data compression or shift cold data into the disk. For query processing, many systems choose to use Just-In-Time Compiling, which helps to reduce the number of CPU instructions. At last, horizontal partitioning is mostly used by distributed systems to scale out.

9 Issues for further research

In this section, we propose some issues, which are not widely discussed and may deserve more attention from the research community.

Issue I: Interactive transaction processing for in-memory database. To achieve better performance, in-memory databases usually assume the one-shot transaction model (e.g., [5, 10, 12, 27, 35, 93, 95, 122]), where transactions are run as stored procedures and no client–server interaction is involved. One-shot transactions do not worry about I/O-related stall between client and server. They keep running to completion if no conflict is observed. Although one-shot transaction processing can achieve impressive throughput, it ignores an important kind of workload, interactive transactions. In many real-world cases, applications operate on databases using SQLs and JDBC/ODBC interfaces. Transactions are executed by sending SQLs one-by-one to the server. A recent report [123] has shown that interactive transactions are much more common than one-shot one in real-world applications. It is reported that 54% responders never or seldom use stored procedure in their DBMSs. Only 16% responders have more than half of transactions run as stored procedures. The reasons for the situation are twofold [124]: (i) stored procedures are difficult to maintain and debug; (ii) they lack portability, making it hard to deploy applications on different platforms and databases. We can conclude that the performance issues of the interactive model are: (i) a transaction can be stalled by network I/O, which requires the execution engine to handle the network I/O efficiently; (ii) a transaction lasts much longer since network latencies are included, which results in more access conflicts. Such differences demand an efficient concurrency control mechanism that is different from that of one-shot transactions.

Issue II: Transaction management with log-based replication. Replication is an effective mechanism to provide high availability and fault tolerance in database systems. However, existing schemes, such as eager or lazy (log-based) replication [125], can hardly satisfy the requirement of both performance and consistency (i.e., data in the primary server and replicas are consistent). There are many NoSQL databases which utilize replication for high availability. They ignore the ACID properties of transaction processing. For instance, Amazon Dynamo [126] and Facebook Cassandra [127] offer "always-on" services, but sacrifice the consistency of read. Bailis et al. [128] introduced the concept of Highly Available Transactions (HATs), a transactional guarantee that does not suffer unavailability or incurs high network latency during system partitions. He pointed out that transaction processing under weak isolation level (e.g., read committed) can achieve high availability in distributed database systems. However, serializable transactions are not achievable in the presence of network partitions. Paxos, which is first described by Leslie Lamport in [129, 130], provides us with a compromise. It can tolerate N server faults (e.g., network partition or system crashes) in a cluster configured with $2N + 1$ servers. Therefore, to provide highly available services, modern database systems often adopt Paxos protocol to replicate data from primary to backup replicas, such as MegaStore [131], Spanner [132] and Spinnaker[133].

Invoking log-based replication has a significant impact on single-node transaction management, especially in terms of the transaction commit protocol. In addition to ensuring log persistence on the primary server, database systems based on Paxos are required to synchronize each log entry to the majority of replicas. This imposes negative impacts on the performance of transaction commit. How to reduce the impact on transaction commit is an unsolved problem, which requires more study. One solution is to relax the constraints on data consistency. Although some isolation levels (e.g., snapshot isolation) are weaker than serializability, they are widely used in real-world applications. Therefore, how to implement a high-performance OLTP system for high availability under these isolation levels is an important topic. Further, it is common that a cluster of commodity machines are equipped with new hardware, including NVM and RDMA. Both of them can have a nonnegligible impact on log-based replication. The former can change the mechanism of logging (e.g., [62]), while the latter can significantly improve the network I/O between replicas. To this end, the protocol of log-based replication may need to be revisited.

Issue III: The way to monitor and understand the running workload. Many studies have proven that a database system can perform better if it has knowledge about the workload. For instance, there are a number of methods designed for read-intensive workload (e.g., [15, 49, 88]). And some methods prefer to distinguish hot and cold data (e.g., [25, 73, 98]). A primary problem to leverage these techniques is to understand the workload. Existing works have provided some ad-hoc solutions. However, we believe there are still many issues to be resolved. First, we need to choose monitoring the workload outside or inside of a database system. Many systems trace the log of workload outside of the database (e.g., [78–80, 119]). It enables precise analysis but introduces time delay. Online monitoring is necessary if the workloads of applications are unpredictable. For example, a peak of transaction load will appear immediately if a sale promotion is conducted on the Internet. It is infeasible for the database system to detect those events. Thus, we need an effective mechanism to support and combine both outside and inside monitoring. Second, inside monitoring can cause performance penalty to transaction processing, and inside monitoring with high efficiency is very difficult to implement. We can choose different monitoring granularities. Information about workloads can be observed and recorded at the partition-level, the tuple-level or even to the degree of memory accessing footprints. But how to find a trade-off between the granularity and its performance penalty is still an open problem. Besides, there may be some simple tricks. For instance, it is interesting that for OLAP query, many methods determine whether a query is network consuming by dynamically observing the sending/dispatching queue of data packets. Maybe it is also possible to leverage the internal data structure of OLTP to obtain some valuable information about workload.

10 Conclusion

In this paper, we performed an extensive review of in-memory OLTP systems. Most of the existing methods focus on improving the system's efficiency or scalability. We summarize the research considerations based on the core components of an in-memory system, including concurrency controller, logging, indexing and transaction compilation. We also briefly surveyed several popular systems and discussed future research issues.

Acknowledgements This work is supported by National Science Foundation of China under Grant Numbers 61702189, 61672232, 61772202 and Youth Science and Technology - Yang Fan Program of Shanghai under Grant Number 17YF1427800.

References

1. Mohan C, Haderle D, Lindsay B, Pirahesh H, Schwarz P (1992) Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. TODS 17(1):94–162
2. Alibaba single day record. https://techcrunch.com/2017/11/11/alibaba-smashes-its-singles-day-record/. Accessed 2018
3. Färber F, Cha SK, Primsch J, Bornhövd C, Sigg S, Lehner W (2012) SAP HANA database: data management for modern business applications. SIGMOD Rec 40(4):45–51
4. Lee J, Kwon YS, Färber F, Muehle M, Lee C, Bensberg C, Lee JY, Lee AH, Lehner W (2013) SAP HANA distributed in-memory database system: transaction, session, and metadata management. In: ICDE. IEEE, pp 1165–1173
5. Stonebraker M, Weisberg A (2013) The VoltDB main memory DBMS. IEEE Data Eng Bull 36(2):21–27
6. Kemper A, Neumann T (2011) Hyper: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE. IEEE, pp 195–206
7. Faerber F, Kemper A, Larson P-Å, Levandoski J, Neumann T, Pavlo A et al (2017) Main memory database systems. Found Trends Databases 8(1–2):1–130
8. Zhang H, Chen G, Ooi BC, Tan K-L, Zhang M (2015) In-memory big data management and processing: a survey. IEEE Trans Knowl Data Eng 27(7):1920–1948
9. Kung H-T, Robinson JT (1981) On optimistic methods for concurrency control. TODS 6(2):213–226
10. Tu S, Zheng W, Kohler E, Liskov B, Madden S (2013) Speedy transactions in multicore in-memory databases. In: SOSP. ACM, pp 18–32
11. Diaconu C, Freedman C, Ismert E, Larson P-A, Mittal P, Stonecipher R, Verma N, Zwilling M (2013) Hekaton: SQL server's memory-optimized OLTP engine. In: SIGMOD. ACM, pp 1243–1254
12. Larson P-Å, Blanas S, Diaconu C, Freedman C, Patel JM, Zwilling M (2011) High-performance concurrency control mechanisms for main-memory databases. Proc VLDB Endow 5(4):298–309
13. Yu X, Pavlo A, Sanchez D, Devadas S (2016) TicToc: time traveling optimistic concurrency control. In: SIGMOD, vol 8, pp 209–220

14. Wu Y, Chan C-Y, Tan K-L (2016) Transaction healing: scaling optimistic concurrency control on multicores. In: SIGMOD. ACM, pp 1689–1704
15. Neumann T, Mühlbauer T, Kemper A (2015) Fast serializable multi-version concurrency control for main-memory database systems. In: SIGMOD. ACM, pp 677–689
16. Loesing S, Pilman M, Etter T, Kossmann D (2015) On the design and scalability of distributed shared-data databases. In: SIGMOD. ACM, pp 663–676
17. Jordan J, Banerjee J, Batman R (1981) Precision locks. In: SIGMOD. ACM, pp 143–147
18. Zheng W, Tu S, Kohler E, Liskov B (2014) Fast databases with fast durability and recovery through multicore parallelism. In: OSDI, pp 465–477
19. Eswaran KP, Gray JN, Lorie RA, Traiger IL (1976) The notions of consistency and predicate locks in a database system. Commun ACM 19(11):624–633
20. Johnson R, Pandis I, Ailamaki A (2009) Improving OLTP scalability using speculative lock inheritance. Proc VLDB Endow 2(1):479–489
21. Jung H, Han H, Fekete A, Heiser G, Yeom HY (2014) A scalable lock manager for multicores. TODS 39(4):29
22. Harizopoulos S, Abadi DJ, Madden S, Stonebraker M (2008) OLTP through the looking glass, and what we found there. In: SIGMOD. ACM, pp 981–992
23. Ren K, Thomson A, Abadi DJ (2012) Lightweight locking for main memory database systems. In: VLDB. vol 6, pp 145–156
24. Pandis I, Johnson R, Hardavellas N, Ailamaki A (2010) Data-oriented transaction execution. Proc VLDB Endow 3(1–2):928–939
25. Xie C, Su C, Littley C, Alvisi L, Kapritsos M, Wang Y (2015) High-performance acid via modular concurrency control. In: SOSP. ACM, pp 279–294
26. Narula N, Cutler C, Kohler E, Morris R (2014) Phase reconciliation for contended in-memory transactions. In: OSDI, pp 511–524
27. Shasha D, Llirbat F, Simon E, Valduriez P (1995) Transaction chopping: algorithms and performance studies. TODS 20(3):325–363
28. Yu X, Bezerra G, Pavlo A, Devadas S, Stonebraker M (2014) Staring into the abyss: an evaluation of concurrency control with one thousand cores. Proc VLDB Endow 8(3):209–220
29. Agrawal R, Carey MJ, Livny M (1987) Concurrency control performance modeling: alternatives and implications. TODS 12(4):609–654
30. Thomasian A (1993) Two-phase locking performance and its thrashing behavior. TODS 18(4):579–625
31. Stonebraker M, Madden S, Abadi DJ, Harizopoulos S, Hachem N, Helland P (2007) The end of an architectural era: (it's time for a complete rewrite). In: VLDB. VLDB Endowment, pp 1150–1160
32. Kallman R, Kimura H, Natkins J, Pavlo A, Rasin A, Zdonik S, Jones EP, Madden S, Stonebraker M, Zhang Y et al (2008) H-store: a high-performance, distributed main memory transaction processing system. VLDB 1(2):1496–1499
33. Faleiro JM, Thomson A, Abadi DJ (2014) Lazy evaluation of transactions in database systems. In: SIGMOD. ACM, pp 15–26
34. Thomson A, Abadi DJ (2010) The case for determinism in database systems. Proc VLDB Endow 3(1–2):70–80
35. Thomson A, Diamond T, Weng S-C, Ren K, Shao P, Abadi DJ (2012) CalvIn: fast distributed transactions for partitioned database systems. In: SIGMOD, pp 1–12
36. Jones EP, Abadi DJ, Madden S (2010) Low overhead concurrency control for partitioned main memory databases. In: SIGMOD. ACM, pp 603–614
37. Pavlo A, Jones EP, Zdonik S (2011) On predictive modeling for optimizing transaction execution in parallel OLTP systems. Proc VLDB Endow 5(2):85–96
38. Wu Y, Arulraj J, Lin J, Xian R, Pavlo A (2017) An empirical evaluation of in-memory multi-version concurrency control. Proc VLDB Endow 10(7):781–792
39. Weikum G, Vossen G (2001) Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. Elsevier, Amsterdam
40. Diaconu C, Freedman C, Ismert E, Larson P-Å, Mittal P et al (2013) Hekaton: SQL server's memory-optimized OLTP engine. In: SIGMOD, pp 1243–1254
41. Berenson H, Bernstein P, Gray J, Melton J, O'Neil E, O'Neil P (1995) A critique of ANSI SQL isolation levels. SIGMOD Rec 24:1–10
42. Fekete A, Liarokapis D, O'Neil E, O'Neil P, Shasha D (2005) Making snapshot isolation serializable. TODS 30(2):492–528
43. Jorwekar S, Fekete A, Ramamritham K, Sudarshan S (2007) Automating the detection of snapshot isolation anomalies. In: VLDB, pp 1263–1274

44. Cahill MJ, Röhm U, Fekete AD (2009) Serializable isolation for snapshot databases. DMoNH 34(4):20
45. Revilak S, O'Neil P, O'Neil E (2011) Precisely serializable snapshot isolation (PSSI). In: ICDE. IEEE, pp 482–493
46. Ports DR, Grittner K (2012) Serializable snapshot isolation in PostgreSQL. Proc VLDB Endow 5(12):1850–1861
47. Wang T, Johnson R, Fekete A, Pandis I (2015) The serial safety net: efficient concurrency control on modern hardware. In: DMoNH. ACM, p 8
48. Adya A, Liskov BH (1999) Weak consistency: a generalized theory and optimistic implementations for distributed transactions. Doctoral dissertation, Massachusetts Institute of Technology
49. Kim K, Wang T, Johnson R, Pandis I (2016) ERMIA: fast memory-optimized database system for heterogeneous workloads. In: SIGMOD. ACM, pp 1675–1687
50. Jung H, Han H, Fekete A, Röhm U, Yeom HY (2013) Performance of serializable snapshot isolation on multicore servers. In: DASFAA (2), Lecture Notes in Computer Science. Springer, pp 416–430
51. Han H, Park S, Jung H, Fekete A, Rohm U, Yeom HY (2014) Scalable serializable snapshot isolation for multicore systems. In: ICDE
52. Malviya N, Weisberg A, Madden S, Stonebraker M (2014) Rethinking main memory OLTP recovery. In: ICDE. IEEE, pp 604–615
53. Yao C, Agrawal D, Chen G, Ooi BC, Wu S (2016) Adaptive logging: optimizing logging and recovery costs in distributed in-memory databases. In: SIGMOD. ACM, pp 1119–1134
54. Johnson R, Pandis I, Stoica R, Athanassoulis M, Ailamaki A (2010) Aether: a scalable approach to logging. Proc VLDB Endow 3(1–2):681–692
55. Wang T, Johnson R (2014) Scalable logging through emerging non-volatile memory. Proc VLDB Endow 7(10):865–876
56. Helland P, Sammer H, Lyon J, Carr R, Garrett P, Reuter A (1989) Group commit timers and high volume transaction systems. In: High performance transaction systems. Springer, pp 301–329
57. Jung H, Han H, Kang S (2017) Scalable database logging for multicores. PVLDB 11(2):135–148
58. Fang R, Hsiao H-I, He B, Mohan C, Wang Y (2011) High performance database logging using storage class memory. In: ICDE. IEEE, pp 1221–1231
59. Huang J, Schwan K, Qureshi MK (2014) NVRAM-aware logging in transaction systems. Proc VLDB Endow 8(4):389–400
60. NVM (2018) https://en.wikipedia.org/wiki/Non-volatile_memory. Accessed 2018
61. Wu Y, Guo W, Chan C, Tan K (2017) Fast failure recovery for main-memory DBMSs on multicores. In: SIGMOD, pp 267–281
62. Arulraj J, Perron M, Pavlo A (2016) Write-behind logging. Proc VLDB Endow 10(4):337–348
63. Comer D (1979) The ubiquitous b-tree. ACM Comput Surv 11(2):121–137
64. Rao J, Ross KA (2000) Making b+-trees cache conscious in main memory. SIGMOD Rec 29:475–486
65. Rao J, Ross KA (1999) Cache conscious indexing for decision-support in main memory. In: VLDB, pp 78–89
66. Schlegel B, Gemulla R, Lehner W (2009) K-ary search on modern processors. In: Proceedings of the fifth international workshop on data management on new hardware, DaMoN '09. ACM, New York, pp 52–60
67. Kim C, Chhugani J, Satish N, Sedlar E, Nguyen AD, Kaldewey T, Lee VW, Brandt SA, Dubey P (2010) Fast: fast architecture sensitive tree search on modern CPUs and GPUs. In: SIGMOD, ACM, pp 339–350
68. Mao Y, Kohler E, Morris RT (2012) Cache craftiness for fast multicore key-value storage. In: EuroSys. ACM, pp 183–196
69. Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N (2018) The case for learned index structures. In: SIGMOD, pp 489–504
70. Leis V, Kemper A, Neumann T (2013) The adaptive radix tree: artful indexing for main-memory databases. In: ICDE. IEEE, pp 38–49
71. Tree R. https://en.wikipedia.org/wiki/trie
72. Böhm M, Schlegel B, Volk PB, Fischer U, Habich D, Lehner W (2011) Efficient in-memory indexing with generalized prefix trees. In: BTW, Germany, pp 227–246
73. Zhang H, Andersen DG, Pavlo A, Kaminsky M, Ma L, Shen R (2016) Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In: SIGMOD. ACM, pp 1567–1581
74. Wang Z, Pavlo A, Lim H, Leis V, Zhang H, Kaminsky M, Andersen DG (2018) Building a Bw-tree takes more than just buzz words. In: SIGMOD conference. ACM, pp 473–488

75. Binna R, Zangerle E, Pichl M, Specht G, Leis V (2018) HOT: a height optimized Trie index for main-memory database systems. In: SIGMOD, pp 521–534
76. Zhang H, Lim H, Leis V, Andersen DG, Kaminsky M, Keeton K, Pavlo A (2018) Surf: practical range query filtering with fast succinct tries. In: SIGMOD, pp 323–336
77. Jacobson G (1989) Space-efficient static trees and graphs. In: 30th annual symposium on foundations of computer science, Research Triangle Park, NC, USA, 30 Oct–1 Nov 1989, pp 549–554
78. Stoica R, Ailamaki A (2013) Enabling efficient OS paging for main-memory OLTP databases. In: DaMoN, p 7
79. Funke F, Kemper A, Neumann T (2012) Compacting transactional data in hybrid OLTP&OLAP databases. Proc VLDB Endow 5(11):1424–1435
80. Eldawy A, Levandoski J, Larson P-Å (2014) Trekking through siberia: managing cold data in a memory-optimized database. Proc VLDB Endow 7(11):931–942
81. Levandoski JJ, Larson P-Å, Stoica R (2013) Identifying hot and cold data in main-memory databases. In: ICDE. IEEE, pp 26–37
82. Cache replacement. https://en.wikipedia.org/wiki/cache-replacement-policies. Accessed 2018
83. DeBrabant J, Pavlo A, Tu S, Stonebraker M, Zdonik S (2013) Anti-caching: a new approach to database management system architecture. Proc VLDB Endow 6(14):1942–1953
84. Lang H, Mühlbauer T, Funke F, Boncz PA, Neumann T, Kemper A (2016) Data blocks: hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: SIGMOD. ACM, pp 311–326
85. Levandoski JJ, Lomet DB, Sengupta S (2013) The Bw-tree: a b-tree for new hardware platforms. In: ICDE. IEEE, pp 302–313
86. Graefe G (2010) A survey of b-tree locking techniques. TODS 35(3):16
87. Lehman PL et al (1981) Efficient locking for concurrent operations on b-trees. TODS 6(4):650–670
88. Cha SK, Hwang S, Kim K, Kwon K (2001) Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In: VLDB, vol 1, pp 181–190
89. Sewall J, Chhugani J, Kim C, Satish N, Dubey P (2011) PALM: parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. Proc VLDB Endow 4(11):795–806
90. Neumann T (2011) Efficiently compiling efficient query plans for modern hardware. Proc VLDB Endow 4(9):539–550
91. Graefe G (1994) Volcano: an extensible and parallel query evaluation system. IEEE Trans Knowl Data Eng 6(1):120–135
92. Yan C, Cheung A (2016) Leveraging lock contention to improve OLTP application performance. Proc VLDB Endow 9(5):444–455
93. Wang Z, Mu S, Cui Y, Yi H, Chen H, Li J (2016) Scaling multicore databases via constrained parallel execution. In: SIGMOD. ACM, pp 1643–1658
94. Wang Z, Mu S, Cui Y, Yi H, Chen H, Li J (2016) Scaling multicore databases via constrained parallel execution. In: SIGMOD, ACM. New York, NY, pp 1643–1658
95. Mu S, Cui Y, Zhang Y, Lloyd W, Li J (2014) Extracting more concurrency from distributed transactions. In: OSDI, pp 479–494
96. Curino C, Jones E, Zhang Y, Madden S (2010) Schism: a workload-driven approach to database replication and partitioning. Proc VLDB Endow 3(1–2):48–57
97. Pavlo A, Curino C, Zdonik S (2012) Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In: SIGMOD. ACM, pp 61–72
98. Taft R, Mansour E, Serafini M, Duggan J, Elmore AJ, Aboulnaga A, Pavlo A, Stonebraker M (2014) E-store: fine-grained elastic partitioning for distributed transaction processing systems. Proc VLDB Endow 8(3):245–256
99. Two-phase commit protocol. https://en.wikipedia.org/wiki/two-phase-commit-protocol. Accessed 2018
100. Lin Q, Chang P, Chen G, Ooi BC, Tan K-L, Wang Z (2016) Towards a non-2pc transaction management in distributed database systems. In: SIGMOD
101. Ren K, Thomson A, Abadi DJ (2014) An evaluation of the advantages and disadvantages of deterministic database systems. Proc VLDB Endow 7(10):821–832
102. Ailamaki A, DeWitt DJ, Hill MD, Wood DA (1999) DBMSs on a modern processor: where does time go? In: VLDB, pp 266–277
103. Sirin U, Tözün P, Porobic D, Ailamaki A (2016) Micro-architectural analysis of in-memory OLTP. In: SIGMOD, Vol. 215922
104. Tözün P, Gold B, Ailamaki A (2013) OLTP in wonderland: where do cache misses come from in major OLTP components?. In: DaMoN. ACM, p 8

105. Miller JE, Kasture H, Kurian G, Gruenwald C, Beckmann N, Celio C, Eastep J, Agarwal A (2010) Graphite: a distributed parallel simulator for multicores. In: HPCA-16. IEEE, pp 1–12
106. Salomie T-I, Subasu IE, Giceva J, Alonso G (2011) Database engines on multicores, why parallelize when you can distribute? In: EuroSys. ACM, pp 17–30
107. Appuswamy R, Anadiotis A, Porobic D, Iman M, Ailamaki A (2017) Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads. Proc VLDB Endow 11(2):121–134
108. Mühlbauer T, Rödiger W, Reiser A, Kemper A, Neumann T (2013) ScyPer: elastic OLAP throughput on transactional data. In: DanaC. ACM, pp 11–15
109. Lahiri T, Neimat M-A, Folkman S (2013) Oracle TimesTen: an in-memory database for enterprise applications. IEEE Data Eng Bull 36(2):6–13
110. Lindström J, Raatikka V, Ruuth J, Soini P, Vakkila K (2013) IBM solidDB: in-memory database optimized for extreme speed and availability. IEEE Data Eng Bull 36(2):14–20
111. MemSQL Inc., MemSQL. http://www.memsql.com. Accessed 2018
112. Freedman C, Ismert E, Larson P-Å (2014) Compilation in the Microsoft SQL Server Hekaton engine. IEEE Data Eng Bull 37(1):22–30
113. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO. IEEE, pp 75–86
114. Fredkin E (1960) Trie memory. CACM 3(9):490–499
115. Wolski A, Raatikka V (2006) Performance measurement and tuning of hot-standby databases. In: International service availability symposium. Springer, pp 149–161
116. Chan C-Y, Ioannidis YE (1998) Bitmap index design and evaluation. SIGMOD Rec 27:355–366
117. Bernstein PA, Hadzilacos V, Goodman N (1987) Concurrency control and recovery in database systems. Addison-Wesley, Boston
118. Altibase. Altibase administrator's manual release
119. Pavlo A, Angulo G, Arulraj J, Lin H, Lin J, Ma L, Menon P, Mowry TC, Perron M, Quah I, Santurkar S, Tomasic A, Toor S, Aken DV, Wang Z, Wu Y, Xian R, Zhang T (2017) Self-driving database management systems. In: CIDR. www.cidrdb.org
120. Peloton. https://github.com/cmu-db/peloton. Accessed 2018
121. Sikka V, Färber F, Lehner W, Cha SK, Peh T, Bornhövd C (2012) Efficient transaction processing in SAP HANA database: the end of a column store myth. In: SIGMOD. ACM, pp 731–742
122. Wang T, Kimura H (2016) Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. Proc VLDB Endw 10(2):49–60
123. Pavlo A (2017) What are we doing with our lives? Nobody cares about our concurrency control research. In: SIGMOD, p 3
124. Zhu T, Wang D, Hu H, Qian W, Wang X, Zhou A (2018) Interactive transaction processing for in-memory database system. In: DASFAA, part II, pp 228–246
125. Gray J et al (1996) The dangers of replication and a solution. SIGMOD Rec 25(2):173–182
126. Decandia G et al (2007) Dynamo: Amazon's highly available key-value store. In: SOSP
127. Cassandra website. http://cassandra.apache.org/. Accessed 2018
128. Bailis P, Davidson A, Fekete A et al (2013) Highly available transactions: virtues and limitations. Proc VLDB Endow 7(3):181–192
129. Lamport L (1998) The part-time parliament. TOCS 16(2):133–169
130. Lamport L (2001) Paxos made simple. ACM SIGACT News 32(4):18–25
131. Baker J, Bond C, Corbett JC et al (2011) Megastore: providing scalable, highly available storage for interactive services. In: CIDR, pp 223–234
132. Corbett JC, Jeffrey D et al (2013) Spanner: Googles globally distributed database. TOCS 31(3):8
133. Rao J, Shekita EJ, Tata S (2011) Using paxos to build a scalable, consistent, and highly available datastore. In: VLDB, pp 243–254

Huiqi Hu is currently an assistant professor in the School of Data Science and Engineering, East China Normal University. He received his Ph.D. Degree from Tsinghua University. His research interests mainly include database system theory and implementation, scalable distributed storage system.

Xuan Zhou is a professor and a vice-dean at the School of Data Science and Engineering, East China Normal University (ECNU). He obtained his B.Sc from Fudan University (China) and his Ph.D from the National University of Singapore, both in Computer Science. Since his graduation in 2005, he had worked as a scientist at the L3S Research Centre (Germany) and the CSIRO ICT Centre (Australia) until the end of 2010. Before he joined ECNU in 2017, he spent 6 years in Renmin University of China, as an associate professor. Xuan's research interests include database system and information retrieval.

Tao Zhu is a Ph.D. student in the School of Data Science and Engineering, East China Normal University, China. His research interests mainly include database system implementation, transaction processing and distributed system.

Weining Qian is currently a professor in computer science at East China Normal University, China. He received his M.S. and Ph.D. in computer science from Fudan University, China in 2001 and 2004, respectively. He served as the co-chair of WISE 2012 Challenge, and program committee member of several international conferences, including ICDE 2009/2010/2012 and KDD 2013. His research interests include Web data management and mining of massive data sets.

Aoying Zhou is a professor on computer science at East China Normal University, China where he is heading the Institute for Data Science and Engineering. He got his master and bachelor degree in computer science from Sichuan University, China in 1988 and 1985, respectively, and won his Ph.D. degree from Fudan University, China in 1993. He is now acting as the vice-director of ACM SIGMOD China and Technology Committee on Database of China Computer Federation. He is serving as a member of the editorial boards of some prestigious academic journals, such as VLDB Journal, and WWW Journal. His research interests include Web data management, data management for data-intensive computing, and in-memory data analytics.