On the Design and Scalability of Distributed Shared-Memory Databases

Author(s):
Loesing, Simon; Pilman, Markus; Etter, Thomas; Kossmann, Donald

Publication Date:
2013-12

Permanent Link:
https://doi.org/10.3929/ethz-a-010034028

Technical Report

Systems Group, Department of Computer Science, ETH Zurich

On the Design and Scalability of Distributed Shared-Memory Databases

by

Simon Loesing
Markus Pilman
Thomas Etter
Donald Kossmann

December 2, 2013

Abstract

To scale-out databases today, partitioning data across several database instances is the common technique. This approach is typically referred to as shared-nothing architecture. In this paper, we propose a new alternative architecture design for distributed relational databases based on two fundamental principles: We logically decouple query processing and transaction management from data storage and we share data across all query processing nodes. The combination of these design choices provides operational flexibility, a property that enables cost-efficient database infrastructures and that becomes increasingly important with the growing market penetration of cloud computing. As a drawback, sharing data among several database nodes adds substantial synchronization overhead. We present techniques for efficient data access and concurrency control to counter this effect. These techniques are enabled by recent hardware trends such as in-memory storage and low-latency networking and provide scalability characteristics that top state-of-the-art shared-nothing databases.

1 Introduction

Modern Web 2.0 applications have requirements that go beyond what traditional relational databases management systems (RDBMSs) can provide. For instance, RDBMSs are inflexible with regard to evolving data and do not support popular data interchange formats such as JSON. In response to these limitations, new design considerations have triggered the emergence of NoSQL (not only SQL) storage systems. NoSQL systems are hyped for their high scalability and availability. However, the fundamental design premise behind the NoSQL phenomenon is operational flexibility [29]. Operational flexibility describes all features that either facilitate and simplify user interaction, or that enable the system to dynamically react to changing conditions. This includes elasticity, the ability to grow or shrink the system on-demand; ease-of-use, the ability to efficiently write and execute any kind of query; and deployment flexibility, the ability to run out-of-the-box on many commodity hardware servers. These properties have become critical success factors with the advent of cloud computing and *big data*.

In this paper, we propose a new architecture for distributed transactional processing that is specifically designed towards operational flexibility. Our goal is to keep the strengths of RDBMSs, namely SQL and ACID transactions, and at the same time provide the benefits that inspired the NoSQL movement. Our architecture is based on two principles.

The first principle is data sharing between all database instances. Data is located in a shared record store. In contrast to *shared-nothing (SN) architectures* [37], database instances do not own a dedicated partition, but have a global view on all data and can execute any query. Consequently, distributed transactions are no longer necessary, a fundamental benefit compared to the shared-nothing approach.

The second principle that boosts operational flexibility, is decoupling query processing and transaction management from data storage. While in traditional RDBMS engines the two are tightly coupled, we logically separate them into two autonomous architectural layers. This break-up enables elasticity and deployment flexibility as both layers can be scaled-out (or scaled-down) independently. Moreover, it allows for highly parallelized query execution that for instance enables scalable online analytical processing (OLAP). Even more intriguing is the possibility to perform online transaction processing (OLTP) and run OLAP queries on separate instances but accessing the same data. This mixed workload scenario enables use-cases not possible in a SN architecture. For example, it allows business intelligence analysis on live data without any impact on production workload.

Sharing data raises several technical challenges we address in this paper. First, shared data access requires synchronization [38]. A recent approach for efficient and lightweight synchronization is based on atomic read-modify-write (RMW) primitives [36]. We use atomic RMW as a key operation to implement multi-version concurrency control (MVCC) [4]. Second, data access involves network communication. In order to keep latencies as low as possible we follow new technical trends and use fast networking technology (i.e., Infiniband [17]) as well as in-memory data storage. In particular, maintaining data entirely in main memory avoids the overhead of disk-oriented storage [39] and allows for low-latency data access. These technologies enable scalability not possible a few years ago.

Database architectures that share data are typically referred to as *shared-disk architectures*. However, as we keep data entirely in main memory, the name *shared-memory architecture* is more appropriate. Traditionally, shared-memory architectures refer to multiprocessor hardware where several processors are connected to a single shared main memory. However, in this paper, we extend the term to describe main memory shared across multiple machines. Note that in reality not all available memory is shared. Query processing nodes interact with an in-memory record store but still operate with their own dedicated memory.

This paper makes three contributions. First, we propose a new shared-memory database architecture designed to meet the demands of modern applications. That is, scalability, elasticity and fault-tolerance. Second, we describe novel techniques to perform distributed MVCC and efficient data access on shared data. We use these techniques in a prototype implementation that performs query processing and transaction management on top of an in-memory record store. Third, we evaluate the effectiveness of our design and compare it to traditional shared-nothing databases.

The paper is organized as follows: Section 2 details the key design principles and technical challenges of the shared-memory architecture. We review related work in Section 3. Section 4 describes a technique to implement concurrency control and illustrates recovery, while Section 5 explains how data is stored and accessed. Section 6 then presents an experimental evaluation of our architecture, and Section 7 concludes the work presented in this paper.

2 A Shared-Memory Database Architecture

This section details the design principles and technical challenges of the shared-memory architecture.

2.1 Design Principles

In the following we will describe key design principles to enable scalability and elasticity. Specifically, we detail the architectural principles motivated in the previous section. Moreover, we highlight the necessity of ACID transactions and complex queries, two fundamental features that have recently been relaxed or simplified in many storage systems.

Shared Data: Shared data implies that every database instance can access and modify all data stored in the system. There is no exclusive data ownership. An entire transaction can be executed and committed/aborted by a single instance. In contrast to SN architectures no application knowledge and expert skills are required in order to correctly partition data to avoid distributed

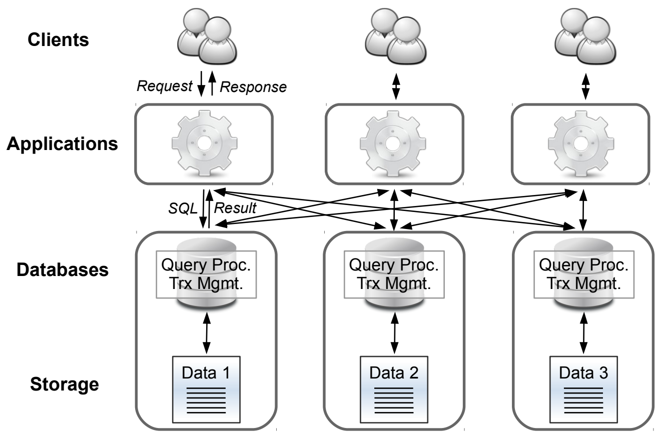

Figure 1: Shared-Nothing Architecture

transactions. Consequently, setting up a running database cluster becomes a lot easier. Additionally, the application's interaction with the database is simplified as partitioning is no longer reflected in the application logic. On the other hand, sharing data requires updates to be synchronized, a constraint we will address in Section 2.2.

Decoupling of Query Processing and Storage: The shared-memory architecture is decomposed into two logically independent layers, transactional query processing and data storage. As the storage layer is independent it can be implemented as a self-contained system that manages data distribution and fault-tolerance transparently with regard to the processing layer. Hence, replication and data relocation tasks are executed in the background without the processing layer being involved. The storage system is in essence a distributed record manager that consists of multiple *storage nodes* onto which data is distributed. Data partitioning is not as performance critical as in SN databases as data location does not determine where queries have to be executed. Instead, to execute queries, the processing layer will fetch records independent of the storage nodes they are located at. This fundamental difference in the communication pattern is highlighted in Figures 1 and 2. Provided that the load on the storage system is evenly balanced, the storage layer scales with the number of storage nodes.

The processing layer consists of multiple autonomous *processing nodes* (not interacting with each other) that access the shared storage system. A mechanism is provided to retrieve data location (e.g., a lookup service) which enables the processing nodes to directly contact the storage node holding the requested data.

Logical decoupling considerably improves agility and elasticity as processing nodes or storage nodes can be added on-demand if processing resources or storage capacity is required respectively. Accordingly, nodes can also be removed

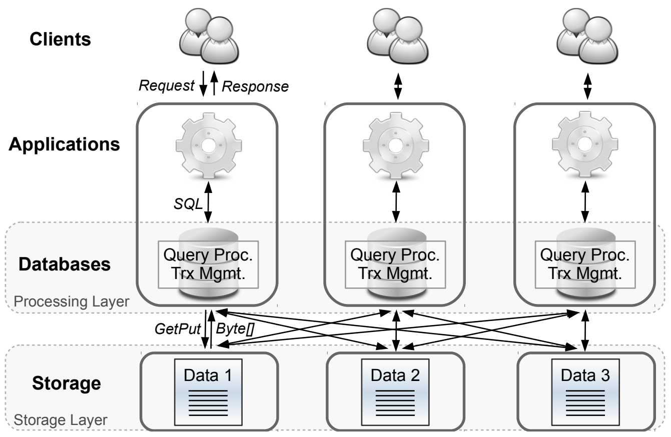

Figure 2: Shared-Memory Architecture

if capacity is not required anymore. At the same time this architecture enables workload flexibility: It is possible to execute different workloads on separate processing nodes. For example, a set of processing nodes can execute an OLTP workload while others simultaneously perform analytical
queries on the same dataset.

In-Memory Storage: Commodity hardware servers available today are equipped with dozens or even hundreds of GBs of main memory. Hence, even in small storage clusters the total memory capacity adds up to TBs. These numbers surpass the storage capacity requirements of most existing applications. For instance, in the popular TPC-C benchmark [40], the capacity required for a warehouse with thousands of items is less than 200 MB. Consequently, it is no longer necessary to rely on slower storage devices (e.g., hard drives) and the entire application data can be kept in main memory. The advantages are obvious, memory provides much lower access latencies and avoids complex buffering mechanisms [39]. In a shared-memory architecture in which processing is decoupled from storage, in-memory storage provides a foundation for scalable transactional processing. DRAM memory is volatile but data loss in case of failures can be prevented using replication. Fault-tolerance (see Section 4.4) and the ability to handle node failures is considered an evident property of any storage system.

ACID Transactions: Transactions ensure isolated execution of concurrent operations and maintain data integrity. From the developer's perspective, transactions are a convenient way to perform consistent data changes and simplify application development. In particular, development is less error-prone as data corruption and anomalies do not occur.

The shared-memory architecture presented in this paper provides full transactional support. That is, ACID transactions can be performed without limitations on all data contained in the database. Transaction management is part of the processing layer and does not make any assumption on how data is accessed and stored. How to achieve scalability while at the same time being highly consistent is one of the challenges addressed in the next subsection.

Complex Queries: SQL is the standardized query language in relational databases enabling complex queries and including features to order, aggregate, or filter records based on predicates. Although alternative storage systems often simplify the query model to improve performance, complex (SQL) queries are not an obstacle to system scalability [34].

In the shared-memory architecture, data location is logically separated from query processing and as a result a processing node will fetch all data it requires to execute a query. This notion can be described as *data is shipped to the query*, and not *the query to the data*. The agility to run the same query on multiple nodes provides increased parallelism and enables scalability assuming that data access is fast enough.

2.2 Technical Challenges

Although shared-memory architectures avoid the limitations of their shared-nothing counterpart, namely sharding and missing agility, they also have constraints which we explain in the following.

Data Access: In a shared-data environment data is likely to be located at a remote location and has to be accessed or modified over the network. Caching data in local buffers is only possible to a limited extent as updates made by one processing node have to be visible to the other nodes instantly. As a result, to provide consistent access, most requests need to fetch the latest record version remotely from the storage layer. Considering that processing a query usually requires multiple records, data access latencies can quickly become a dominant factor in query execution time.

In light of restricted buffering possibilities, traditional techniques to manage relational data have to be reevaluated. A major concern in this context is the correct granularity of data storage. That is, whether it is beneficial to group records into pages in order to retrieve or store several records with one request, or whether choosing a finer storage granularity (e.g., single record) is a more favorable approach. In the shared-memory architecture, we suggest to store data at the granularity of a record as this provides a good trade-off between the number of network messages and the amount of network traffic generated. We will justify this substantial design choice and detail its implications in Section 5.

A second challenge of shared data relates to data access paths and indexing. Analogous to data, indexes are shared across multiple processing nodes and are therefore subject to concurrent modifications from distributed locations. As a result, the shared-memory architecture requires distributed indexes that support atomic operations but are at the same time highly scalable. As a solution, we propose a scalable latch-free distributed B+Tree index in Section 5.2.

In addition to the right techniques, state-of-the-art networking technologies such as 10 Gb Ethernet or Infiniband can contribute to reduce the overhead of remote data access. These technologies provide latency and bandwidth not possible a decade back and consequently enable our proposed shared-memory architecture to scale to new levels. For instance, Infiniband allows Remote Direct Memory Access (RDMA) in a few microseconds and is three orders of magnitude faster than a random read on a local hard-disk.

Concurrency Control: Shared data records can be updated by any processing node and therefore concurrency control is required across all processing nodes. Although common pessimistic and optimistic concurrency control mechanisms [5] can be used in a shared-memory architecture, distribution requires the mechanisms to be examined from a new angle. For instance, a lock-based approach requires centralized lock management. Assuming network latency dominates lock acquisition time, the lock manager quickly becomes a bottleneck that limits transaction throughput. Similarly, increased data update latencies might cause more write-conflicts in an optimistic protocol resulting in higher abort rates. Accordingly, a concurrency control mechanism that minimizes the overhead of distribution will allow for the highest scalability.

In this paper we present a distributed MVCC protocol. A key feature is conflict detection using atomic RMW primitives. Atomic RMW operations have become popular in recent years as they enable non-blocking synchronization. In the context of the shared-memory architecture, they allow to perform data updates only if a record has not been changed since it has last been read. As a result, conflicts can be identified with a single call. Atomic RMW primitives are a lightweight mechanism that allows for efficient distributed concurrency control. The details of this technique are presented in Section 4.

2.3 Limitations

The architecture and the techniques presented in this work perform best in the context of local area networks (LAN). LANs allow low-latency communication which is a critical performance factor for several reasons. First, shared data implies limited buffering and causes heavy data access over the network. Second, in-memory data requires synchronous replication in order to prevent data loss in case of failures. Wide area networks (e.g., between data-centers) have substantially higher communication cost than LANs and are therefore unsuited for these requirements.

Network bandwidth is another constraint that can potentially become a bottleneck. The processing and the storage layer constantly exchange data and factors such as heavy load or large record sizes can cause network saturation.

Some of the techniques we present in this paper introduce additional constraints which are detailed at a later point.

3 Related Work

Multi-server databases that share data have been subject to various research over the years. Oracle RAC [8], IBM DB2 Data Sharing [19] and Oracle Rdb [26] are based on a shared-disk architecture. These systems use a distributed global lock-manager to synchronize data access and ensure concurrency control. Data is stored on disk and the granularity of sharing data is a page. In contrast, our approach is based on in-memory storage and reduces data access granularity to single records. A more fundamental difference lies in the database design. In traditional DBMS engines transaction management and data storage is typically tightly coupled. Our architecture separates both components into independent architectural layers allowing for elasticity and more deployment flexibility without sacrificing performance. The advantages of decoupling transaction management and storage have been highlighted in previous work [24, 27]. ScaleDB [35] is a recent shared-disk database that implements this principle. However, no implementation details and performance results have been published so far. A record manager that has many similarities with our approach is Hyder [6]. Hyder uses a shared-data architecture to provide transactional access to indexed records stored in flash. Transaction management is logically decoupled from storage and is performed in a software layer that can easily be extended with a rich query interface.

Shared-nothing databases have been explored since the 1980s and are nowadays the predominant architecture for parallel databases. A thorough review of the architectural benefits and the techniques commonly used is provided by Dewitt and Gray [13]. MySQL Cluster [30] is one example of a state-of-the-art SN database that horizontally partitions tables across database nodes. An alternative is VoltDB [41] which is based on the design of H-Store [20]. To minimize distributed transactions, VoltDB can automatically identify an optimal partitioning scheme based on predefined queries. Microsoft Windows SQL Azure [7] is a cloud service that provides managed database instances on demand. The service includes a *federation* feature that allows to partition data across several database instances. However, join and aggregation operations across partitions are not supported and have to be handled at the application layer.

Main memory databases [14, 22] take advantage of memory performance to optimize query processing in the scope of a single machine. They can very efficiently process OLTP workloads, but typically struggle in scaling analytical queries. A problem the shared-memory architecture overcomes. Additionally, the techniques used are mostly not applicable to a distributed system because of higher latencies.

Recently, relational databases have been challenged by the emergence of NoSQL storage systems which typically relax consistency guarantees in favor of more scalability and availability. In contrast, our approach aims at achieving these properties while being highly consistent and providing ACID transactions. NoSQL systems, in particular key-value stores are mostly based on an SN architecture and partition data across nodes. Amazon's Dynamo [12] was one of the first scalable key-value stores that uses eventual consistency to provide

	Shared Data	Decoupling	In-Memory	ACID Txs	Rich Query
Shared-Memory	✓	✓	✓	✓	✓
Oracle RAC	✓	-	-	✓	✓
ScaleDB	✓	✓	-	✓	✓
Hyder	✓	✓	-	✓	(✓)
VoltDB	-	-	✓	✓	✓
Azure SQL	-	-	-	✓	✓
Google BigTable	-	-	-	-	-
Google Megastore	✓	✓	-	✓	-

Table 1: Comparison of Selected Databases and Storage Systems

high availability. Other systems, such as Google's BigTable [9] or Spinnacker [33] provide strong consistency and atomic single-key operations but no ACID transactions. G-Store [11] does support ACID transactions for user-defined groups of data objects. The major restriction is that these groups must not overlap. ElasTras [10], another key-value store, limits the scope of transactions to single partitions. Two systems do not have this limitation: Sinfonia [1] and MegaStore [2]. Sinfonia is a SN system that uses two-phase commit (2PC) to reach agreement on transaction commit. Megastore implements transaction semantics on top of BigTable. The system is designed to replicate data across data-centers and uses the Paxos protocol [21] to reach consensus on transaction execution order. However, data is partitioned into groups of objects called *entity group* and transaction execution is only efficient inside and not across entity groups. None of the presented systems provides a query interface that can compare with SQL. However, many strongly consistent NoSQL stores are equivalent to atomic record stores and support basic data operations as required by the storage layer in our architecture.

Table 1 provides an overview of the discussed systems with regard to our design principles. Although the principles have already been implemented in many systems, the shared-memory architecture is, to the best of our knowledge, the first database system to provide all properties together.

4 Transaction Processing and Concurrency Control

Database concurrency control ensures that transactions can run in parallel on multiple processing nodes without violating data integrity. In this section we describe a MVCC protocol. More specifically, we detail a distributed variant of *snapshot isolation* (SI) [3] that uses atomic RMW primitives for conflict detection. The SI protocol is part of the transaction management component on each processing node and guarantees ACID properties. To provide SI semantics across processing nodes, each node interacts with a dedicated authority, the commit manager (Figure 3). The commit manager keeps track of all running transactions and manages snapshot information. Evaluating our design using lock-based protocols will be subject to future work.

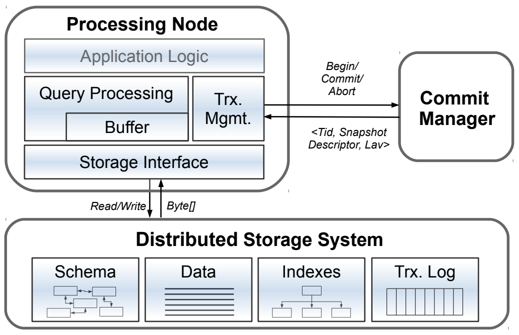

Figure 3: Architecture Components

4.1 Distributed Snapshot Isolation

MVCC stores multiple versions of every data item. Each time a transaction updates an item, a new version of that item is created. When a transaction starts, it retrieves a list with all data item versions it is allowed to access. In SI, a transaction is only allowed to access the versions that were written by already finished transactions (and never the ones written by transactions that are still running). This is the so-called *consistent snapshot* the transaction operates with.

In SI, transactions perform operations on their snapshot isolated from other transactions and eventually check for conflicts at commit time. It follows that transactions temporarily store data updates (insert, update, and delete operations) in a local buffer and apply them to the shared data store during commit. The commit of a transaction T_1 will only be successful if none of the items in the write set have been changed externally by another transaction T_2 that is committing or has committed since T_1 has started. If T_1 and T_2 run in parallel and modify the same item, two scenarios can happen. First, T_2 writes the changed item to the shared store before it is read by T_1. In that case, T_1 will notice the conflict (as the item has a newer version). Second, T_1 reads the item before it has been written by T_2. If this happens, T_1 must be able to detect the conflict before it writes the item. For this purpose, we use atomic RMW operations such as *load-link and store-conditional* (LL/SC) [18]. LL/SC is a pair of instructions that reads a value (load-link) and allows to update the latter atomically only if it has not changed in the meantime (store-conditional). LL/SC is stronger than *compare-and-swap* as it solves the ABA-Problem [28].

LL/SC is the key to conflict detection. If all updates of transaction T_1 can be applied successfully, there are no conflicts and the transaction can commit. If one of the LL/SC operations fails, the respective data item has been modified by transaction T_2 and there is a write-write conflict. If a conflict is detected, T_1 will abort and revert all the changes made to the data store. The transaction that was able to successfully execute all its updates wins and commits (in our example T_2). Conflict detection at commit time is usually referred to as *First Committer Wins* strategy.

SI provides an isolation level that avoids many of the common concurrency control anomalies and is sufficient for the vast majority of use cases. This isolation level is ensured as long as all transactions execute their updates atomically using LL/SC operations no matter on which processing node they are running. Some anomalies (e.g., write skew) prevent SI to guarantee serializability in all cases [15]. Solutions for serializable SI have been proposed [32] and we mean to extend our protocol to provide serializability in the near future.

4.2 Life-cycle of a Transaction

We explained conflict detection in the previous section, in this section we complete the illustration of the SI protocol by describing the life-cycle of a transaction. First, we enumerate the different transaction states:

1. **Begin:** In an initial step a transaction contacts the commit manager to retrieve a unique transaction id (*tid*) and a snapshot descriptor. The latter is a set of versions that specify the transaction's snapshot. Once this step is completed, the transaction is started. The commit manager is described in Section 4.3.

2. **Running:** While running, a transaction can perform read, insert, update, and delete operations on any data item. Read operations retrieve the valid version from the store and put it in the transaction buffer in case the item is re-accessed by the transaction. Update operations are temporarily stored. Evidently, a read operation on an updated item will return the new value. Data storage and buffering are detailed in Section 5.

3. **Try-Commit:** Before starting the commit procedure a *write set* containing the ids of updated items is written to the transaction log. This is required for fault-tolerance (see Section 4.4). Next, we proceed with applying updates to the storage system as described in the previous section. On success, the transaction can commit. Otherwise we abort.

4. (a) **Commit:** All data updates have been executed. Next, the indexes are altered to reflect the updates and a commit flag is written to the transaction log. Finally, the commit manager is notified.

   (b) **Abort:** During abort, data updates already applied to the shared store are removed. Of course, a transaction can also be directly aborted by the application. In this case, no updates have been executed (as we skipped the *Try-Commit* state) and therefore no rollback is necessary. Last, the commit manager is notified.

4.3 Commit Manager

The commit manager is a key component for concurrency control that keeps track of completed transactions. Based on that information, starting transactions can retrieve a valid snapshot. In addition to snapshot management, the commit manager assigns system-wide unique *transaction ids (tids)* to identify transactions and computes the *lowest active version number (lav)* used for garbage collection.

Transaction ids are automatically incremented integer values that uniquely identify a transaction. Every running transaction requires a *tid* before executing data operations. Given its system-wide uniqueness, the *tid* is not only used to identify a transaction, but also defines the version number for updated data items. In other words, a transaction will create new versions for updated data items with the *tid* as version number. Due to the fact that *tids* are always incremented, version numbers will increase accordingly. Hence, as *tids* and version numbers are synonyms, the set of *tids* of completed transactions also defines the snapshot for newly starting transactions.

The snapshot descriptor is a data structure that specifies which versions a transaction can access. It consists of two elements: A *base version number* b, that is the highest *tid* until which all transactions have completed and a set of newly committed *tids* N. As the name indicates, N contains all committed transactions with $tid > b$ but not $b+1$. When $b+1$ commits, the base version is incremented until the next non-committed *tid* is reached. The implementation of the snapshot descriptor is simple and involves low computational cost. b is an integer and N is a bitset. Starting with b at offset 0, each consecutive bit in N represents the next higher *tid* and if set indicates a committed transaction. As a result, the snapshot descriptor is small even in the presence of many parallel transactions. For example, $N \leq 13$ KB with 100,000 newly committed transactions.

When accessing a data item with a version number set V, the transaction will read the version with the highest version number v matching the snapshot descriptor. More formally, we define the set of valid versions numbers (or version number set) the transaction can access V' as:

$$V' := \{\ x \mid x \leq b \vee x \in N\ \}$$

The version with number v is accessed:

$$v := max(\ V \cap V'\ )$$

The lowest active version number is equal to the lowest base version number among all running transactions. That is, the highest version number globally visible to all transactions. Version numbers smaller than the *lav* are possible candidates for garbage collection (see Section 5.3).

In order to communicate with the processing nodes, the commit manager provides a lightweight interface. The following three functions are supported:

- $start() \rightarrow (tid, snapshot\_descriptor, lav)$: Signal the start of a new transaction. Returns a newly generated *tid*, a snapshot of the current state, and the *lav*.

- *setCommitted*(*tid*) $\rightarrow$ *void*: Signals that the transaction *tid* has successfully committed. This internally updates the list of committed transactions used to generate the snapshot descriptors.
- *setAborted*(*tid*) $\rightarrow$ *void*: Same as above but for aborted transactions.

Although these operations can be efficiently processed with low computational cost, the amount of transactions a single commit manager can process is limited. In order to scale, several commit managers can run in parallel. To guarantee correctness the commit managers have to synchronize on two components. First, the uniqueness of every assigned *tid* has to be ensured. This can be implemented using an atomically incremented counter managed in the shared store. We assume that the storage system provides an atomic increment operation. Every commit manager can use the incremented counter value as a new system-wide unique *tid*. To prevent synchronization from becoming a performance problem, commit managers can increase the counter by a high value (e.g., 100) to get a range of *tids* they can assign to new transactions. Second, the commit managers have to exchange their most recent snapshots (i.e., the list of committed transactions). We again use the shared store for synchronization. In short intervals (e.g., 5 ms) every commit manager writes its current snapshot to the shared store and thereafter reads the latest snapshots of all other commit managers. As a result, every commit manager gets a global view that is at most delayed by a few milliseconds. This is not an issue as it is legitimate for transactions to operate on a delayed snapshot. Nevertheless, the older the snapshot, the higher the probability of conflicts and the higher the abort rate.

4.4 Fault-Tolerance and Recovery

This section details the implications of node failures and explains the recovery process to ensure data integrity. Node failures can happen any time and affect every component.

4.4.1 Failure of a processing node

Processing nodes act according to the crash-stop model. That is, in case of transient or permanent failures all active transactions on that node are considered failed and have to be aborted. Failures are noticed by an eventually perfect failure detector based on timeouts. If a failure is detected a recovery process is started to roll back all active transactions of the failed node and revert the changes made in the storage system. Correct recovery is enabled by a transaction log that contains the status of running transactions.

The transaction log is stored in the storage system. It is thus not a file as in traditional databases but it is implemented as an ordered map of log entries. Every entry maps a *tid* to the state of the respective transaction. Before applying any data updates to the storage system, a transaction writes a *write set* to the log. The write set contains the processing node id, a timestamp as well as a list of ids of items to be updated. In the final stage of the commit process, the write set is overwritten by a flag (committed or aborted) tagging the transaction as completed.

Once started, the recovery process first discovers the active transactions of the failed node. This involves retrieving the highest *tid* from the commit manager and iterating backwards over the transaction log domain until the lowest active version number is reached. The *lav* implicitly acts as a rolling checkpoint. Once we encounter a relevant transaction, we use its write set to iterate over the items and revert potential changes made by the transaction.

After a failure, connected clients lose their connection and can safely assume that all running transactions have been or are currently being aborted. The failed transactions can be re-executed once the connection to another running or a newly started processing node has been established.

4.4.2 Failure of a storage node

The storage system must be able to handle node failures transparently with regard to the processing nodes. A failure must not interrupt operation and not lead to data loss. Availability is guaranteed by replicating data. As data is kept in volatile storage (main memory) a node must ensure that data is replicated to a replica before a request can be acknowledged. This implies synchronous replication regardless of the replication protocol used (Read-One/Write-All, majority quorum, etc.). If a node fails, the storage system fails-over to the replicas and ensures on-going processing of requests. Eventually, the storage cluster reorganizes in order to restore the specified replication level.

In Section 6 we present a prototype implementation that uses a management node to detect failures. The same node manages partitioning, restores the replication factor and enables the processing nodes to look-up the location of replicas. To prevent a single point of failure, several management nodes with a synchronized state are required.

4.4.3 Failure of a commit manager

The failure of the commit manager has system-wide impact as processing nodes are no longer able to start new transactions. Hence, starting a new commit manager is a requirement to continue transaction processing. The active transactions at the moment of failure can still commit as the commit manager is not required for completion.

To prevent the commit manager from being a single point of failure, multiple commit managers can run in a cluster. We assume that all commit managers are known to the processing nodes. As pointed out previously, state information is synchronized and thus, if a processing node loses contact with a commit manager it will switch to another one.

A new commit manager can restore its state by retrieving the last used *tid* and the most recently committed transactions from the transaction log. This is not necessary in a multi-commit manager configuration as the servers will synchronize their state automatically.

5 Data Access and Storage

Processing nodes provide a SQL interface to applications and thus enable complex queries on relational data. The query processor parses and optimizes incoming queries and employs the iterator model to access records. Records are fetched and modified using basic data operations such as *read* and *write* and are stored using the key-value data model. The latter maps an identifier to an arbitrary binary value and is therefore well-suited to store relational records. This section explains how relational data is mapped to the key-value model and details data access methods.

5.1 Data Mapping

The mapping of relational data to the key-value model is straightforward: Every relational record (or row) is stored as exactly one key-value pair. The key is a unique *record identifier (rid)* which is by default an atomically incremented numerical value. The value field contains a serialized set of all versions of the record. This row-level storage scheme is a significant design decision as it minimizes the number of storage accesses. For instance, a single read operation can retrieve a record with all its versions. A transaction running on a processing node can thereafter pick the correct version according to its snapshot descriptor. On update, a transaction adds a new version to the record and writes back the entire record during commit. Again, a single write request applies the update or identifies a conflict (if the update is rejected). If supported by the storage system, a separate domain can be created for every relational table.

Grouping records into pages, like disk-oriented databases do in order to reduce the number of I/O operations is of limited use in a shared-memory architecture. In particular, every record can be remotely changed and therefore needs to be re-fetched on access. Thus, a more coarse-grained storage scheme will not reduce the number of requests to the storage system but only increase the amount of network traffic. In contrast, a more fine-grained storage scheme (e.g., store every version of a record as a key-value pair) will require additional requests to identify new versions. In such a system, a committing transaction will have to retrieve new versions of every updated item and as a consequence conflict detection becomes expensive. Although this approach will reduce network traffic as the unit of transfer is smaller, we prioritize for performance reasons an approach that minimizes the number of network requests.

To access data, a processing node can look up records by using an index structure. Every index entry contains a reference to the *rid* of the respective record. Using that *rid*, a processing node can fetch the record data containing all available versions of that specific tuple (Figure 4). To update a record, it first has to be read by the transaction. Next, a new version of the tuple reflecting the changes of the update operation will be added to the set of versions of the record. The entire record is kept in the transaction buffer and further updates to the record can directly modify the newly added version. As explained previously, the record is written to the storage system at commit time.

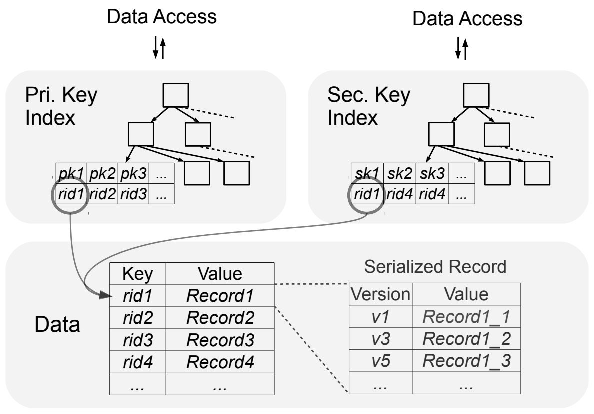

Figure 4: Storage and Access of Data Records

5.2 Latch-free Index Structures

Indexes are a fundamental data structure to access data. An index is built for a relational attribute (or a set of attributes) and provides a lookup mechanism to find matching records for a given attribute key. The supported operations are lookup, insert, delete, and update of index entries. As indexes are usually accessed and modified by multiple transactions they need concurrency control. While traditionally index integrity is maintained using latches, more recently latch-free algorithms gained popularity by delivering more throughput in the presence of many cores.

In a shared-memory architecture indexes can be modified by multiple processing nodes at the same time. Hence, the integrity of an index must be maintained across multiple machines. In a distributed context the absence of latches becomes even more advantageous compared to traditional latch-based approaches as network communication increases latch acquisition cost and there is guaranteed system-wide progress. Index synchronization is analogous to transaction concurrency control, the indexes are entirely stored in the storage system and atomic RMW operations are used to apply changes.

5.2.1 Design Considerations

An important performance consideration and optimization goal is to minimize the number of requests to the storage system, and thus the number of network messages to execute index operations. Considering this objective, we decided not to include versioning information in the index. Instead of maintaining a key-reference entry for every existing version, there is only a single index entry per record. As a result, we do not need to insert a new index entry on every record update (i.e., whenever a new version is added), but solely when the indexed key of a record is changed. Consequently, less index modifications significantly improve concurrency.

The disadvantage of this approach and the reason to include versioning information in the index is that it is no longer possible to identify for which version an index entry is valid. As this information is absent, it may happen that on lookup a transaction finds an index entry and retrieves a record that is not valid according to its snapshot descriptor. However, as these additional read operations are performed independently of the index, they do not affect concurrency. Of course, non-valid records are ignored by the transaction. Furthermore, index-only scans are not possible as the entry's validity can not be verified. Altogether, the limitations of version unaware indexes are an acceptable trade-off given the improved concurrency. Based on these principles, we describe a latch-free distributed B+tree.

5.2.2 B+Tree Index

The B+tree is the most used index structure in databases. Over the last decades, many variations have been optimized for concurrent access [23, 16]. Particularly noteworthy is the Bw-tree [25], a latch-free variant for modern hardware. Using the techniques of the Bw-tree we implemented a latch-free B+tree concurrently accessible by multiple nodes.

The index is entirely stored in the storage system. On lookup, a transaction accesses the root node and traverses the tree until the leaf-level is reached. To minimize storage system requests, large parts of the tree can be cached on the processing nodes. Assuming all tree nodes have been previously read and are in the cache, we only need to verify on traversal if the leaf node has changed. If so, we check backwards the higher levels of the tree for changes and retrieve all changed nodes. In a best case scenario, we thus only require one request per lookup operation. The tree is ordered, consequently point-lookups as well as range queries are supported.

5.3 Garbage Collection

Every update adds a new version to a record and over time records become increasingly larger. To prevent data items from growing infinitely, garbage collection (GC) is necessary. Garbage collection deletes the data versions and index entries that will never be accessed again. We apply two garbage collection strategies. The first one is *eager* and cleans up data records and indexes as part of an update or read operation respectively. The second strategy is *lazy* and executes GC as a background task that runs in regular intervals (i.e., one or several times a day). The latter approach is necessary for rarely accessed or modified records.

Record GC is part of the update process. Before a transaction writes back an updated record, it first verifies if some older versions can be removed. To determine the evictable versions, a transaction uses the lowest active version number it has obtained from the commit manager on transaction start. Given a record with version numbers V and a lowest active version lav, we define C

as the set of version numbers of a record visible to all transactions:

$$C := \{ \ x \mid x \in V \wedge x \leq lav \ \}$$

The set of garbage collectable version numbers G is:

$$G := \{ \ x \mid x \in C \wedge x \neq max(C) \ \}$$

The version with number $max(C)$ is never garbage collected to guarantee that at least one version of the item always remains. All versions which number is in G can be safely deleted before writing the record back to the storage system.

Removed record versions can be referenced by index entries that are also subject to garbage collection. In contrast to garbage collection of records, index GC is performed during read operations. When a transaction executes an index lookup and afterwards reads all matching records, it verifies if some of the index entries can be removed. Given an index entry k with key a, we define V_a as the version number set of all versions that contain a with $V_a \subseteq V$. k can be removed from the index if the following condition holds:

$$V_a \backslash G = \varnothing$$

Index nodes are consistently updated. If the atomic operation fails, processing is continued and GC is retried during the next read operation.

5.4 Buffering Strategies

In this section, we present three approaches to buffering in the shared-memory architecture. The effectiveness and performance implications of each strategy are illustrated in Section 6.6.

5.4.1 Transaction Buffer

Transactions operate on a particular snapshot and do not necessarily require the latest version of a record. Accordingly, data records, once read by a transaction, can be buffered and reused in the scope of that specific transaction. Every transaction has its own private buffer that caches all accessed records for the duration of the transaction's lifetime. This caching mechanism has several obvious limitations. There is no shared buffer across transactions and buffering is solely beneficial if a transaction accesses the same record several times.

5.4.2 Shared Record Buffer

The shared record buffer is used by all transactions on a processing node. It acts as a caching layer between the transaction buffers and the storage system. Although records can be changed by remote processing nodes and new transactions need to access the storage system to fetch the most recent version of a record, transactions running in parallel can benefit from a shared buffer. For instance, if a transaction retrieves a record, the same record can be *reused* by a transaction that has started before the first one (i.e., a transaction with a smaller snapshot descriptor).

This strategy is implemented using *version number sets* (as defined in Section 4.3). In the buffer we associate to each record a version number set that specifies for which version numbers the record is valid. Comparing the version number set of the transaction's snapshot descriptor with the version number set of the buffered record allows to determine if the buffer entry can be used or if the transaction is too recent. If a transaction with version number set V wants to read a buffered record with version number set C, the following conditions are used to verify the validity of the entry:

(1) $V \subseteq C$: The buffer is recent enough. We can access the record from the buffer and no interaction with the storage system is necessary.

(2) $V \nsubseteq C$: The cache might be outdated. We first get V_{max}, the version number set of the most recently started transaction on the processing node. Then, we fetch the record from the storage system and replace the buffer entry. Finally C is set to V_{max}. As all transactions in V_{max} committed before the record was read, it is certain to be a valid version number set.

The idea here is to keep C as big as possible to improve the likelihood that condition 1 holds for future data accesses. If a record is not in the buffer yet, we apply the same procedure as in condition 2. If the buffer is full a common replacement strategy (e.g., Least-Recently-Used) can be used to evict entries. Record updates are applied to the buffer in a *write-through* manner. Each time a transaction performs an update operation in the *Try-Commit* phase, the changes are written to the storage system and if successful to the buffer as well. C is set to the union of *tid* and V_{max}. V_{max} is also a valid version number set for updated records because if a transaction in V_{max} would have changed the record, the write operation to the storage system would have failed. Updates to the buffered record and the version number set are executed as atomic operations to ensure consistent buffer changes.

In the presence of multiple concurrent transactions accessing the same items on a processing node, global buffering reduces the number of read requests to the storage system and thus lowers data access latencies. This comes at the price additional data management overhead and more memory consumption on the processing nodes.

5.4.3 Shared Buffer with Version Set Synchronization

This variant is an extension of the previously described shared buffer. The key idea of this approach is to use the storage system to synchronize the version number sets of records. The advantage is that a processing node can verify if a buffered record is still valid by retrieving its version number set and only afterwards re-fetch the record if necessary. Additionally, this strategy saves network bandwidth as a version number set is tiny compared to a record.

Version sets are maintained in a separate domain in the storage system. Record updates are handled the same way as the shared record buffer except that a transaction not only writes the data changes to the storage system but also updates the version number set entry. The mechanism for data access is slightly different. The conditions to access a buffered record are as follows:

1. $V_{tx} \subseteq C$: The buffer entry is valid.
2. $V_{tx} \nsubseteq C$: The cache might be outdated. We fetch the record's version number set C' from the storage system.
   (a) If $C' = C$ the buffered record is still valid.
   (b) If $C' \neq C$ the record must be re-fetched. Moreover, C is replaced by C'.

Although this strategy reduces network traffic, it comes at the expense of additional update overhead. For each record update, two requests have to be sent to the storage system.

An optimization to reduce the number of additional storage system requests is *record grouping*. Instead of maintaining one version number set per record, multiple records are grouped in a *cache unit* and share a common version number set. By default several sequential records of a relational table are assigned to one cache unit. The number of records per cache unit is a configuration parameter. The mechanism for record access and update remains the same except that once the version number set is updated, all buffered records of a cache unit are invalidated. The advantage of this strategy is that many fewer version number sets have to be written and read from the storage system. For instance, multiple updates to the same cache unit only require the version number set to be updated once. On the other hand, as a trade-off, records are invalidated more often and need to be re-fetched from the storage system.

This strategy benefits from a workload with a high read ratio as cache units will be invalidated less often. The higher the update ratio, the more buffered records will be invalidated and the additional update overhead will at some point outweigh the saved requests to the storage system.

6 Experimental Evaluation

This section provides an experimental evaluation of the techniques and the architecture presented in this work. We first describe the implementation and the benchmark environment before presenting experimental results. Our experiments use the TPC-C benchmark that is popular both in industry and academia.

6.1 Implementation and Setup

The architecture is composed of two architectural layers. The top layer, responsible for query processing and transaction management, consists of two components: The processing node (PN) and the commit manager (see Figure 3). Both components have been implemented in 15,608 lines of C++ code. The processing node provides an interface that allows to process any SQL statement. However, our prototype does not contain a query parser and optimizer yet, and therefore every query we run is implemented with a custom query plan. Transaction management and data access is implemented as described in the Sections 4 and 5 respectively.

The processing node interacts with the RamCloud (RC) storage system [31]. RC is a strongly consistent in-memory key-value store designed to operate in low-latency networks. Data is accessed using a client library that supports atomic *get*/*put* operations (i.e., LL/SC) and allows organizing key-value pairs in tables. Tables can be range partitioned in order to distribute load across storage nodes (SN). In our experiments, tables are by default evenly partitioned across all available storage nodes. To guarantee fault-tolerance RamCloud supports remote backup with fast recovery. The backup mechanism synchronously replicates every *put* operation to other nodes and thereafter asynchronously writes it to persistent storage. The replication factor (RF) specifies the number of data copies.

The benchmark infrastructure consists of 12 servers. Each machine is equipped with two quad core Intel Xeon E5-2609 2.4 GHz processors, 128 GB DDR3-RAM and a 256 GB Samsung 840 Pro SSD. A server has two NUMA nodes that consist each of one processor and 50% of total memory. A process (or node) is by default assigned to one NUMA node. This assignment allows to run two processes per server (usually a PN and a SN) and thus doubles the maximum number of nodes (24). We did not notice a performance impact compared to single process operation.

The servers are connected to a 40 Gbit QDR Infiniband network. All communication between components uses this network. Alternatively, the implementation can be configured to run on a 10 Gbit Ethernet network. All nodes are connected to the same switch.

6.2 The TPC-C Benchmark

The TPC-C is an OLTP database benchmark that models the activity of a wholesale supplier. The benchmark consists of five transactions that include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses.

Load is generated by *terminal* clients that emulate users entering data on an input screen. Every terminal operation results in one database transaction. The processing nodes have custom query plans for every SQL query of the TPC-C, and thus can execute all transactions. Terminals are run separately from the system under test (SUT) and communicate with the PNs over Ethernet. Our TPC-C implementation slightly differs from the official specification. First, we have removed wait times so that terminals continuously send new requests. The number of terminal threads is selected so that the peak throughput of the SUT is reached. Second, the default measurement interval for a TPC-C run is 10 minutes. The benchmark is executed 5 times for each configuration and the presented results are averaged over all runs.

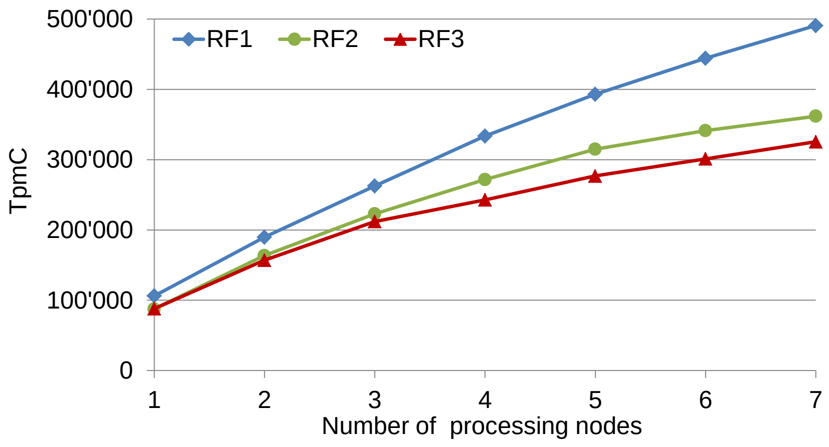

Figure 5: Scale-Out (Vary PN, 5SN, Vary RF)

The size of the database is determined by the number of warehouses (WH). Each warehouse is populated with 30,000 customers and 30,000 orders with an average of 10 order-lines each. The default population for every TPC-C run is 100 WH. In our implementation, one warehouse occupies approximately 189 MB of RamCloud memory space.

The primary TPC-C metric is the new-order transaction rate (TpmC). That is, the number of successfully executed new-order transactions per minute. Not included are aborted transactions or transactions that exceed the TPC-C response time threshold. The TpmC represents around 45% of all issued transactions so that the absolute number of transactions processed by the database is higher. Additionally, we measure latency and transaction abort rate as these metrics give valuable information about system behavior. For most conducted experiments, the TpmC performance turned out to be predictable and the measured variations were very low.

6.3 Scale-Out

The scale-out experiments evaluate the scalability of the shared-memory architecture. The number of PNs, SNs and the RF is described for each experiment. If not mentioned otherwise, we use a single commit manager.

Processing Node. Figure 5 shows the scalability of different replication factors with varying number of processing nodes. With no replication (RF1) the throughput grows continuously from 106,202 TpmC with 1 PN to 490,870 TpmC with 7 PNs. The performance numbers are a good illustration that the shared-memory architecture can scale out well given a fast storage layer with low data access latencies. The throughput increase could even be more linear, if the TPC-C benchmark would not suffer from data contention. Two transactions increasingly fail and cause the transaction abort rate to grow (see Table 2). The

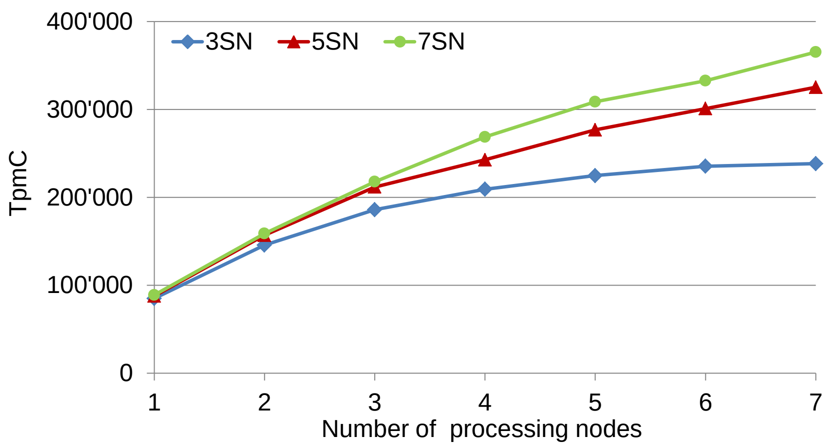

Figure 6: SN Scale-Out (Vary PN, Vary SN, RF3)

Number PNs	1	3	5	7
TpmC	106,202	262,496	393,109	490,870
Payment abort (%)	7.52	18.62	27.50	34.62
Delivery abort (%)	2.49	8.22	13.07	18.31

Table 2: Tx Abort Rate (Vary PN, 5SN, RF1)

payment transaction (43% of all transactions) conflicts in updating the warehouse entries (only 100 tuples). The delivery transaction (4% of all transactions) competes in delivering (deleting) new orders.

With replication the system can tolerate storage node failures. RF3 reaches a peak throughput of 325,388 TpmC with 7 PNs, thus 33.71% less than RF1. This difference is caused by higher latencies of basic data operations. Higher latency directly affects transaction response time and thus reduces the number of transactions a worker thread can process. For example, in the 7 PN configuration the delivery transaction takes on average 151.01 ms with RF3 and only 91.94 ms with RF1. Conceptually it is obvious that operations requiring replication (i.e., *put* and *delete*) take more time. However, during replication RamCloud blocks, and as a side effect unnecessarily delays *get* requests. We assume that without this implementation issue the performance difference would be much lower. The transaction abort rates with RF3 are similar than with RF1 as the probability of conflict is alike (higher processing time compensates for lower throughput).

Storage Node. Figure 6 shows the scale-out of the storage layer. In a configuration with 3SN, the storage layer becomes the bottleneck as soon as we run more than three processing nodes. Adding more nodes (5SN and 7SN) allows for more scalability and throughput increases. As shown previously, growing data contention generated by the TPC-C becomes the limiting factor and will

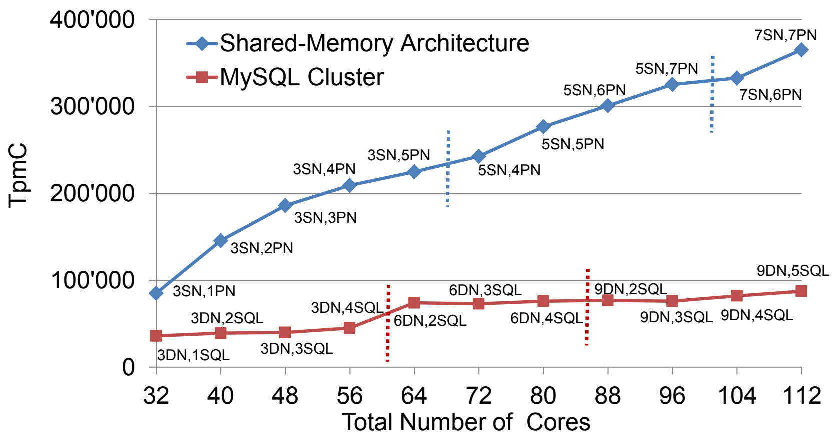

Figure 7: MySQL Cluster (Vary #Cores, RF3)

	TpmC	Latency (mean $\pm \sigma$, ms)	TP99 (ms)	TP999 (ms)
Infiniband	490,870	19.34 ± 23.84	101.79	169.44
10Gb Ethernet	151,560	63.93 ± 80.60	387.67	483.69

Table 3: Throughput, Latency (7PN, 5SN, RF1)

eventually cause the throughput to stabilize. Nonetheless, this experiment is a good illustration on how adding machines to both architectural layers improves performance.

6.4 Comparison to MySQL Cluster

MySQL Cluster is a shared-nothing database system that has been continuously optimized over the last years. A cluster configuration consists of three components: *Management nodes* that monitor the cluster, *Data nodes* (DN) that store data in-memory and process queries, and *SQL nodes* that provide an interface to applications and act as federators towards the data nodes. By default, MySQL Cluster automatically partitions every table by primary key. However, the partitioning key can also be specified manually to ensure that related data is stored on the same shard. MySQL Cluster synchronously replicates data. Internally, *replication groups* require the number of data nodes to be a multiple of the replication factor. Unfortunately, no option is provided to fully replicate tables across all partitions, which is a useful feature for read-only tables.

Our benchmark environment runs MySQL Cluster 7.3.2 and uses the Infiniband network for communication. The RF is 3 and thus we benchmark configurations with 3, 6 and 9 data nodes. We use a varying number of SQL nodes and two management nodes. Terminals use the JDBC interface to exe-

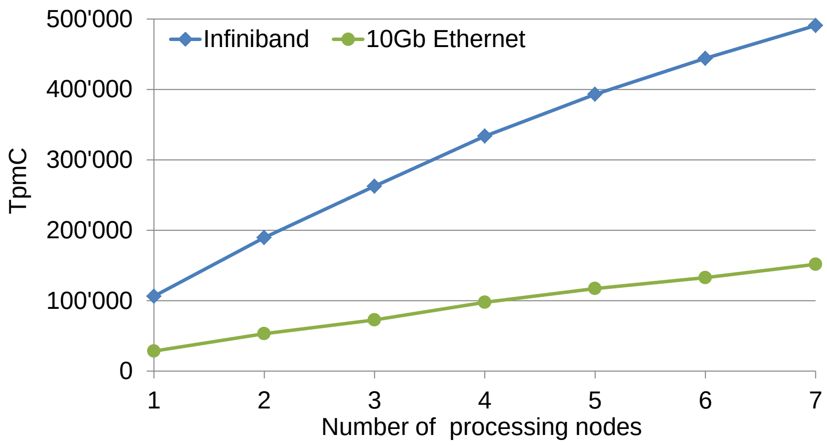

Figure 8: Network (Vary PN, 5SN, RF1)

cute transactions against the SUT. All tables are partitioned by *warehouse id* except the *item* table. Items are partitioned by primary key as they have no dependencies.

Figure 7 compares MySQL Cluster to the shared-memory architecture. The figure presents the peak TpmC values for varying number of CPU cores (i.e., the sum of all cores available to the database cluster). Each data point corresponds to the configuration with the highest throughput for the given number of cores. In comparison to the shared-memory architecture, MySQL Cluster struggles. The throughput only slightly increases with the number of cores from 35,890 to a maximum of 87,400 TpmC. The jump between 56 and 64 cores reflects the switch from 3 to 6 data nodes. The next step from 6 to 9 data nodes however, does not produce a noticeable effect. The scalability is presumably limited by the partitioned item table which causes a lot of distributed transactions. The results underline that correct partitioning in a crucial performance factor in SN architectures. The experiment highlights that the techniques presented in this work efficiently interact and provide comparable, if not higher performance than state-of-the-art shared-nothing databases. At the same time, the shared-memory approach provides dynamic scale-out and enables mixed workloads. These advantages make it a serious alternative to common shared-nothing designs.

6.5 Network

Throughout the paper we emphasized that fast data access is a major requirement for scalability. This experiment highlights the importance of low-latency data access in a shared-memory architecture. Figure 8 compares the throughput achieved with an Infiniband network to the performance of a 10 Gb Ethernet network. The experiment varies the number of storage nodes. RF is set to 1 and the number of SNs is constant. The TpmC results on Infiniband are more

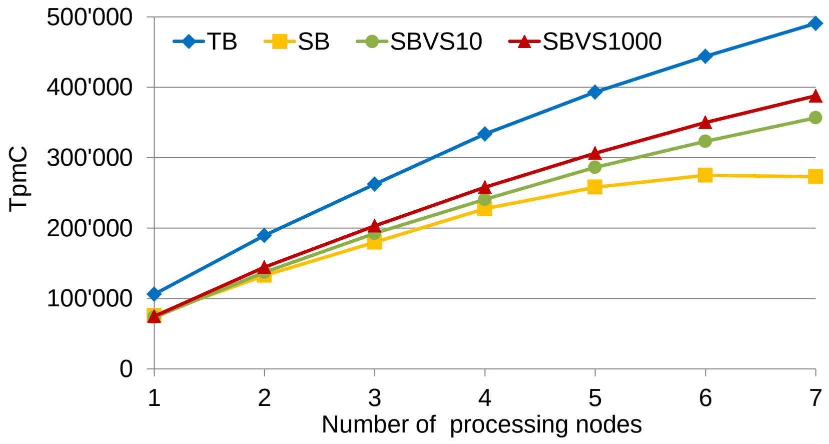

Figure 9: Buffering Strategy (Vary PN, 5SN, RF1)

than three times higher than the results achieved with Ethernet independent of the number of processing nodes. This difference is a direct effect of network latencies. Infiniband uses RDMA and by-passes the networking stack of the operating system to provide much lower latencies. Table 3 summarizes the results for the fastest configuration with 7 PNs. The second column shows the mean transaction response time in ms and the standard deviation. Both values confirm the measured divergence in throughput. The last two columns show the 99th and 99.9th percentile response time and highlight the stable performance of both networking technologies. In the Infiniband configuration the total bandwidth usage of one storage node is 344.8 MB/s (in and out). Consequently, the network is not saturated.

6.6 Buffering Strategies

Figure 9 shows a performance comparison of the buffering strategies presented in Section 5.4. The *transaction buffer (TB)* used in all previous experiments is the best strategy for the TPC-C and reaches the highest throughput. The *shared record buffer (SB)* performs worse because the overhead of buffer management outweighs the caching benefits. The cache hit ratio of 1.42% is very low. The *shared buffer with version set synchronization (SBVS)* tested with cache unit sizes of 10 and 1000 has a considerably higher cache hit ratio (37.37% for SBVS1000). Nevertheless, this can not compensate for the cost of additional storage system requests to the version number set table. A key insight from these results is that with fast RDMA the overhead of buffering data does not pay of for workloads such as TPC-C.

7 Conclusions

In this paper we introduced a new database architecture based on the shared-memory principle. The architecture decouples transactional query processing and data storage into two independent layers and thus enables elasticity and workload flexibility. Data is stored in main memory in a distributed record manager which is shared among all database instances. To address the synchronization problem in shared-data environments, we presented a protocol to perform distributed multi-version concurrency control using lightweight atomic RMW operations for conflict detection. Furthermore, we explained how relational data can be efficiently stored and accessed to enable scalable query processing.

The experimental evaluation highlights the ability of the shared-memory architecture to scale-out to a high number of nodes and emphasizes the importance of low-latency networking to achieve high performance. In addition, we compared to shared-nothing databases and achieved a much higher throughput than the popular MySQL Cluster database. Providing competitive performance while at the same time being elastic and workload flexible, makes the shared-memory architecture well-suited for unpredictably evolving workloads. It is thus a perfect candidate for *big data*.

References

[1] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, 27(3):5:1–5:48, 2009.

[2] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. CIDR '11, pages 223–234, 2011.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. SIGMOD '95, pages 1–10, 1995.

[4] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[5] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[6] P. Bernstein, C. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR '11*, pages 9–20, 2011.

[7] D. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full SQL language support in microsoft SQL Azure. SIGMOD '10, pages 1021–1024, 2010.

[8] S. Chandrasekaran and R. Bamford. Shared cache - the future of parallel databases. ICDE '03, 2003.

[9] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: a distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.

[10] S. Das, D. Agrawal, and A. El Abbadi. Elastras: an elastic transactional data store in the cloud. HotCloud '09, 2009.

[11] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. SoCC '10, pages 163–174, 2010.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. SOSP '07, pages 205–220, 2007.

[13] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[14] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. SIGMOD '84, pages 1–8, 1984.

[15] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[16] G. Graefe. A survey of B-tree locking techniques. *ACM Trans. Database Syst.*, 35(3):16:1–16:26, 2010.

[17] Infiniband. http://www.infinibandta.org/, Apr. 2013.

[18] E. Jensen, G. Hagensen, and J. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, 1987.

[19] J. Josten, C. Mohan, I. Narang, and J. Teng. Db2's use of the coupling facility for data sharing. *IBM Systems Journal*, 36(2):327–351, 1997.

[20] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[21] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

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

[23] P. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[24] J. Levandoski, D. Lomet, M. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR '11*, pages 123–133, 2011.

[25] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-tree: a B-tree for new hardware platforms. ICDE '13, 2013.

[26] D. Lomet, R. Anderson, T. Rengarajan, and P. Spiro. How the Rdb/VMS data sharing system became fast. Technical Report CRL 92/4, 1992.

[27] D. Lomet, A. Fekete, G. Weikum, and M. Zwilling. Unbundling transaction services in the cloud. CIDR '09, 2009.

[28] M. Mages. ABA prevention using single-word instructions. Technical Report RC23089 (W0401-136), 2004.

[29] C. Mohan. History repeats itself: sensible and nonsensql aspects of the nosql hoopla. In *EDBT '13*, pages 11–16, 2013.

[30] MySQL Cluster. http://www.mysql.com/, Apr. 2013.

[31] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. Rumble, E. Stratmann, and R. Stutsman. The case for RAMCloud. *Commun. ACM*, 54(7):121–130, 2011.

[32] V. Padhye and A. Tripathi. Scalable transaction management with snapshot isolation on cloud data management systems. CLOUD '12, pages 542–549, 2012.

[33] J. Rao, E. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4(4):243–254, 2011.

[34] M. Rys. Scalable sql. *Queue*, 9(4):30:30–30:37, 2011.

[35] ScaleDB. http://www.scaledb.com/, Apr. 2013.

[36] A. Singhal, R. Van der Wijngaart, and P. Barry. Atomic read modify write primitives for I/O devices. Technical Report Intel White Paper, 2008.

[37] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[38] M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, 54(6):72–80, 2011.

[39] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). VLDB '07, pages 1150–1160, 2007.

[40] TPC. *TPC benchmark C 5.11.* TPC Counsil, 2010.

[41] VoltDB. http://www.voltdb.com/, Apr. 2013.