On the Design and Scalability of Distributed Shared-Data Databases

Simon Loesing † Markus Pilman † Thomas Etter † Donald Kossmann †*

† Department of Computer Science
ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

* Microsoft Research
Redmond, WA, USA
donaldk@microsoft.com

ABSTRACT

Database scale-out is commonly implemented by partitioning data across several database instances. This approach, however, has several restrictions. In particular, partitioned databases are inflexible in large-scale deployments and assume a partition-friendly workload in order to scale. In this paper, we analyze an alternative architecture design for distributed relational databases that overcomes the limitations of partitioned databases. The architecture is based on two fundamental principles: We decouple query processing and transaction management from data storage, and we share data across query processing nodes. The combination of these design choices provides scalability, elasticity, and operational flexibility without making any assumptions on the workload. As a drawback, sharing data among multiple database nodes causes synchronization overhead. To address this limitation, we introduce techniques for scalable transaction processing in shared-data environments. Specifically, we describe mechanisms for efficient data access, concurrency control, and data buffering. In combination with new hardware trends, the techniques enable performance characteristics that top state-of-the-art partitioned databases.

1. INTRODUCTION

Modern large-scale web applications have requirements that go beyond what traditional relational databases management systems (RDBMS) provide. For instance, traditional RDBMS are inflexible with regard to evolving data and are too rigid for dynamic cloud environments. In response to these limitations, new design considerations have triggered the emergence of NoSQL (Not Only SQL) storage systems. NoSQL systems have been hyped for their scalability and availability. However, the fundamental design premise behind the NoSQL phenomenon is operational flexibility [39]. Operational flexibility describes all features that either facilitate 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 evaluate an architecture for distributed transaction processing that is designed towards operational flexibility. Our goal is to keep the strengths of RDBMS, namely SQL and ACID transactions, and at the same time provide the benefits that inspired the NoSQL movement. This architecture is based on two principles: First, data is shared across all database instances. In contrast to distributed partitioned databases [41, 60], database instances do not exclusively own a partition but can access the whole data and execute any query. As a result, transactions can be performed locally. This is a key benefit over partitioned database in which distributed transactions can significantly affect scalability. The second principle 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 architectural layers. This break-up facilitates elasticity and deployment flexibility as both layers can be scaled out independently [37]. Applying both principles results in a two-tier architecture in which database instances operate on top of a shared record store. Hence, the name *shared-data architecture*.

The shared-data architecture has shown to scale for analytical workloads (OLAP). For instance, it is the foundation behind the success of Apache Hadoop [2]. This paper uses the same elemental principles to allow for scalable online transaction processing (OLTP). In a shared-data architecture, it is even possible to run an OLTP workload and perform analytical queries on separate instances but accessing the same data. This mixed workload scenario enables scalable analytics on live production data. Although the focus of this paper is on OLTP workloads, we outline the implications of performing mixed workloads.

Sharing data raises several technical challenges we address: First, shared data access requires synchronization [52]. We use lightweight *load-link and store-conditional* (LL/SC) primitives [25] as a foundation to implement efficient multiversion concurrency control (MVCC) [5] and provide latch-free indexes [35]. Second, data access involves network communication. We provide techniques to minimize network requests and keep latencies low. In particular, we rely on new technical trends such as fast networking technology (i.e., InfiniBand [24]) as well as in-memory data storage. It is the specific combination of techniques that enables scalability not possible a few years ago.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2751519.

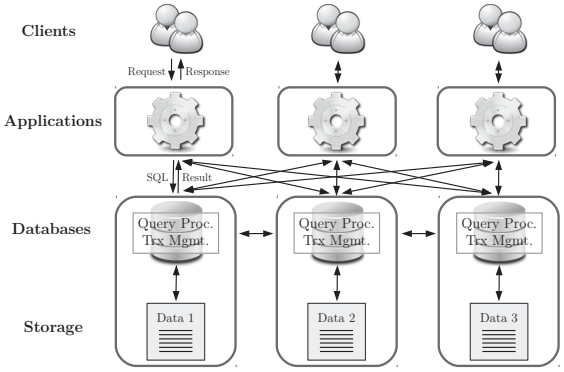

Figure 1: Partitioned databases

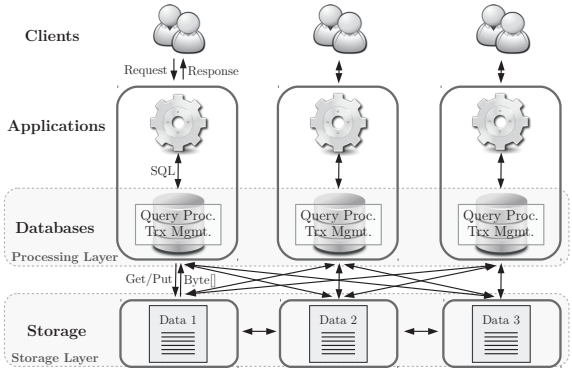

Figure 2: Shared-data databases

This paper makes three contributions: First, we propose an architecture design for distributed relational databases that enables scalability, fault-tolerance, and elasticity without making any assumptions on the workload. Second, we demonstrate that the right combination of established concepts and new hardware trends enables scalable transaction processing in shared-data systems. Accordingly, we describe techniques for efficient and consistent data access as part of a reference implementation of the shared-data architecture, a database system called *Tell*. Third, we show the effectiveness of our design and evaluate it against VoltDB [54], MySQL Cluster [41] and FoundationDB [19].

The paper is organized as follows: Section 2 details the key design principles and technical challenges of shared-data architectures. We review related work in Section 3. Section 4 describes Tell and explains how concurrency control and recovery are implemented. Section 5 details data access and shared storage. Section 6 presents an experimental evaluation of Tell, and Section 7 concludes.

2. SHARED-DATA ARCHITECTURE

This section describes the design principles and technical challenges behind the shared-data architecture.

2.1 Design Principles

In the following, we detail the key design principles motivated in the previous section. Moreover, we highlight the necessity for ACID transactions and complex queries, two features that have recently been relaxed or simplified in many storage systems [12, 16, 45].

Shared data: Shared data implies that every database instance can access and modify all data stored in the database. There is no exclusive data ownership. A transaction can be executed and committed by a single instance. In contrast to partitioned databases, no workload knowledge is required to correctly partition data and minimize the number of distributed transactions. Sharing data provides the following benefits: First, setting up a database cluster is straightforward. Second, the 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 address in Section 2.2.

Decoupling of Query Processing and Storage: The shared-data architecture is decomposed into two logically independent layers, transactional query processing and data storage. The storage layer is autonomous. It is implemented as a self-contained system that manages data distribution and fault-tolerance transparently with regard to the processing layer. Hence, replication and data re-distribution 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* (SN). Data partitioning is not as performance critical as in partitioned databases as data location does not determine where queries have to be executed. Instead, to execute queries, the processing layer accesses records independent of the SNs they are located at. This fundamental difference in the communication pattern is highlighted in Figures 1 and 2.

The processing layer consists of multiple autonomous *processing nodes* (PN) that access the shared storage system. A mechanism is provided to retrieve data location (e.g., a lookup service) that enables the processing nodes to directly contact the storage node holding the required data.

Logical decoupling considerably improves elasticity as PNs or SNs can be added on-demand if processing resources or storage capacity is required respectively. Accordingly, nodes can be removed if capacity is not required anymore. Moreover, the architecture enables workload flexibility. That is, the possibility to execute different workloads. For example, some PNs can run an OLTP workload, while others perform analytical queries on the same dataset.

In-Memory Storage: Modern commodity hardware clusters typically have terabytes of main memory at their disposal. These numbers exceed the capacity requirements of most applications. For instance, in the TPC-C benchmark [57], a warehouse requires less than 200 MB. Consequently, it is no longer necessary to rely on slow storage devices (e.g., hard disks), and instead, the whole data can be kept in-memory. The advantages are obvious: In-memory storage provides low access latencies and avoids complex buffering mechanisms [53]. However, DRAM memory is volatile and data loss must be prevented in case of failures. A common approach to ensure fault-tolerance is replication.

ACID Transactions: Transactions ensure isolated execution of concurrent operations and maintain data integrity. From a 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. Transactions are indispensable for the majority of applications and are therefore a major feature. We aim at providing full transaction support. That is, ACID transactions can be performed without limitations on the whole data. Transaction management is part of the processing layer and does not make any assumption on how data is accessed and stored.

Complex Queries: SQL is the query language in RDBMS. It enables complex queries and includes operators to order, aggregate, or filter records based on predicates. Although alternative systems often simplify the query model to improve performance, complex (SQL) queries are not an obstacle to system scalability [47]. In the shared-data architecture, data location is logically separated from query processing. As a result, PNs retrieve the records required to execute a query. This notion can be described as *data is shipped to the query*. The ability to run the same query on multiple nodes provides increased parallelism and scalability.

2.2 Technical Challenges

Shared-data architectures run any workload and facilitate elasticity. However, they introduce new challenges.

Data Access: In a shared-data environment, data is likely to be located at a remote location and has to be accessed over the network. Caching data in local buffers is only possible to a limited extent as updates made by one PN have to be visible to the others instantly. As a result, to provide consistent access, most requests retrieve the latest record version remotely from the storage layer. Data access latencies can quickly become a dominant factor in query execution time and thus, minimizing the interaction with the storage layer is an optimization goal in shared-data systems.

In light of restricted buffering possibilities, traditional techniques to manage relational data have to be re-evaluated. 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 a request, or whether choosing a finer storage granularity (e.g., single record) is a more favorable approach. We suggest to store data at the granularity of a record as this provides a good trade-off between the number of network requests and the amount of traffic generated. We justify this design choice and detail the 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 PNs and are therefore subject to concurrent modifications from distributed locations. As a result, the shared-data architecture requires distributed indexes that support atomic operations and that are at the same time highly scalable. As a solution, we propose a scalable latch-free distributed B+Tree index in Section 5.3.

In addition to the right techniques, low-latency networking technologies such as InfiniBand contribute to reduce the overhead of remote data access. These technologies provide latency and bandwidth guarantees not available a decade back and consequently enable to scale to new levels. For instance, InfiniBand enables Remote Direct Memory Access (RDMA) within 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 PNs. Although common pessimistic and optimistic concurrency control approaches [6] can be used, 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. 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 allows for the highest scalability.

In this paper, we present a distributed MVCC protocol. A key feature is conflict detection using LL/SC primitives, a particular type of atomic *read-modify-write* (RMW) operation [50]. Atomic RMW operations have become popular in recent years as they enable non-blocking synchronization. In a shared-data architecture, they allow for performing 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. LL/SC 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 and are 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 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 records can cause network saturation. Some of the techniques we present introduce additional constraints that are detailed at a later point.

3. RELATED WORK

Distributed databases that share data have been subject to various research over the years. Oracle RAC [11], IBM DB2 Data Sharing [28], and Oracle Rdb [36] are based on a shared-data architecture. These systems use a 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. More fundamentally, these systems rely on traditional tightly-coupled RDBMS engines.

An initial design of a shared-data architecture based on the principles presented in Section 2.1 has been introduced by Brantner et al. [8]. Moreover, the benefits of decoupling data storage from query processing and transaction management with regard to flexibility and elasticity have been highlighted in several publications [34, 37]. A commercial database system that implements our design principles and that has been developed in parallel to Tell is FoundationDB [19]. FoundationDB provides a "SQL Layer" that enables complex SQL queries on top of a transactional key-value store. FoundationDB has many similarities with Tell such as support for in-memory storage and optimistic MVCC protocol. However, critical implementation details (e.g., indexing, commit validation) have not been published yet. The purpose of this paper is to specifically provide these details. The experimental evaluation highlights that it is the right combination of techniques that is fundamental to achieve high performance. In the TPC-C benchmark Tell outperforms FoundationDB by a factor of 30 (Section 6.5).

Several additional shared-data databases have been published recently. ElasTras [15] is a shared-data database de-

	Shared Data	Decoupling	In-Memory Storage	ACID Transactions	Complex Queries
Tell (Shared-Data Architecture)	✓	✓	✓	✓	✓
Oracle RAC	✓	-	-	✓	✓
FoundationDB	✓	✓	✓	✓	✓
Google F1	✓	✓	-	✓	✓
OMID	✓	✓	-	✓	✓
Hyder	✓	✓	-	✓	(✓)
VoltDB	-	-	✓	✓	✓
Azure SQL Database	-	-	-	✓	✓
Google BigTable	-	-	-	-	-

Table 1: Comparison of selected databases and storage systems

signed for multi-tenancy. Data storage is decoupled from transaction management but data partitions (or tenants) are exclusively assigned to PNs. Transactions can only be performed on single partitions. In our architecture, PNs can access all data and the scope of a transaction is not limited. Another shared-data database is Google F1 [49]. F1 is built on top of Spanner [13] that has evolved from Megastore [3]. F1 enables scalable transactions but is designed with regard to cross-datacenter replication. The reported latencies to perform a commit are 50-150 ms. We assume low-latency data access in LANs. A system that implements many of our design principles is OMID [20]. OMID implements MVCC on top of a large data store to provide transaction support. However, unlike our approach, OMID requires a centralized component for conflict detection and commit validation. Hyder [7] is a transactional shared-data record manager. Records are stored in a log structure. Updates are appended to the log in total order, and processing servers rollforward the log to reach a common state. Hence, transaction management is decoupled from storage.

Distributed partitioned databases are nowadays widespread. A thorough review of the architectural benefits is provided by Dewitt and Gray [17]. Partitioned databases provide scalability for partition-friendly workloads but suffer from distributed transactions. H-Store [29] and its commercial successor VoltDB [60] horizontally partition tables inside and across nodes. VoltDB sequentially processes transactions on single partitions without the overhead of concurrency control. Calvin [56] speeds up distributed transactions by reaching agreement before a transaction is executed. Jones et al. [26] propose to speculatively execute local transactions while waiting for distributed transactions to complete. Azure SQL Database [10] is a cloud service that enables to partition data across instances but does not support cross-partition joins. In a shared-data system, the problem of distributed transactions does not arise as PNs can access all data. Partitioning is transparent with regard to the PNs and transactions are local. Accordion [48] and E-Store [55] are two recent systems that propose data placement systems to enable online repartitioning [51] and provide elasticity for partitioned databases. Although these approaches considerably improve agility, adding or removing node implies re-partitioning and moving data. In Tell, elasticity is more fine-grained and PNs can be added without any cost.

Main memory databases [32, 58] take advantage of memory performance and cache affinity to optimize processing in the scope of a single machine. A main memory database designed for processing mixed workloads is HyPer [30]. HyPer creates copy-on-write snapshots of the data in virtual memory to process OLAP workload independently of on-going OLTP processing. A recent extension [40] allows to offload analytical processing onto secondary machines to improve OLAP performance. The main difference to Tell is that OLTP queries are only processed on a single master server.

Recently, relational databases have been challenged by the emergence of NoSQL systems. NoSQL stores violate our design principles and typically relax consistency guarantees in favor of more scalability and availability. Amazon's Dynamo [16] was one of the first scalable key-value stores that provides eventual consistency to achieve high availability. Other systems such as BigTable [12] or Spinnacker [45] provide strong consistency and atomic single-key operations but no ACID transactions. G-Store [14] does support ACID transactions for user-defined groups of data objects. The major restriction is that these groups must not overlap. A system that does not have this limitation is Sinfonia [1]. Sinfonia is a partitioned store that uses Two-Phase Commit to reach agreement on transaction commit. None of the previous NoSQL stores supports a query language as powerful as SQL. However, many strongly consistent NoSQL systems are equivalent to atomic record stores and support basic operations as required by the storage layer in Tell.

Table 1 provides an overview of the discussed systems with regard to our design principles.

4. TRANSACTION PROCESSING AND CONCURRENCY CONTROL

Figure 3 presents an overview of the components of Tell. The processing node is the central component that processes incoming queries and executes transactions. PNs interact with the commit manager, a lightweight service that manages global transaction state (Section 4). Furthermore, a management node monitors the system and initiates a recovery process the moment a failure is detected. Finally, data is stored in a distributed storage system consisting of multiple SNs. The storage layer manages data records, indexes, as well as an undo log for recovery (Section 4.4). To enable transactional processing, the storage system must support consistent get/put operations on single records. Moreover, Tell depends on the availability of the storage system in order to process queries.

Concurrency control ensures that transactions can run in parallel on multiple PNs without violating data integrity. In this section, we describe a MVCC protocol [5]. More specifically, we detail a distributed variant of *snapshot isolation* (SI) [4] that uses LL/SC primitives for conflict detection. The SI protocol is part of the transaction management component on each PN in Tell and guarantees ACID properties. SI is an optimistic protocol [31] in which transactions are never prevented from making progress. Evaluating our design with lock-based protocols is subject to future work.

4.1 Distributed Snapshot Isolation

SI is a MVCC protocol that has been implemented in several database systems (e.g., Oracle RDBMS, PostgreSQL, Microsoft SQL Server). 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. This is the so-called *consistent snapshot* the transaction operates with. Transactions check for conflicts at commit time. Updates (insert, update, and delete operations) are buffered and applied to the shared store during commit. In the remainder of the paper, we use the term "apply" to indicate that updates are installed in the store.

The commit of transaction T_1 is only successful if none of the items in the *write set* have been changed externally (applied) 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 back. For this purpose, Tell executes an LL/SC operation in the storage layer. LL/SC is a pair of instructions that reads a value (load-link) and allows for performing an update 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 [38]. That is, a write operation on a modified item fails even when the value matches the initially read value. 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, there is a write-write conflict and T_1 will abort (i.e., revert all changes made to the store).

SI avoids many of the common concurrency control anomalies. However, some anomalies (e.g., write skew) prevent SI to guarantee serializability in all cases [18]. Proposed solutions for serializable SI [9, 27, 61] are not designed for distributed systems or require centralized commit validation. Still, we mean to provide serializable SI in the near future.

To provide SI semantics across PNs, each node interacts with a dedicated authority, the *commit manager*.

4.2 Commit Manager

The commit manager service manages global snapshot information and enables new transactions to retrieve three elements: A system-wide unique *transaction id (tid)*, a *snapshot descriptor* and the *lowest active version number (lav)*.

Transaction ids are monotonically incremented numeric values that uniquely identify a transaction. Every running transaction requires a *tid* before it can execute data operations. Given its system-wide uniqueness, the *tid* is not only used to identify transactions but also defines the version number for updated data items. In other words, a transaction creates new versions for updated data items with the *tid* as version number. Due to the fact that *tids* are 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 (i.e., the set of accessible version numbers) for starting transactions.

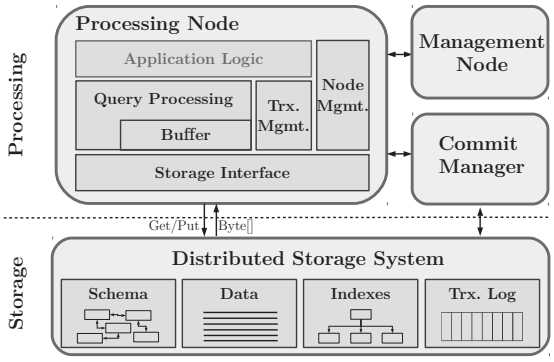

Figure 3: Tell architecture and components

The snapshot descriptor is a data structure computed for each transaction that specifies which versions can be accessed. It consists of two elements: First, a *base version number* b indicating that b and all earlier transactions have completed. Second, a set of newly committed *tids* N. 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 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. Hence, the snapshot descriptor is small even in the presence of many parallel transactions (e.g., $N \leq 13$ KB with 100,000 newly committed transactions).

When accessing a data item with a set of version numbers (or version number set) V, the transaction reads the version with the highest version number v matching the snapshot descriptor. More formally, we define the valid 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 active transactions. That is, the highest version number globally visible to all transactions. Version numbers smaller than the *lav* are candidates for garbage collection (Section 5.4).

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

- $start() \rightarrow (tid, snapshot\_descriptor, lav)$: Signal the start of a new transaction. Returns a new *tid*, a snapshot of the current state, and the *lav*.
- $setCommitted(tid) \rightarrow void$: Signals that *tid* has successfully committed. Updates the list of committed transactions used to generate snapshot descriptors.
- $setAborted(tid) \rightarrow void$: Same as above but for aborts.

To scale and ensure fault-tolerance, several commit managers can run in parallel. Commit manager nodes use the shared store to synchronize on *tids* and on the snapshot. The uniqueness of every *tid* is ensured by using an atomically incremented numeric counter in the storage system. PNs update the counter using LL/SC operations to ensure that

tids are never assigned twice. To prevent counter synchronization from becoming a bottleneck, PNs can increment the counter by a high value (e.g, 256) to acquire a range of *tids* that can be assigned to transactions on-demand. We opted for continuous ranges of *tids* because it is simple to implement. However, the approach has limitations (e.g., higher abort rate). Using ranges of interleaved *tids* [58] is subject to be implemented in the near future. To synchronize the current snapshot (i.e., the list of committed transactions), we use the storage system as well. In short intervals, every commit manager writes its snapshot to the store and thereafter reads the latest snapshots of the other commit managers. As a result, every commit manager gets a globally consistent view that is at most delayed by the synchronization interval. Operating on delayed snapshots is legitimate and does not affect correctness. Nevertheless, the older the snapshot, the higher the probability of conflicts. In practice, a synchronization interval of 1 ms did not noticeably affect the overall abort rate (Section 6.3.3).

4.3 Life-cycle of a Transaction

This section describes the life-cycle of a transaction and enumerates the different transaction states:

1. **Begin:** In an initial step a transaction contacts the commit manager to retrieve its snapshot.
2. **Running:** While running on a PN, a transaction can access or update any data item. Read operations retrieve the item from the store, extract the valid version and cache it in case the item is re-accessed. Updates are buffered on the PN in the scope of the transaction. Data storage and buffering are detailed in Section 5.
3. **Try-Commit:** Before starting the commit procedure, a log entry with the ids of updated items is written to the transaction log. This is a requirement for recovery and fail-over (Section 4.4). We proceed with applying updates to the store using LL/SC operations. On success, the transaction commits. Otherwise, we abort.
4. (a) **Commit:** All data updates have been applied. Next, the indexes are altered to reflect the updates, and a commit flag is set in the transaction log. Finally, the commit manager is notified.
   (b) **Abort:** On conflict, applied data updates are rolled back. A transaction can also be aborted manually. In this case, no updates have been applied (as we skipped the *Try-Commit* state). Last, the commit manager is notified.

4.4 Recovery and Fail-Over

This section illustrates how node failures are addressed. Failures are detected by the management node using an eventually perfect failure detector based on timeouts. Although we describe fail-over for single node failures, handling multiple failures concurrently is supported.

4.4.1 Failure of a processing node

PNs act according to the crash-stop model. That is, in case of transient or permanent failures, all active transactions on a failed node are aborted. In particular, committing transactions with partially applied updates must be reverted to ensure correctness. As soon as a failure is detected, a recovery process is started to roll back the active transactions of the failed node. Correct recovery is enabled by a transaction log that contains the status of running transactions.

The transaction log is an ordered map of log entries located in the storage system. Before applying updates, a transaction must append a new entry to the log. Every entry is identified by the *tid* and consists of the PN id, a timestamp, the *write set*, and a flag to mark the transaction committed. The *write set* is a list of updated record ids.

When the recovery process is started, it 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 until the lowest active version number is reached. The *lav* implicitly acts as a rolling checkpoint. Once we encounter a relevant transaction, we use the *write set* and revert the changes made by the transaction. That is, the version with number *tid* is removed from the records. On completion of the recovery process, all transactions of the failed node have been rolled back. The management node ensures that only one recovery process is running at a time. However, a single recovery process can handle multiple node failures.

4.4.2 Failure of a storage node

The storage system must handle node failures transparently with regard to the processing nodes. In order to remain operational, PNs require data access and assume a highly available storage system. Obviously, a failure must not lead to data loss. Moreover, downtime must be minimized as it will cause transaction processing to be delayed. Availability is guaranteed by replicating data. As data is kept in volatile storage (main memory) a SN ensures that data is replicated before acknowledging a request. 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 enables on-going processing of requests. Eventually, the system re-organizes itself and restores the replication level. Ensuring high availability in simple record stores is a well studied field [12, 16].

The storage layer in Tell uses a management node to detect failures. The same node manages partitioning, restores the replication factor, and enables PNs 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

In a single commit manager configuration, a failure has system-wide impact. PNs are no longer able to start new transactions, and the system is blocked until a new commit manager is available. Once all active transactions at the moment of failure have committed (the commit manager is not required for completion), a new commit manager can be started. To restore its state, the commit manager retrieves the last used *tid* and the most recently committed transactions from the transaction log.

To prevent a single point of failure and increase availability, multiple commit managers can operate in a cluster. As pointed out previously, commit managers synchronize their state. If a commit manager becomes unavailable, PNs automatically switch to the next one. New commit managers can determine the current state from the data of the other managers. Recall that the commit managers regularly write their state to the storage system. Hence, state information is accessible independently of commit manager availability.

5. DATA ACCESS AND STORAGE

Tell provides a SQL interface and enables complex queries on relational data. The query processor parses incoming queries and uses the iterator model to access records. Records are retrieved and modified using basic data operations, such as *read* and *write*, and are stored in a key-value format. In this section, we explain how relational data is mapped to the key-value model and detail 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 one key-value pair. The key is a unique *record identifier (rid)*. *Rids* are monotonically incremented numerical values. The value field contains a serialized set of all the versions of the record. This row-level storage scheme is a significant design decision as it minimizes the number of storage accesses. For instance, with a single read operation, a transaction retrieves all versions of a record and can select the one that is valid according to the snapshot. On update, a transaction adds a new version to the record and writes back the entire record during commit. Again, a single atomic write request applies the update or identifies a conflict.

Grouping records into pages, as disk-oriented databases do in order to reduce the number of I/O operations [22], is of limited use in a shared-data architecture. A record needs to be re-fetched on every access because it can be changed anytime by remote PNs. Consequently, a coarse-grained storage scheme would not reduce the number of requests to the storage system but only increase network traffic. In contrast, a more fine-grained storage scheme (e.g., store every version as a key-value pair) would require additional requests to identify added versions. Committing transactions would have to check for new versions before applying updates and consequently conflict detection would become more expensive. Although storing single versions reduces network traffic, we opt for an approach that minimizes network requests.

To read a record, PNs rely on index structures. Indexes contain references to the *rid* and enable a PN to retrieve records with all versions (Figure 4). To update a record, it is first read. Next, a new version of the tuple reflecting the changes of the update is added to the set of versions. The entire record is kept in the transaction buffer, and further updates to the record directly modify the newly added version. Finally, the record is applied at commit time.

To further minimize the number of network requests and improve performance, Tell aggressively batches operations (i.e., several operations are combined into a single request). Batching enables transactions to access multiple records with a single request but it is also used to combine concurrent read operations from different transactions on the same PN. Batching is part of the interface to the store and therefore enables to combine requests across multiple transactions.

5.2 Mixed Workloads

OLTP workloads typically consist of short-running transactions that access few records. Every transaction requires a limited amount of remote accesses and utilizes low bandwidth. Thus, OLTP execution can be parallelized and scales with the number of PNs (Section 6). OLAP workloads, on the other hand, perform long running queries that access large amounts of data. For instance, an OLAP query might execute an aggregation that involves a full table scan (i.e., access all the records of a table). To process this query, a PN must read (and transfer over the network) all the records of the table. Obviously, processing this query has limitations with regard to latency and bandwidth utilization.

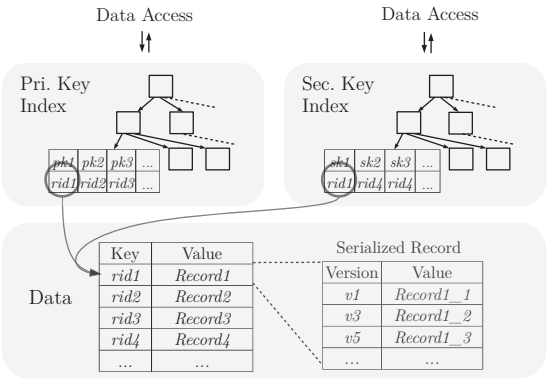

Figure 4: Storage and access of data records

To enable the efficient execution of mixed workloads, we propose to push down selected relational operators into the storage layer. For instance, executing simple operations such as selection or projection in the SN would enable to reduce the size of the result set and lower the amount of data sent over the network. A more advanced solution is to execute a shared scan over the entire data in the storage layer as suggested in [46, 59]. A dedicated thread could scan all stored records and pre-filter data for complex OLAP queries [44]. The challenge is to perform the scan efficiently without affecting on-going get/put operations. Mixed workloads are beyond the scope of this paper. Nevertheless, the concept of performing basic processing tasks in the storage layer is a promising direction for future work.

5.3 Latch-free Index Structures

Indexes provide an efficient lookup mechanism to identify matching records for a given attribute (or a set of attributes). Indexes can be accessed by multiple transactions in parallel and therefore require concurrency control. In a shared-data architecture, integrity must be maintained across nodes as an index can be modified by multiple PNs. Traditionally, index integrity is enforced using latches. However, latch-free algorithms have become popular recently [23]. In a distributed setting, the absence of latches is even more desirable as network communication increases latch acquisition cost. Moreover, the absence of latches ensures system-wide progress. Accordingly, Tell provides a latch-free B+tree.

5.3.1 B+Tree Index

The B+tree is one of the most used index structures in relational databases. Over the last decades, many variations have been optimized for concurrent access [21, 33]. Particularly noteworthy is the Bw-tree [35], a latch-free variant for modern hardware. Taking the techniques of the Bw-tree as a reference, we have implemented a latch-free B+tree concurrently accessible by multiple PNs.

The B+Tree index is entirely stored and maintained in the storage layer (every node is stored as a key-value pair) and can be accessed by all PNs. To synchronize index updates, every node in the tree is atomically modified in the storage system (using LL/SC operations). On lookup, a transaction accesses the root node and traverses the tree until the leaf-level is reached. B+tree nodes are cached in the processing

layer to improve traversal speed and minimize storage system requests. Caching works as follows: All index nodes with exception of the leaf level are cached. The leaf-level nodes are always retrieved from the storage system. If the range of the leaf node (lowest and largest value) does not match the data in its parent (i.e., the leaf node has been split or merged), then the parent nodes are recursively updated (i.e., the latest version is retrieved from the storage system) to keep the cache consistent. The caching mechanism is independent of the synchronization of index changes.

5.3.2 Design Considerations

In order to prevent unnecessary network requests and increase scalability, versioning information is excluded from all indexes. Instead of maintaining a key-reference entry for every existing version, only a single index entry per record is maintained. As a result, it is not necessary to insert new index entries on every record update (i.e., whenever a new version is added) but only when the indexed key is modified. Less index insertions improve index concurrency.

Version unaware indexes have limitations. In particular, it is not possible to identify for which versions an index entry is valid. In the absence of that information, a transaction may retrieve a record that is not valid according to its snapshot descriptor. Although these reads are unnecessary, they are performed independently of the index and do not affect index concurrency. In addition, index-only scans are not possible as the validity of an entry can not be verified.

5.4 Garbage Collection

Updates add new versions and over time records become larger. To prevent data items from growing infinitely, garbage collection (GC) is necessary. Garbage collection deletes the versions and index entries that are never going to be accessed again. We use 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 performs GC in a background task that runs in regular intervals (e.g., every hour). The latter approach is useful for rarely accessed records.

Record GC is part of the update process. Before applying a record, a transaction first verifies if older versions can be removed. To determine 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 the lav, we define C as the version number set 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 $max(C)$ is not garbage collected to guarantee that at least one version of the item always remains. All versions whose number is in G can be safely deleted before the record is written back to the storage system.

Index entries are also subject to garbage collection. Index GC is performed during read operations. During the read operation following an index lookup, a transaction verifies if some of the index entries are obsolete. 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 LL/SC operation fails, GC is retried with the next read. In the evaluation (Section 6) both GC strategies are used.

5.5 Buffering Strategies

In this section, we present three approaches to improve buffering in a shared-data architecture. The effectiveness of each strategy is illustrated in Section 6.7.

5.5.1 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, are buffered and reused in the scope of the transaction. Every transaction maintains a private buffer that caches all accessed records for the duration of the transaction's lifetime. This caching mechanism has obvious limitations: There is no shared buffer across transactions, and buffering is only beneficial if transactions access the same record several times.

5.5.2 Shared Record Buffer

The second approach is based on a shared record buffer 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 modified by remote PNs and new transactions need to access the storage system to retrieve the most recent record version, 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 an older snapshot).

This strategy is implemented using *version number sets* (as defined in Section 4.2). 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 for determining if the buffer entry can be used or if the transaction is too recent. If a transaction with version number set V_{tx} wants to read a record with version number set B, the following conditions are used to verify the validity of the buffer entry:

1. $V_{tx} \subseteq B$: The buffer is recent enough. We can access the record from the buffer and no interaction with the storage system is necessary.

2. $V_{tx} \not\subseteq B$: 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 retrieve the record from the storage system and replace the buffer entry. Finally, B 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 rational behind V_{max} is to keep B as big as possible to improve the likelihood that condition 1 holds for future data accesses. Once a record is retrieved from the buffer, we check if the versions contained in the record are valid according to V_{tx}. This step prevents *phantom* reads.

To insert a record into the buffer, we apply the procedure of condition 2. If the buffer is full a replacement strategy (e.g., Least-Recently-Used) is used to evict entries. Record updates are applied to the buffer in a *write-through* manner. Each time a transaction applies an update, the changes are

written to the storage system and if successful, to the buffer as well. B is set to the union of *tid* and V_{max}. V_{max} is a valid version number set for updated records because if a transaction in V_{max} would have changed the record, the LL/SC operation to the storage system would have failed. Buffer updates are atomic to ensure consistency.

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 of additional management overhead and increased memory consumption on the PNs.

5.5.3 Shared Buffer with Ver. 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 on version number sets of records. The advantage is that a PN can verify if a buffered record is valid by retrieving its version number set. If the buffer is not valid, the record is re-fetched. This strategy saves network bandwidth as version number sets are small compared to records.

Record updates are handled the same way as for 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. A buffered record is accessed according to the following conditions:

1. $V_{tx} \subseteq B$: The buffer entry is valid.
2. $V_{tx} \not\subseteq B$: The cache might be outdated. We fetch the record's version number set B' from the storage system.
   (a) If $B' = B$ the buffered record is still valid.
   (b) If $B' \neq B$ the record must be re-fetched. Moreover, B is replaced by B'.

Although this strategy reduces network traffic, it comes at the expense of additional update overhead. For each record update, two requests are performed in the storage layer.

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, multiple sequential records of a relational table are assigned to a cache unit. 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 less 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, records are more frequently invalidated and re-fetched.

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 are invalidated and eventually the update overhead outweighs saved requests.

6. EXPERIMENTAL EVALUATION

This section presents an evaluation of Tell. We first describe environment and methodology before presenting experimental results. Our experiments are based on the TPC-C benchmark that is popular both in industry and academia.

6.1 Implementation and Environment

The PN in Tell is implemented in 15,608 lines of C++ code. Transaction management and data access are performed with the techniques described in Sections 4 and 5 respectively. PNs have a synchronous processing model. That is, a thread processes a transaction at a time. While waiting for a I/O request to complete, another thread takes over.

PNs interact with the RamCloud storage system (RC) [43]. 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* and *Put* operations (i.e., LL/SC). The key space is range-partitioned to distribute load across SNs. RC supports remote backup with fast recovery [42]. The backup mechanism synchronously replicates every *put* operation to the replicas and thereafter asynchronously writes it to persistent storage. The replication factor (RF) specifies the number of data copies. RC uses replication for fault-tolerance only. All requests to a particular partition are sent to the master copy.

The benchmarking 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 units that consist each of one processor and half the memory. A process is by default assigned to one NUMA unit. This assignment allows to run two processes per server (usually a PN and a SN) and thus doubles the maximum number of logical nodes (24). The servers are connected to a 40 Gbit QDR InfiniBand network. All nodes use 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. Load is generated by *terminal* clients. Every terminal operation results in one database transaction. Terminals are run separately from the system under test (SUT). Our TPC-C implementation differs from the specification (Rev. 5.11) [57]. Specifically, we have removed wait times so that terminals continuously send requests to the PNs. The number of terminals threads is selected so that the peak throughput of the SUT is reached.

The size of the database is determined by the number of warehouses (WH). The default population for every TPC-C run is 200 WHs. In our implementation, a warehouse occupies approximately 189 MB of RamCloud memory space.

The standard TPC-C transaction mix is write-intensive with a write ratio of 35.84% The throughput metric is the new-order transaction rate (TpmC). That is, the number of successfully executed new-order transactions per minute. Not included are failed transactions (i.e., aborted or taking too long). The TpmC represents around 45% of all issued transactions. The TPC-C standard mix inserts large amounts of data (i.e., at high throughput up to 30 GB/min) and restricted us to limit the measurement interval to 12 minutes for every TPC-C run. The benchmark is executed 5 times for each configuration and the presented results are averaged over all runs. The measured performance was predictable and the variations were very low.

To evaluate read-intensive scenarios, we propose an additional TPC-C mix. This mix consists of three transactions and provides a read-ratio of 95.11%. As we change the percentage of new-order transactions, throughput is measured in number of transactions per second (Tps). Table 2 gives an overview of both workload mixes.

Workload Mix	Write Ratio	Throughput Metric	Transaction Mix				
			New-Order	Payment	Delivery	Order Status	Stock Level
Write-Intensive (Standard)	35.84%	TpmC	45%	43%	4%	4%	4%
Read-Intensive	4.89%	Tps	9%	0%	0%	84%	7%

Table 2: Write and read-intensive workloads for the TPC-C benchmark

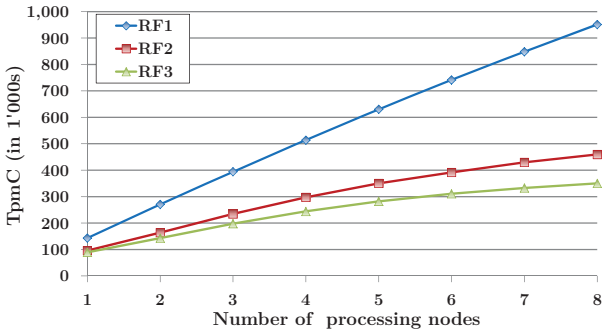

Figure 5: Scale-out processing (write-intensive)

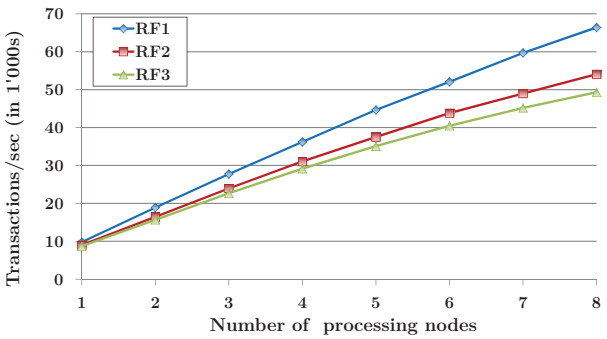

Figure 6: Scale-out processing (read-intensive)

Commit Managers	1	2	3
TpmC	958'187	955'759	955'608
Tx abort rate (%)	14.75	15.59	15.91

Table 3: Commit managers (write-intensive)

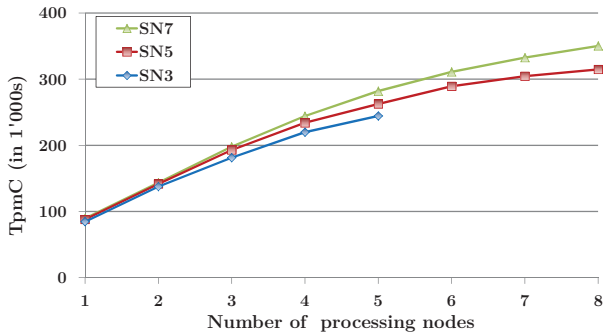

Figure 7: Scale-out storage (write-intensive)

6.3 Scale-Out

This section studies scale-out behavior and highlights that Tell can scale with an increasing number of nodes. We conducted experiments in which the number of PNs, SNs, and commit managers is varied individually and analyzed the effect on throughput.

6.3.1 Processing

Figure 5 presents results for the standard TPC-C mix. For this experiment, we increase the number of PNs. All other components have enough resources and do not limit performance (7 SNs, 1 commit manager).

With no replication (RF1), throughput increases with the number of processing resources from 143,114 TpmC with 1 PN to 958,187 TpmC with 8 PNs. The TPC-C suffers from data contention on the *warehouse* table and therefore throughput does not increase linearly. Contention is reflected in the overall transaction abort rate (for all TPC-C transactions) that grows from 2.91% (1 PN) to 14.72% (8 PNs).

Increasing the replication factor adds overhead as updates are synchronously replicated. With RF3, we reach a peak throughput of 350,257 TpmC with 8 PNs, thus 63.2% less than with RF1. Synchronous replication increases latency and consequently affects the number of transactions a worker thread can process. In the 8 PN configuration, the average transaction response time increases from 10.69 ms with RF1 to 29.67 ms with RF3. Increasing the number of processing threads to compensate for higher access latencies has no effect because of excessive context switching on the PNs. The transaction abort rate with RF3 is similar to the one with RF1 as the probability of conflict is alike (higher processing time compensates for lower throughput). Increasing data contention causes more aborts and throughput decreases. For example, with 10 WHs throughput reaches 341,917 TpmC (8 PNs, RF3).

Figure 6 shows the results of the same experiment but running the read-intensive TPC-C mix. As read operations only retrieve records from the master copy (and do not require interaction with the replicas), their latency is not affected by an increase in the replication level. Consequently, the higher the read-ratio of a workload, the less the performance impact of replication. Figure 6 illustrates this observation. With RF3 and 8 PNs, the throughput is 25.7% lower than with RF1, a considerable improvement compared to the write-heavy scenario.

Figures 5 and 6 emphasize that Tell enables scalable OLTP processing with an increasing number of processing resources. The performance loss due to synchronous replication under write-intensive workloads is a common effect related to replication as a technique that is not specific to the shared-data architecture or the RamCloud implementation.

6.3.2 Storage

Figure 7 shows the scale-out of the storage layer. We execute the TPC-C standard mix with RF3 on three storage configurations (3, 5, and 7 SNs) and measure the system throughput with increasing load. In all configurations the storage layer is not a bottleneck, and therefore, the throughput difference is minimal. The configuration with 3 SNs can not run with more than 5 PNs. With 6 PNs, the benchmark generates too much data to fit into the combined memory capacity of 3 SNs. Consequently, the storage resources in a cluster should be determined by the required memory capacity and not by the available CPU power.

6.3.3 Commit Manager

Table 3 shows the performance impact of running several commit managers (with 8 PNs, 7 SNs, and RF1). Commit managers use a simple protocol to assign unique *tids* and keep track of completed transactions (Section 4). In

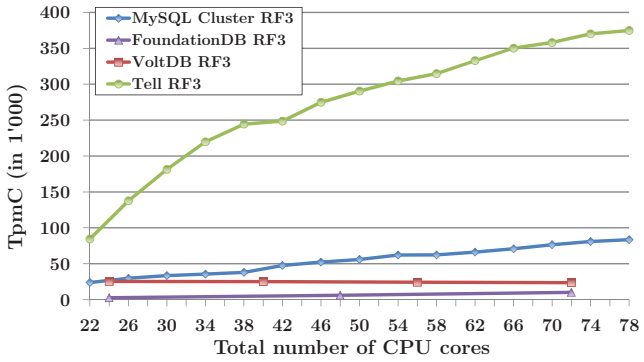

Figure 8: Throughput (TPC-C standard)

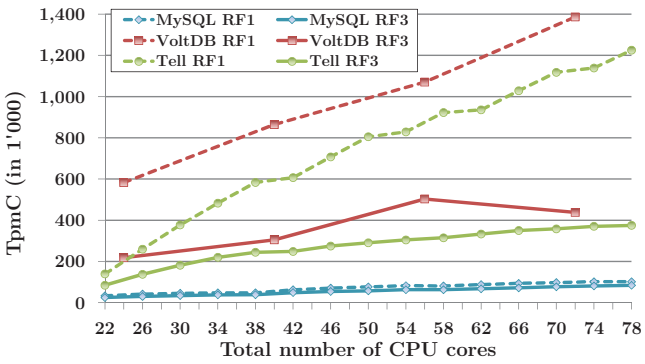

Figure 9: Throughput (TPC-C shardable)

		Small 22-24cores (mean ± σ, ms)	Large 70-72cores (mean ± σ, ms)
Standard	Tell	14 ± 17	32 ± 41
	MySQL Cluster	34 ± 26	43 ± 40
	VoltDB	706 ± 2159	655 ± 1875
	FoundationDB	149 ± 183	120 ± 138
Shard	Tell	14 ± 17	32 ± 41
	VoltDB	62 ± 102	22 ± 59

Table 4: TPC-C transaction response time

particular, they are not required for complex tasks such as commit validation. The state among commit managers is synchronized using the approach described in Section 4.2. Table 3 shows that running the TPC-C benchmark with a synchronization delay of 1ms caused no significant impact on throughput and on the transaction abort rate. Consequently, the commit manager component is not a bottleneck.

6.4 Comparison to Partitioned Databases

This section evaluates Tell against two state-of-the-art partitioned databases, VoltDB and MySQL Cluster:

VoltDB [54] is an in-memory relational database that partitions data and serially executes transactions on each partition. The more transactions can be processed by single partitions, the better VoltDB scales. Consequently, we use single-partition transactions whenever possible. Transactions are implemented as stored procedures that are precompiled to optimize execution. VoltDb supports replication (called K-factor) and enables to asynchronously write a transaction log to the SSDs (VoltDB command log). As a result, it provides similar durability guarantees as RamCloud. VoltDB Enterprise 4.8 is run in configurations of 1, 3, 5, 7, 9, and 11 nodes (8 cores each) with 6 partitions per node (as advised in the official documentation). The nodes communicate with TCP/IP over InfiniBand. Our implementation follows the performance related recommendations described in [60]. In particular, the terminals invocate transactions asynchronously and use multiple node connections.

MySQL Cluster [41] is another partitioned database with an in-memory storage engine (i.e., NDB). A cluster configuration consists of three components: *Management nodes* (MN) 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 DNs. MySQL Cluster synchronously replicates data. Our benchmark environment runs MySQL Cluster 7.3.2 and uses the InfiniBand network as well. The benchmark is executed with 3, 6, and 9 DNs. We vary the number of SQL nodes and use two MNs. Terminals use prepared statements to execute transactions against the SUT.

With regard to partitioned databases, the data model and the workload require careful consideration. The TPC-C data model is ideal for sharding. Most tables reference the warehouse id that is the obvious partitioning key. The read-only item table can be fully replicated across all partitions. The TPC-C workload is more problematic. According to the TPC-C specification [57], remote new-order (clause 2.4.1.8) and remote payment (clause 2.5.1.6) transactions access data from several warehouses and therefore require cross-partition transactions. In the TPC-C standard mix, the ratio of cross-partition transactions is about 11.25%.

Figure 8 compares VoltDB and MySQL Cluster to Tell. The figure presents the peak TpmC values for the TPC-C standard mix with RF3 on a 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. For instance, a minimal VoltDB cluster consists of 3 nodes (24 cores). The minimal Tell configuration with 22 cores consists of 1 PN (4 cores), 3 SNs (12 cores), 2 commit managers (4 cores), and 1 MN (2 cores). In comparison to Tell, VoltDB and MySQL Cluster do not scale with the number of cores. While Tell reaches a throughput of 374,894 TpmC with 78 cores, MySQL Cluster and VoltDB achieve 83,524 and 23,183 TpmC respectively. The scalability of the partitioned databases is presumably limited by distributed transactions that require coordination across nodes. In particular, VoltDB suffers from cross-partition transactions as throughput decreases the more nodes are added. This observation is emphasized by VoltDB's high latency (Table 4). MySQL Cluster is slightly faster than VoltDB because single-partition transactions are not blocked by distributed transactions. The experiment highlights that the techniques presented in this work efficiently interact. Moreover, the shared-data architecture enables much higher performance than partitioned databases for OLTP workloads that are not fully shardable.

In a second experiment, we evaluate Tell given a workload that is optimized for partitioned databases. To that end, we modify the TPC-C benchmark and remove all cross-partition transactions. This alternative workload, called *TPC-C shardable*, replaces remote new-order and payment transactions with equivalent ones that only access one WH.

Figure 9 compares Tell to VoltDB and MySQL Cluster using the TPC-C shardable with RF1 and RF3. Again, we

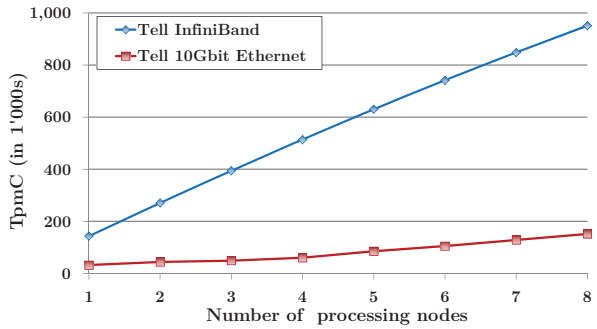

Figure 10: Network (write-intensive)

	TpmC	Latency (mean ± σ, ms)	TP99 (ms)	TP999 (ms)
InfiniBand	958,187	10.69 ± 13.02	76.48	100.62
10Gb Ethernet	151,632	69.41 ± 87.99	542.03	644.15

Table 5: Network latency (write-intensive)

vary the number of cores. With this workload VoltDB fulfills the scalability promises and achieves more throughput than Tell. The peak performance of VoltDB with RF1 is 1.387 Mio TpmC. Tell achieves 1.225 Mio TpmC, thus 11.7% less than VoltDB. In addition, a shardable workload results in much better latency for VoltDB (Table 4). MySQL Cluster is only 1-2% faster than with the standard workload. The numbers emphasize that the shared-data architecture enables competitive OLTP performance. Even with a perfectly shardable workload, the achieved throughput is in the same ballpark as state-of-the-art partitioned databases.

6.5 Comparison to Shared-Data Databases

Section 3 introduced FoundationDB [19], another shared-data database. FoundationDB is known for its fast key-value store and has recently released a "SQL Layer" that allows for SQL transactions on top of the key-value store. Our environment runs FoundationDB 3.0.6 with in-memory data storage and RF3 (redundancy mode triple). The benchmark is executed on configurations with 3, 6, and 9 node in both layers. The smallest configuration (24 cores) achieves 2,706 TpmC and the largest (72 cores) reaches 10,047 TpmC. Although FoundationDB scales with the number of cores, the throughput is more than a factor 30 lower than Tell (Figure 8). FoundationDB has not published many implementation details. For instance, it is not known how commit validation occurs and to what extent the native benefits of InfiniBand are used. Moreover, the SQL Layer is still new and we believe that FoundationDB will work on improving the performance of the SQL Layer in the near future. Nevertheless, these results indicate that if not done right, shared-data systems show very poor performance.

6.6 Network

Throughout the paper, we have emphasized that fast data access is a major scalability requirement. In this experiment, we highlight the importance of low-latency data access in a shared-data architecture. Figure 10 compares the throughput using an InfiniBand network to the performance of 10 Gb Ethernet. The network is used for communication between PNs and SNs as well as among SNs. In the experiment, we vary the number of PNs. The RF is 1 and the number of SNs is 7. The TpmC results on InfiniBand are more than six times higher than the results achieved with Ether-

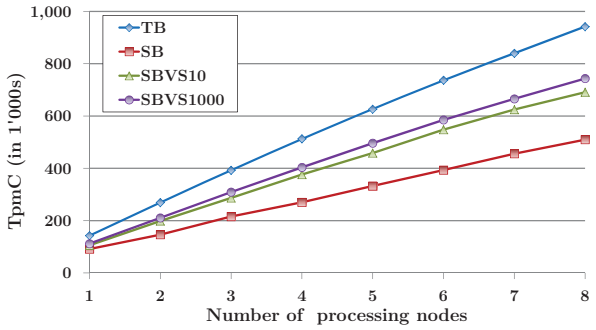

Figure 11: Buffering strategies (write-intensive)

net independent of the number of PNs. 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. As a result, data can be accessed faster and processing time decreases. As Tell uses a synchronous processing model where transactions block until data arrives, reducing latency has a positive effect on throughput. Table 5 summarizes the results for the fastest configuration with 8 PNs. The second column shows the mean transaction response time and the standard deviation. Both values reflect the measured difference in throughput. The last two columns show the 99th and 99.9th percentile response time. The low number of outliers indicates that both networks are not congested. In the InfiniBand configuration, the total bandwidth usage of one SN is 125.9 MB/s (in and out). Thus, the network is not saturated.

6.7 Buffering Strategies

Figure 11 shows a comparison of the buffering strategies presented in Section 5.5. The configuration uses 7 SNs and RF1. 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 had a considerably higher cache hit ratio (37.37% for SBVS1000). Nevertheless, this could not compensate for the cost of additional update requests to the storage system. A key insight is that with fast RDMA the overhead of buffering data does not pay off for workloads such as the TPC-C.

7. CONCLUSIONS

In this paper, we described Tell, a database system based on a shared-data architecture. Tell decouples transactional query processing and data storage into two layers to enable elasticity and workload flexibility. Data is stored in a distributed record manager that is shared among all database instances. To address the synchronization problem and enable scalable performance with present the right combination of specific techniques coupled with modern hardware.

The experimental evaluation highlighted the ability of the shared-data architecture to scale out with the number of cores. Furthermore, we compared Tell to alternative distributed databases and achieved a higher throughput than VoltDB, MySQL Cluster, and FoundationDB in the popular TPC-C benchmark. The shared-data architecture enables competitive performance while at the same time being elastic and independent of any workload assumptions.

8. REFERENCES

[1] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. SOSP'07, pages 159–174, 2007.

[2] Apache Hadoop. http://hadoop.apache.org/. Nov. 06, 2014.

[3] 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.

[4] 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.

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

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

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

[8] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. SIGMOD'08, pages 251–264, 2008.

[9] M. Cahill, U. Röhm, and A. Fekete. Serializable isolation for snapshot databases. SIGMOD'08, pages 729–738, 2008.

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

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

[12] 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.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. OSDI'12, pages 251–264, 2012.

[14] 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.

[15] S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: an elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1):5:1–5:45, 2013.

[16] 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.

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

[18] 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.

[19] FoundationDB. https://foundationdb.com/. Feb. 07, 2015.

[20] D. Gomez Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: lock-free transactional support for distributed data stores. ICDE'14, pages 676–687, 2014.

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

[22] J. Gray and A. Reuter. *Transaction processing: concepts and techniques.* Morgan Kaufmann, 1992.

[23] T. Horikawa. Latch-free data structures for DBMS: design, implementation, and evaluation. SIGMOD'13, pages 409–420, 2013.

[24] InfiniBand. http://www.infinibandta.org/. Nov. 06, 2014.

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

[26] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. SIGMOD'10, pages 603–614, 2010.

[27] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. VLDB'07, pages 1263–1274, 2007.

[28] 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.

[29] 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.

[30] A. Kemper and T. Neumann. HyPer: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. ICDE'11, pages 195–206, 2011.

[31] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[32] 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.

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

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

[35] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-tree: a B-tree for new hardware platforms. ICDE'13, pages 302–313, 2013.

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

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

[38] M. Mages. ABA prevention using single-word instructions. Technical Report RC23089 (W0401-136), 2004.
[39] C. Mohan. History repeats itself: sensible and Nonsen aspects of the NoSQL hoopla. EDBT'13, pages 11–16, 2013.
[40] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. ScyPer: elastic OLAP throughput on transactional data. DanaC'13, pages 11–15, 2013.
[41] MySQL Cluster. http://www.mysql.com/products/cluster/. Nov. 06, 2014.
[42] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. SOSP'11, pages 29–41, 2011.
[43] 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.
[44] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. ICDE'08, pages 60–69, 2008.
[45] 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.
[46] W. Ronald. A technical overview of the Oracle Exadata database machine and Exadata storage server. Technical Report Oracle White Paper, 2012.
[47] M. Rys. Scalable SQL. *Commun. ACM*, 54(6):48–53, 2011.
[48] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proc. VLDB Endow.*, 7(12), 2014.
[49] J. Shute, R. Vingralek, B. Samwel, et al. F1: a distributed SQL database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, 2013.
[50] A. Singhal, R. Van der Wijngaart, and P. Barry. Atomic read modify write primitives for I/O devices. Technical Report Intel White Paper, 2008.
[51] G. H. Sockut and B. R. Iyer. Online reorganization of databases. *ACM Comput. Surv.*, 41(3):14:1–14:136, 2009.
[52] M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, 54(6):72–80, 2011.
[53] 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.
[54] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.
[55] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. ElmoreA, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3), 2014.
[56] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. SIGMOD'12, pages 1–12, 2012.
[57] Transaction Processing Performance Council (TPC). TPC Benchmark C Specification ver. 5.11, 2010.
[58] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. SOSP'13, pages 18–32, 2013.
[59] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.*, 2(1):706–717, 2009.
[60] VoltDB. http://www.voltdb.com/. Nov. 06, 2014.
[61] M. Yabandeh and D. Gómez Ferro. A critique of snapshot isolation. EuroSys'12, pages 155–168, 2012.