ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud

Sudipto Das Shashank Agarwal Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara, CA, USA
{sudipto, shashank, agrawal, amr}@cs.ucsb.edu

ABSTRACT

Cloud computing has emerged as a pervasive platform for deploying scalable and highly available Internet applications. To facilitate the migration of data-driven applications to the cloud: *elasticity*, *scalability*, *fault-tolerance*, and *self-manageability* (henceforth referred to as *cloud features*) are fundamental requirements for database management systems (DBMS) driving such applications. Even though extremely successful in the traditional enterprise setting – the high cost of commercial relational database software, and the lack of the desired cloud features in the open source counterparts – relational databases (RDBMS) are not a competitive choice for cloud-bound applications. As a result, *Key-Value* stores have emerged as a preferred choice for scalable and fault-tolerant data management, but lack the rich functionality, and transactional guarantees of RDBMS. We present ElasTraS, an **Elas***tic* **Tra***n***S***actional relational database*, designed to scale out using a cluster of commodity machines while being *fault-tolerant* and *self managing*. ElasTraS is designed to support both classes of database needs for the cloud: (i) large databases partitioned across a set of nodes, and (ii) a large number of small and independent databases common in *multi-tenant* databases. ElasTraS borrows from the design philosophy of scalable *Key-Value* stores to minimize distributed synchronization and remove scalability bottlenecks, while leveraging decades of research on transaction processing, concurrency control, and recovery to support rich functionality and transactional guarantees. We present the design of ElasTraS, implementation details of our initial prototype system, and experimental results executing the TPC-C benchmark.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Concurrency, Transaction processing*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Algorithms, Design, Performance.

Keywords

Cloud computing, Elastic Data Management, Transactions, ACID, Scalability, Fault-tolerance.

1. INTRODUCTION

Cloud computing has emerged as a pervasive paradigm for hosting Internet scale applications in large computing infrastructures. Major enabling features of the cloud include *elasticity of resources*, *pay per use*, *low time to market*, and the *perception of unlimited resources and infinite scalability*. As a result, there has been a widespread migration of IT services from enterprise computing infrastructures (i.e., networked cluster of expensive servers) to cloud infrastructures (i.e., large data-centers with hundreds of thousands of commodity servers). Since one of the primary uses of the cloud is for hosting a wide range of web applications, scalable data management systems that drive these applications are a crucial technology component in the cloud. Relational databases (RDBMS)[1] have been extremely successful in the enterprise setting for more than two decades, but data management in the cloud requires DBMS's to be *elastic*, *scalable*, *fault-tolerant*, *self managing* and *self-recovering* – features which most RDBMS's lack.[2] Furthermore, the *elasticity* and *pay-per-use* model of infrastructure in the cloud poses a new challenge for a cloud DBMS which now has to also optimize its cost of operation [25]. This facet is not important in an enterprise setting where upfront investments are made on the infrastructure, and the goal is to optimize performance for a given infrastructure. System designers for the cloud therefore favor *elastic Key-Value* stores [10–12, 15, 21]. Even though these systems have the desired cloud features, they provide minimal consistency guarantees and significantly reduced functionality – thereby placing unprecedented burden on the application developers who often lack the expertise to deal with such intricacies [20, 31]. Therefore a huge chasm exists in terms of data management functionality in the cloud: on one hand are the RDBMSs which support rich functionality and guarantee transactional access to multiple tables but lack the cloud features, while on the other hand are *Key-Value* stores which possess the cloud features but provide minimal consistency guarantees and single row based access functionality.

Even before the advent of cloud computing, Internet application designers had articulated the need for scalable databases. Various ad-hoc approaches have been used to scale databases in practice. Popular approaches like Master-Slave configurations, replication, or use of object caches (such as Memcached) work well for read-intensive workloads, but are not suited for update-intensive workloads. Partitioning (or sharding) a database has been used as a primary method for providing scalability of updates. Though extremely useful, large partitioned database installations are often hard to manage, and partitioning supported by traditional RDBMS's is often fraught with various problems (refer to Section 3 for more details). Furthermore, existing partitioning approaches primarily address database scalability; the issues of elasticity, fault-tolerance, load balancing, and the ability to automatically recover from failures remain largely unaddressed. A DBMS in the cloud should

[1]RDBMS refers to relational databases like MySQL etc., while DBMS refers to the general class of data stores, including non-relational systems.
[2]Some commercial systems (like Oracle RAC and MS SQL Azure) support some of these features, but no solution exists in the open source domain.

address all of these issues of *elasticity, scalability, fault-tolerance, and self-manageability, while providing transactional guarantees which can be used to build useful applications.*[3]

We propose ElasTraS, an **Elas***tic* **Tra***n***S***actional relational database* for the cloud which is *scalable* and *elastic* to adapt to varying loads, while being *fault-tolerant* and *self managing* to eliminate the administration overhead often attributed to a large RDBMS installation. ElasTraS has been designed to use the Infrastructure as a Service (IaaS) abstraction of cloud services to provide data management functionality. ElasTraS can handle both types of databases commonly found in the cloud: (i) large database instances whose size ranges are in the order of tens of terabytes and are partitioned across a cluster of nodes [4, 36]; and (ii) a large number of small databases commonly observed in a *multi-tenant* DBMS where each database is small and corresponds to independent clients, but thousands of such databases are managed by the same system. Conceptually, ElasTraS operates at the granularity of *partitions* – for (i), each large database instance consists of hundreds or thousands of partitions, and for (ii), each partition is in itself a self-contained database, and there are hundreds to thousands of such independent single-partition databases. Database partitions in ElasTraS are the granules for load-balancing, elasticity, fault-tolerance, and transactional access. Scalability and high availability is achieved by limiting update transactions to a single partition. For the multi-tenant case, ElasTraS can provide full transactional RDBMS functionality, while for large database instances, ElasTraS relies on schema level partitioning of the database to support rich functionality even though transactions are limited to single partitions. This approach solves a majority of problems with current partitioning approaches while leveraging partitioning to scale out on a cluster of commodity servers. In [13], we presented a short preliminary overview of ElasTraS, and in this paper, we present the rationale, detailed design, and implementation of the system. Kraska et al [25] proposed a framework for "rationing" consistency for optimizing the operational cost. Our design rather maintains consistency as the constant feature, and leverages from peaks and troughs in usage to deliver elastic scaling to optimize cost and resource utilization.

ElasTraS is designed to support a relational data model and foreign key relationships amongst tables in a database. Given a set of partitions of the same or different databases, ElasTraS can: (i) deal with workload changes resulting in the growing and shrinking of the cluster size, (ii) load balance the partitions, (iii) recover from node failures, (iv) if configured for dynamic partitioning, then dynamically split hot or very big partitions, and merge consecutive small partitions using a partitioning scheme specified in advance, and (v) provide transactional access to the database partitions – all without any human intervention; thereby considerably reducing the administrative overhead attributed to partitioned databases, while scaling to large amounts of data and large numbers of concurrent transactions. Note that ElasTraS is not designed to break the petabyte or exabyte barrier and scale to many thousands of servers while providing transactional access – only a handful of enterprises deal with such huge scale, and specialized solutions are often needed to cater to their specific requirements. Rather, ElasTraS has been designed to scale to tens of terabytes of data, and a few tens of commodity servers – a range that encompasses a majority of the applications projected to move to the cloud, but lack a scalable transactional RDBMS that suits their requirements.

We present the design of ElasTraS, provide implementation details of a prototype system, and demonstrate the feasibility of the design by experiments based on the standard TPC-C benchmark [34] workload in a cloud infrastructure. For large database instances, we demonstrate how schema level partitioning can be used to design practical applications while limiting transactional access to a single partition. By executing all transactions in the TPC-C benchmark, we demonstrate that the restriction of limiting transactions to a single partition is not a substantial restriction, and practical database applications can be built using the supported set of operations.

[3]Transactional refers to the ACID properties of database transactions [35].

Contributions

- We identify and articulate the design principles which can be used in designing scalable data management systems with the cloud features while providing transactional guarantees.
- We propose the design of a system suitable for large single tenant databases as well as multi-tenant databases with a large number of small databases. For large single tenant databases, we demonstrate how schema level partitioning can be used to limit transactions to a single partition.
- We provide an implementation design of ElasTraS and use the TPC-C benchmark [34] for performance evaluation in Amazon's cloud. An ElasTraS deployment on a 30 node cluster with over 1 TB of data served thousands of concurrent clients while sustaining a throughput of more than 0.2 million transactions per minute.

2. DESIGN OF ELASTRAS

ElasTraS is designed to handle both large single-tenant databases, as well as a large number of small databases for multiple tenants [36]. For a multi-tenant database, a tenant's database is typically small and fits into a single database partition, while in a large single-tenant deployment, a database can span multiple partitions. Henceforth in this paper, the term *partition* either refers to a partition of a database, or a database for a tenant that fits completely in a partition. We now explain the design of ElasTraS assuming that a set of partitions are provided. We remark that the design of ElasTraS is not dependent on any partitioning scheme, and the partitioning technique used is an implementation detail.

2.1 Desiderata for a Cloud bound DBMS

A DBMS deployed in the cloud warrants the following features:

SCALABILITY. The Cloud provides the illusion of infinite resources which allows applications to easily scale out using the cloud. Hence a DBMS deployed in the cloud should be able to scale both with data as well as with the number of users.

ELASTICITY. The *pay-per-use* model of Cloud and the ability to dynamically add and remove physical resources (storage and computing) on-demand, lends *elasticity* to the cloud. Systems deployed in the cloud can therefore exploit this elasticity to dynamically expand or shrink as per the observed load characteristics. This elasticity allows considerable savings in terms of resources and operational cost when compared to an under-utilized static system that over-provisions for peak load.

FAULT-TOLERANCE. As a result of the scale of the Cloud infrastructures, and the use of commodity components, failures in the cloud are a norm rather than an exception. Hence, systems for the cloud should be designed to handle failures in order to remain operational (perhaps at a reduced level) in the presence of failures.

Transactional Access. RDBMS provide atomic, consistent, and durable access to data at various granularities which allows the developers to reason about the data and correctness of the applications, which further results in simplification of the design of the applications. Therefore, support for transactions is a desired feature for a DBMS in the cloud.

The above features also enable monitoring the deployed systems for workload imbalances and failures, and dynamically load balance heavily loaded servers or dynamically instantiate servers to replace failed servers while minimizing the need for human intervention.

2.2 Design Rationale

Our approach to infuse cloud features in an RDBMS is grounded on the same principles that have resulted in virtually unlimited scalability in *Key-Value* stores. *Key-Value* stores (e.g., [11,12,15]) treat a single key (or row) as the fundamental unit of access granularity. Databases, on the other hand, treat the entire database as a single unit (from a user's perspective) and rely on concurrency control protocols to enable concurrent data accesses. Although application developers can potentially access the entire database within a single transaction, in practice this is rarely the case. Rather, most data accesses are localized and access only a few related data items potentially on multiple tables. Based on these observations, we summarize the design rationales of ElasTraS which are similar to *Key-Value* stores, specifically Bigtable [11]:

- **Separate System and Application State** We refer to the meta information critical for the proper functioning of the system as the *system state*, and the actual data stored in the DBMS as the *application state* [1]. For instance, in a dynamically partitioned database, the mapping of partitions to servers (i.e. catalog tables) is a part of the *system state*. The *system state* is critical and needs stringent consistency guarantees, but is orders of magnitude smaller and less dynamic than the *application state*. On the other hand, the *application state* requires varying degrees of *consistency* and operational flexibility, and hence can use different and potentially less stringent protocols for guaranteeing these requirements. This separation allows for the use of different classes of protocols and techniques when dealing with the different components of the system. Protocols with stringent consistency and durability guarantees (examples include use of Paxos [27]) ensure safety of the *system state* without causing a scalability bottleneck, while the *application state* is maintained using less expensive protocols.

- **Limited distributed synchronization is practical.** Use of light-weight coarse grained locking for loose coupling of the nodes allows a light weight synchronization mechanism that scales to a large number of nodes, while obviating expensive distributed synchronization protocols. Our design uses *timed leases* to ensure mutual exclusion for critical operations and to deal with node failures.

- **Limit interactions to a Single Node.** Limiting operations to a single node allows the system to horizontally partition the load as well as data [1, 23]. In addition, the failure of a component in the system does not affect the operation of the remaining components, and allows for graceful degradation of performance in the presence of failures while obviating distributed synchronization for the *application state*. This is achieved by limiting update transactions to access a single database partition and hence a single node.

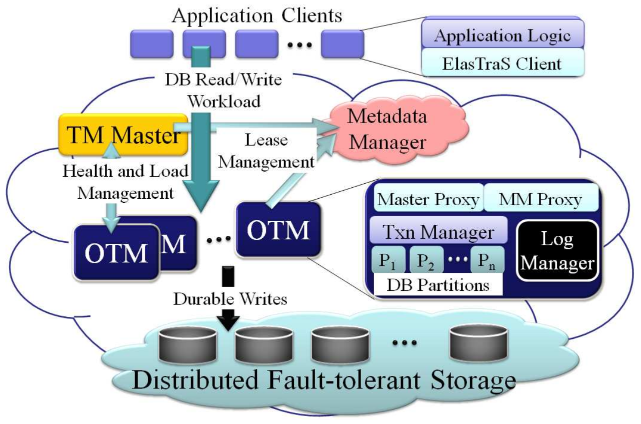


Figure 1: ElasTraS architecture overview.

- **Partition a database at the schema level.** Most RDBMS applications have a database schema involving multiple tables, and transactions often access multiple tables for related data tuples. Partitioning at the database schema level – unlike table level partitioning as supported by most RDBMS– allows related data fragments from multiple tables to be collocated in a single database partition. This collocation ensures support for a rich class of transactions while only accessing a single partition.

- **Decouple Ownership from Data Storage.** Ownership refers to the exclusive rights to access and modify data. Partitioning ownership therefore effectively partitions the data, while decoupling the storage of data from ownership minimizes data movement and allows a lightweight transfer of ownership when moving partitions for fault-tolerance, load balancing, and elasticity. This is achieved by storing data in a decoupled distributed storage, and partitioning ownership amongst transaction managers.

2.3 Design and Architecture Overview

Figure 1 provides a high-level architectural overview of ElasTraS. At the bottom of the stack is the distributed storage, which acts as a fault-tolerant durable storage and decouples storage of data from the ownership. At the heart of the system are the *Owning Transaction Managers (OTM)* which *own* one or more partitions and provide transactional guarantees on them. The *TM Master* of the system is responsible for monitoring the OTMs, recovering from node failures, and performing load balancing and elastic scaling. The *Metadata Manager (MM)* maintains the *system state*, which comprises of a mapping of the database partitions to their owners, and the *leases*. Leases granted by the MM to the TM master as well as the OTMs ensure mutual exclusion in critical system tasks, and provide low overhead distributed synchronization. At the top of the stack is the ElasTraS *client library* which applications link to, and is designed to hide the complexity of locating partitions in the presence of various system dynamics. We now provide a conceptual description of the different components of the system, and the guarantees provided by each component.

Distributed Fault-tolerant Storage (DFS): The Distributed Fault-tolerant Storage (DFS) is a consistent, append-only, replicated storage manager which can tolerate node failures (mostly single site failures, and multiple site failures in certain cases) while ensuring durability of writes. This abstraction of the storage layer provides fault-tolerance while allowing quick recovery from OTM failures, without the actual OTM node recovering. The storage layer takes care of replication and fault-tolerance, while the OTMs ensure that the same object or file is not being written to concurrently. We assume that the storage layer provides consistency guarantees for reads and writes. Since all the nodes in ElasTraS are located within the same infrastructure and are connected by a low latency and high throughput network, such an assumption is reasonable and is supported by present storage systems such as HDFS [22]. But such guarantees come at the cost of expensive writes to the DFS, and hence ElasTraS minimizes the number of DFS accesses.

Owning Transaction Managers (OTM): An OTM executes transactions, guarantees ACID properties, and performs concurrency control and recovery on partitions. Each OTM is granted exclusive access rights to the disjoint partitions which it *owns*. An OTM is assigned *ownership* of multiple database partitions, but treats each partition independently – which and how many partitions an OTM serves is determined by the *TM master*, and the design of an OTM is not dependent on this decision. Each OTM serves about tens to hundreds of partitions depending on the load on the system. It relies on the *metadata manager* to ensure that it has exclusive access to a partition. An OTM caches the contents of the partitions it owns, thereby preventing expensive accesses to the DFS layer. Exclusive ownership allows caching of updates and the content of a partition at an OTM without any cross OTM synchronization, or expensive cache coherence protocols, while being able to provide strong transactional guarantees at a low cost. To deal with OTM failures and guarantee durability in the presence of failures, the transaction commit log of an OTM is stored in the DFS, and the log entries are “forced” to the DFS on transaction commit.

Figure 1 also shows the components of an OTM: the transaction manager executes the transactions on a partition, and the log manager is responsible for guaranteeing the durability of committed transactions while efficiently managing expensive accesses to the DFS. In addition, an OTM also houses a number of proxies for communicating with other components in the system.

TM Master: The *TM Master* is responsible for monitoring and managing the system and performing the necessary corrective actions for the proper functioning of the OTMs in the presence of failures. The responsibilities of the TM master include: assigning partitions to OTMs, partition reassignment for load balancing and elasticity, and detecting and recovering from OTM failures. To ensure correctness of the operations and safety of the system, there is only a single active TM master in an ElasTraS deployment, and the MM ensures this. For fault-tolerance and to avoid a single point of failure, there are multiple TM master processes executing on different nodes at any instant of time– but only one of these processes can perform the duties of the TM master, the rest are backup processes waiting to take over in case the acting TM master fails. The TM master periodically communicates with the OTMs and the MM to determine the status of the OTMs, the state of the leases, and the current assignment of the partitions, and makes partition assignment decisions based on the load characteristics.

Metadata Manager (MM): The *Metadata Manager* stores the critical *system state* [1], viz., the mapping of partitions to OTMs, and leasing information for the OTMs and the TM master to guarantee exclusive access and to deal with failures and dynamics in the system. The MM also maintains timed leases that are granted to the OTMs and the TM master which have to be periodically renewed. The MM guarantees that at any instant, a lease is owned by only a single node. Since this information is critical to the correct functioning of the system, to ensure fault-tolerance, the state machine of the MM is replicated on multiple nodes in order to be able to tolerate node failures, and a protocol with stringent safety guarantees is used to ensure consistent state replication (see Paxos and a variant ZAB [27, 32]). This protocol ensures safety of data in the presence of arbitrary failures and network partitions, while liveness is ensured if a majority of replicas can communicate. This design of the MM is reminiscent of the design of Chubby [9] used in a number of systems at Google [11]. Since the MM only stores the meta information which is extremely small, is not frequently updated, and also does not reside in the data path, the MM is lightly loaded and is not a scalability bottleneck. But the availability and liveness of the MM is critical for the proper functioning of ElasTraS.

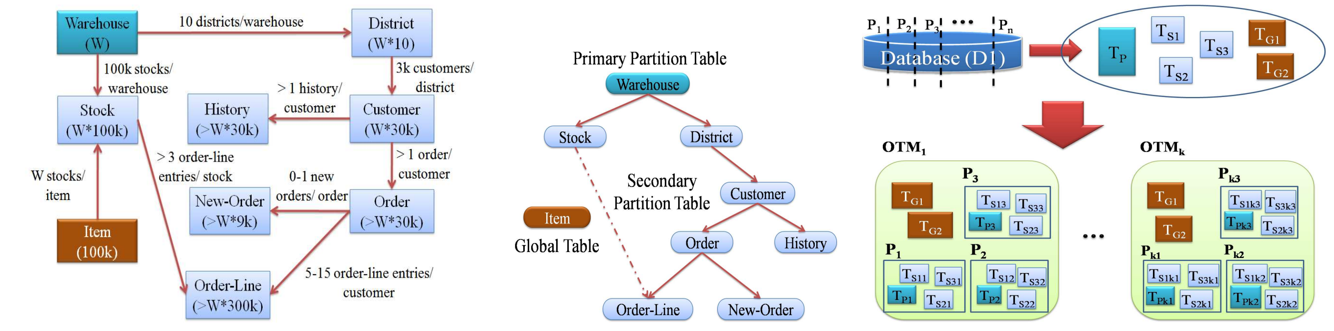
3. SCHEMA LEVEL PARTITIONING

In this section, we discuss the partitioning of a database to allow ElasTraS to scale out when dealing with large database instances. Recall that partitioning is needed only for large database instances, and in the case of a *multi-tenant* database, each database instance is small and fits into a single database partition. Partitioning is a common technique used for scaling databases, particularly for scaling updates, by distributing the partitions across a cluster of nodes, and routing the writes to their respective partitions. But managing large RDBMS installations with a large number of partitions poses huge administrative overhead primarily due to the following reasons: partitioning itself is a tedious task, where the database administrator has to decide on the number of partitions, bring the database offline, partition the data on multiple nodes, and then bring it back up online.[4] Additionally, the partitioning and mapping of partitions to nodes are often static, and when the load characteristics change or nodes fail, the database needs re-partitioning or migration to a new partitioned layout, resulting in administrative complexity and downtime. Furthermore, partitioning is often not transparent to the application, and applications often need modification whenever data is (re)partitioned. Moreover, most database systems only support partitioning at a single table level and do not automatically collocate partitions of two related tables for access locality. This often results in expensive cross-partition operations, and the application logic has to ensure the correctness of such cross-partition operations since the DBMSs often do not support such operations.

Schema level partitioning allows designing practical and meaningful applications while being able to restrict transactional access to a single database partition. The rationale behind schema level partitioning is that in a large number of database schemas and applications, transactions only access a small number of related rows which can be potentially spread across a number of tables. This access or schema pattern can be used to group together related data into the same partition, while allowing unrelated data in different partitions, and thus limiting accesses to a single database partition. A canonical example is a “Tree schema” where one table in the database acts as the root of the schema tree, and the primary key of the root table appears as a foreign key in the rest of the tables. All database accesses use the primary key of the root table, and hence, the tables in the database can be partitioned using the primary key of this root table. The TPC-C schema [34], which represents an e-commerce application, provides an example. The TPC-C schema comprises of nine tables in total, with the `Warehouse` table as the root table, and all tables (except the `Items` table) have the **`w_id`** (the primary key of the `Warehouse` table) as a foreign key, allowing the database to be partitioned using **`w_id`**. The Telecom Application Transaction Processing (TATP) benchmark [5] is another example of a tree schema. We remark that the use of tree schema for partitioning in an implementation choice, and the design of ElasTraS is not

[4]See `http://highscalability.com/unorthodox-approach-database-design-coming-shard` for a related discussion.

[5]`http://tatpbenchmark.sourceforge.net/TATP_Description.pdf`



(a) TPC-C Schema. (b) Latent tree structure. (c) Partitioning leveraging tree structure.

Figure 2: Schema Level Partitioning of the Database using TPC-C schema as an example.

tied to it. Furthermore, ElasTraS can be configured for both static as well as dynamic partitioning. Unlike the statically partitioned configuration whether the partitions are statically determined, for a dynamically partitioned configuration, ElasTraS is also responsible to split a partition when it grows big, and merge partitions (if they can be merged) when a partition has too little data.

We now provide an example where specific properties of a database schema is used for schema level partitioning in the TPC-C schema. Typically, many database schemas have a latent tree structure which can be leveraged for effective partitioning, and are referred to as "Tree schemas" [24]. For instance, consider the TPC-C schema [34] illustrated in Figure 2(a). The `Warehouse` table, which also determines the size of the database, drives a foreign key relationship (direct or indirect) with all other tables except the `Item` table. Additionally, all tables except the `Stock` and `Order-Line` tables have only a single foreign key relationship. Interestingly, the `Item` table is a static table whose size does not change during the execution of the benchmark. If we only consider the growing tables, and depict tables as vertices and the foreign key relationship as directed edges between the vertices, the resulting graph is a tree with `Warehouse` as the root (Figure 2(b)). We refer to the root table as the **Primary Partition Table**, and other nodes of the tree as the **Secondary Partition Table**. Based on this tree structure, the database is partitioned as follows: the *Primary Partition Table* is partitioned independent of the other tables using its primary key. Since the primary table's key is part of the keys of all the secondary table, the *Secondary Partition Tables* are partitioned based on the primary's partition key. In the TPC-C schema, the `w_id` attribute is the primary key of the `Warehouse` table and hence is used for partitioning the secondary tables. For example, if `w_id` 100 belongs to partition P_1, then all the rows of tables `District` and `Stock` with foreign key `w_id` 100 must also be mapped to partition P_1 to ensure access locality. This same partitioning scheme is used to partition the rest of secondary tables at all levels of the tree. This partitioning scheme allows collocation of all related rows from different tables into a single partition, and thus minimizes the need for cross-partition update transactions.

An important observation is that even though the `Customer` table has `District` as its parent, it is the keys of the Primary Table `Warehouse` that determines how the `Customer` table (and all other tables in the tree) is split. As a result, even though the `Order-Line` table has multiple parents, it still does not cause any complications since both parents share the `Warehouse` table as their ancestors, and one of the incoming edges to the `Order-Line` table can be ignored (as shown in Figure 2(b) where the edge between `Stock` and `Order-Line` is represented as a dashed line). Note that the only exception is the `Item` table. Since the `Item` table is read-only, and is primarily used as a lookup table, there is no need to split it. Rather such static lookup tables are replicated on all the partitions of the database. We refer to such read-only tables as **Global Tables**. Since the global tables are primarily used as a read-only lookup table, transactions need not validate the reads on global table(s). Figure 2(c) provides an illustration of the proposed partitioning scheme.

Note that since the contents of the *global tables* is the same for all the partitions of the same database, and since an OTM can potentially host multiple partitions of the same database, replicating a *global table* at every partition is wasteful. In ElasTraS, global tables are rather replicated at the level of OTMs (see Figure 2(c)), i.e., an OTM hosting at least one partition of a database will host only a single copy of the global table(s) in that database. An OTM loads the global tables when it starts serving the first partition of a database, and only serves global tables for databases for which the OTM is serving at least one global table.

4. IMPLEMENTATION DETAILS

We now explain the details of the ElasTraS prototype. Note that each OTM in ElasTraS is analogous to a single node relational database. We implement our own custom TM due to the following reasons: *First*, most open source database implementations are not designed to operate using the append-only semantics supported by the DFS. Since the DFS is a crucial part of the ElasTraS design to ensure fault-tolerance, and most open-source DFS implementations support append only semantics, the TMs have to adhere to the semantics of the DFS. *Second*, ElasTraS relies on schema level partitioning where a database partition consists of a number of related tables, and must support the dynamic reassignment of partitions to TMs. Most open source RDBMS support only partitioning at the level of single tables, and do not provide off the shelf support for dynamic partition assignment. Due to the similarity in the design of ElasTraS with Bigtable [11], a lot of design and implementation choices are borrowed from the design of Bigtable and its open source counterpart HBase [21].

4.1 DFS and Metadata Manager

Our implementation uses the Hadoop Distributed File System (HDFS) [22] as the distributed fault-tolerant storage layer. HDFS provides all the guarantees of a DFS outlined in the Section 2.3.[6] HDFS differs from POSIX compliant replicated file systems, and provides low cost block level replication. An HDFS deployment cinsists of a number of DataNodes (DN) which store the data, and

[6]Earlier versions of HDFS had issues with durability for appends, but the latest (unreleased) version 0.21.0 provides stable append functionality.

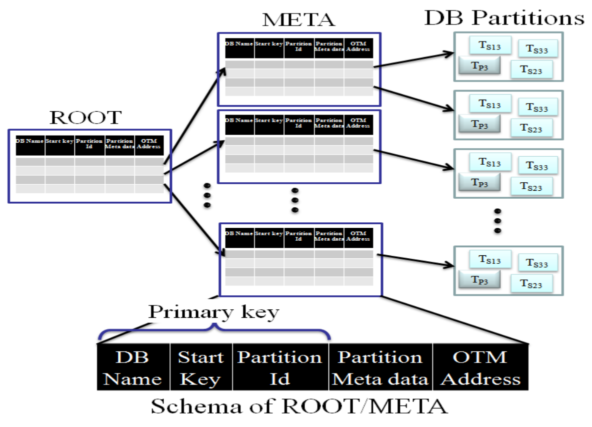


Figure 3: Structure of ElasTraS Catalog.

a write is acknowledged only when it has been replicated in memory at a configurable number of DNs. Since DFS writes are more expensive compared to normal disk writes, ElasTraS has been designed to optimize the number of writes to HDFS, and batch writes whenever possible.

For the Metadata Manager, we use Zookeeper [37]. Zookeeper uses a variant of the Paxos protocol [27], referred to as the Zookeeper Atomic Broadcast (ZAB) [32], for guaranteeing strong consistency amongst the replicas. Zookeeper exposes a file system interface to the clients (referred to as znodes), where a zookeeper client is atomically granted exclusive access to a znode for a certain period of time. This primitive is used for implementing the timed *leases* which are granted to the OTMs and the TM master. Zookeeper clients have to periodically renew the lease on a znode, and failure to do so results in the lease becoming available, and can be granted to another node requesting the lease. To reduce load on the system and the overhead of polling a znode to become available, Zookeeper allows registering *watches*, and the watchers are notified whenever there is a change in the status of a znode for which it has registered a watch. Each OTM and the TM master acquires exclusive write access to a znode corresponding to the server and writes its IP address to the file of the znode. An OTM or the TM master is operational only if it owns its znode. The IP addresses in the znodes are then used by the OTMs to locate the TM master and the TM master to know about the OTMs in the system. The OTMs register a watch on the master's znode, while the TM master registers watches for each OTM's znode. This event driven mechanism allows ElasTraS to scale to large numbers of OTMs.

4.2 Catalog

ElasTraS uses dynamic partition assignment for fault-tolerance, load balancing, and elasticity. It thus maintains catalog information for a mapping of the partitions to the OTMs which are serving the partition. This information is critical for the health and proper functioning of the system, and hence we categorize this information as the *system state* [1] and is maintained in the *metadata manager*. The actual implementation of the catalog tables is dependent on the partitioning scheme used. In this section, we explain the detailed implementation of the catalog table for the partitioning scheme for "Tree schemas" described in Section 3.

Recall that when partitioning the tree schema, the primary key of the *primary partition table* is used as the partition key. In our current implementation, we range partition the keys of the primary partition table, and assign different key ranges to different partitions. To maintain the mapping of ranges, a three level hierarchy structure similar to a B$^+$ tree is used with some minor modifications (this structure is also very similar to that used in Bigtable for tablet location [11]). Figure 3 provides an overview of the structure of the ElasTraS catalog. Each node in the structure is a database partition, the first two levels consist of partitions from special system databases, while the third level consists of the partitions for the databases which are stored in ElasTraS. The root of the structure is a database referred to as ROOT, and the next level is a database referred to as META. Each of these system level databases are comprised of only single tables. The ROOT is a special database which always consists of a single partition, and stores the locations of the META database partitions, while the META database partitions store the locations of the user level database partitions. A partition in ElasTraS is identified by the name of the database to which the partition belongs, the start key of the partition, and a unique identifier of a partition assigned to it at the time of its creation. The start key of a partition corresponds to the start of the range of the primary table's key which belongs to the partition. These three attributes together form the key for the ROOT and META database tables. Since the ROOT is never split, the Catalog structure is always a three level structure. Since the Catalog database partitions (i.e. the first two levels of the structure) are equivalent to user level database partitions, they can be served by any live OTM in the system. The *metadata manager* only maintains the address of the OTM which is serving the ROOT, and this information is used by the ElasTraS clients to determine the mapping of partitions to the OTMs.

4.3 Client Library

The ElasTraS client library provides a wrapper for the system, and is designed to hide to complexity of looking up and opening connections with the OTM that is serving the partition to which a transaction request is issued, and also hides the dynamics of the system due to nodes failing or partitions being moved for load balancing and elasticity. The application clients link to an ElasTraS client library to submit the transactions, and the ElasTraS client is responsible for looking up the catalog information and route the request to the appropriate OTM, retry the requests in case of any failure, and re-route the requests to the new OTM when a partition moves as a result of a failed OTM or for load balancing. In a more traditional web-services architecture, the client library is equivalent to a proxy/load balancer, except that it is linked with the application server and executes at the same node.

An ElasTraS client determines the location of the OTM serving a specific partition by querying the catalog tables (Figure 3). When a client makes the first connection to ElasTraS, it has to perform a three level lookup: (i) a query to the MM to determine the location of the ROOT, (ii) a query to the ROOT to determine the location of the appropriate META for the partition to be accessed, and (iii) a query to the META to determine the location of the OTM serving the partition. This first lookup is expensive, but a client caches this information for subsequent accesses to the same partition. The catalog tables are accessed again only if there is a cache miss, or an attempt to connect to a cached location fails. Performance is further improved by pre-fetching rows of the catalog tables to reduce the latency of subsequent lookups for different partitions.

4.4 Transaction Management

Concurrency Control: The OTMs are responsible for executing transactions on the partitions. As noted earlier, an OTM resembles a single node transactional RDBMS. Standard concurrency control techniques [5, 17, 26] can therefore be used for transaction management in an OTM. In our prototype, we chose to implement Optimistic Concurrency Control (OCC) [26] which guarantees serializable execution of transactions. In OCC, transactions do not obtain locks when reading or writing data, they rather rely on the optimistic assumption that there are no conflicts with other concurrently executing transactions. Before a transaction commits, it thus has to be validated to guarantee that the optimistic assumption was indeed correct and the transaction did not conflict with any other concurrent transaction. In case a conflict is detected, the transaction is aborted and rolled back. If validation succeeds, the transaction is committed and the updates of the transaction, which were kept local to the transaction till this point, are made global and visible to other transactions. We implemented parallel validation of transactions which requires a very short critical section for validation, and hence permits more concurrency. More details of OCC and parallel validation can be found in [26]. Note that if transactions do not require serializable isolation, then weaker forms of isolation (for instance Snapshot Isolation [5]) can also be implemented in the OTM to allow more concurrency.

Efficient Log Management: Logging is a critical aspect to guarantee durability of committed transactions in case of OTM failures. Every update made by a transaction is appended to the write ahead log (WAL). Before a transaction commits, the appends to the WAL do not require any durability guarantees and hence can be buffered. Once a transaction has successfully validated, and before it can commit, a `COMMIT` record for this transaction is appended to the log, and the log is *forced* to DFS to ensure durability of the transaction. To allow quick recovery from OTM failures, and recovery of the state of a failed OTM, ElasTraS maintains the transactional WAL in the DFS. Since the WAL is append-only, it adheres to the requirements of the DFS. The following optimizations are made to the log implementation to minimize expensive DFS accesses: (i) no entry is made to mark the start of a transaction, (ii) a `COMMIT` entry is forced only for update transactions, (iii) no log entry is made for an aborted transaction, the absence of a `COMMIT` record implies an abort, (iv) group commits are implemented to batch forcing of writes and to commit transactions as groups [35], and (v) to prevent one log thread which is appending to the DFS from blocking other appends, two log threads sharing a common log sequence number (LSN) and writing to two separate files are used [11]. (i) and (ii) ensures that there are no unnecessary log writes for read-only transactions. Buffering and group commits allow the batching of updates and are optimizations for improving throughput. To further reduce the number of DFS accesses, a single WAL is shared by all the partitions at an OTM. Each log entry therefore has a partition identifier to allow the association of log entries to partitions during recovery. Some DFS writes might observe stalls owing to various intermittent error conditions, and (v) allows one thread to continue processing the log requests, while another thread performs these writes. These optimizations ensure that transaction performance is not affected significantly even though the WAL is stored in the DFS.

Buffer Management: Similar to a buffer pool in databases, an OTM in ElasTraS maintains an in-memory image of the updates of the most recent transactions, which are periodically flushed to the DFS. This flush is equivalent to a checkpointing operation in an RDBMS [35]. But buffer pool management in ElasTraS has one marked difference compared to that in traditional databases. Owing to the append-only nature of the DFS, any database pages in the DFS are immutable, and new updates are appended to new pages. For this, we use a design similar to SSTables in Bigtable [11, 21] which were designed to deal with append only file system semantics, while allowing efficient reads. Furthermore, since updates to the rows are not in place, reads must merge the contents of the SSTables on disk to the main memory buffer of updates.

4.5 Storage and Buffer Management

Storage Layout. The append-only nature of the DFS does not allow ElasTraS to leverage from the page based model of storage commonly used in RDBMS [35]. We therefore chose a format for storing data which has been particularly designed for append-only storage. This design is similar to the design of SSTables which are used for storing data in Bigtable [11]. Unlike a database page which is modified in-place, an SSTable is an immutable structure where data is kept sorted by the keys of the tables, and supports fast lookups and scans. The structure of an SSTable is very similar to a row-oriented database page organization with one single difference: instead of rows stored in the database page, an SSTable is a collection of blocks. Similar to database pages, an SSTable also stores an index of blocks which is used to determine the locations of blocks within an SSTable. More details of the SSTable layout and implementation optimizations can be found in [11]. In Bigtable, a row-key is an uninterpreted string of bytes, while in ElasTraS a row's primary key can itself have a structure and be comprised of a number of attributes. To accommodate this added feature, only the comparator of the row key needs modification so that it is cognizant of the internal structure of the row-key and it does not view the keys as an uninterpreted string of bytes.

Buffer Pool Management. The append-only nature also requires a change in how the buffer pool is managed inside an OTM. It is common for a DBMS to cache the contents of database pages in a buffer pool while the pages are accessed. ElasTraS also performs similar caching of the contents of the SSTables. But since updates are not allowed in place, contrary to that of pages in an RDBMS, a difference arises in how updates are handled. In ElasTraS, updates are maintained as a separate main memory buffer which is periodically flushed to the DFS as new SSTables. Flushing of the update buffers to the DFS is performed asynchronously without blocking new updates. Old log entries are also cleaned up as the write buffers are flushed. The append-only semantics allows for more concurrency for reads and writes, but also adds two new design challenges: (i) updates to the rows in the database render old data obsolete, and SSTables stored in the DFS are thus fragmented by the presence of stale data, and (ii) in order to make reads see the latest updates, the results of reads should be merged with the SSTable cache and the new updates buffer. Addressing (i) is similar to garbage collection and is discussed in Section **??**. We address (ii) in a manner similar to [11] by merging the in-memory updates cache with the SSTable cache, and since in both cases, data is kept sorted by keys, the merge is efficient, and once merged, this merged result is incrementally maintained. An LRU policy is used for the buffer pool, and eviction of a buffered SSTable is done by removing it from the buffer and does not result in any DFS writes.

Garbage Collection. Every time the write buffer is flushed to the DFS, a new SSTable is created. Since read operations need to merge the contents of the SSTables with that of the write buffer, having too many SSTables will lead to high cost of reads. Furthermore, as noted earlier, since SSTables are immutable, this creates holes as data is updated. To deal with these issues, SSTables are periodically defragmented, and small SSTables are merged into larger SSTables (referred to as Compaction in [11]). Since a new SSTable is created as a result of this garbage collection and merge phase, reads can proceed while the merge is in progress, and are blocked momentarily only when the new SSTable replaces the old ones.

4.6 Fault tolerance in ElasTraS

Overall system recovery. Every OTM periodically sends heartbeat messages to the TM master to report its status, and the load on the different partitions which this OTM is serving. The master uses this information for load balancing and elasticity. Recall that the MM maintains leases (or znodes) for all the OTMs and the TM master, and the TM master registers watches on the znodes for the OTMs. In case of a failure of an OTM, the TM master times out on the heart beats from the OTM, and OTM's lease with the MM also expires. The master then atomically deletes the znode corresponding to the failed OTM. This deletion ensures that if the lease timeout was the result of a network partition, the OTM will not be able to serve the partitions once it re-establishes contact with the MM. An OTM, on the other hand, will stop serving its share of partitions if it looses the lease with the MM. This ensures safety of the system in the presence of network partitions. Once the znode for a failed or partitioned OTM is deleted, the master now has ownership of the partitions owned by the failed OTM, and reassigns them to other live OTMs. Once assigned access to a failed partition, an OTM performs recovery on the partition, before the partition is brought online. Note that the DFS allows recovery of the partitions and obviates the need to wait for the failed OTM to come back online.

ElasTraS also tolerates failures of the TM master. Recall that the TM master also has an associated znode, and the acting TM master owns the lease for this znode, while the standby TM master processes register watches on this znode. In case of a failure of the acting master, its lease on the znode expires, and the standby masters are notified of this change. The MM ensures that only one of these processes is granted the lease. On successfully acquiring the lease, the new master node writes its IP address to the master's znode, and becomes the new TM master for the system. The OTMs also register watches on the master's znode, and are notified of this change in leadership. The OTMs read the address of the new master and use it for sending future heartbeats. Note that a TM master fail over does not affect client transactions in progress since the master is not in the access path of the clients.

Recovery of an OTM. Write ahead logging and forcing of log entries before transaction commit ensures durability and transaction recoverability. On detecting a failed OTM, the TM master reassigns the partitions owned by the failed OTM to other live OTMs, and the new OTM performs recovery on the partition before the partition is back online. In the present implementation, uncommitted data is never pushed to the DFS, and hence, only REDO recovery [35] is needed. The OTM reads through the WAL and recreates the buffer pool to recover updates from all the committed transactions. Only those transactions that have a `COMMIT` record in the WAL are considered committed. Once the REDO pass over the WAL is complete, the contents of the buffer pool is flushed to the DFS, old files are deleted, and the OTM starts serving the partition. Version information in the SSTables is used to guarantee idempotence of updates, and guarantees safety and correctness with repeated failures [35].

4.7 Elasticity and Load Balancing

Elasticity is one of the major goals of the design of ElasTraS and not only allows the system to react to increases in load on the system, but also to reduce the operating cost of the system during usage troughs. The TM master is responsible for performing elastic load balancing of the partitions across the system. Every live OTM periodically sends heartbeats to the TM master which includes information about the load on the OTM, i.e., the number of partitions, the number of requests per partition, and other system usage parameters. The TM master aggregates this information across the entire system over a period of time to infer the load characteristics. In the present implementation, a threshold based algorithm is used to make a decision of increasing (or decreasing) the number of OTMs. If the system load is above a threshold, then new OTM instances are added to the system. When a new OTM starts, it obtains its lease and the address of the TM master from the MM, and reports to the TM master. On receipt of the first report, the TM master's load balancing algorithm selects some partitions from the list of online partitions, informs the corresponding OTMs to stop serving the partitions. Once the OTMs stop serving the partitions, the TM master assigns them to the new OTM(s). On the other hand, if the TM master determines that the load on the OTMs is less than the threshold, it consolidates the partitions to decrease the number of OTMs. The TM master determines which OTMs to terminate, and notifies the OTMs, which then close the partitions and terminate themselves. Once the partitions are closed, the TM master reassigns them to the set of live OTMs, which can then start serving these partitions. Since this transfer of ownership is coordinated by the TM master, it is fast, and involves flushing the buffer pool and closing the partitions at the old OTM, reopening the partitions at the new OTM, and updating the catalog tables at the MM to reflect this new assignment. No transfer of leases or recovery of partitions is needed as in the case of OTM failures. In the present implementation, the transactions in progress on the partitions being transferred are aborted and have to be restarted at the new OTM. The ElasTraS client library hides this complexity by retrying these aborted transactions, and routing the transactions to the new OTM. The application clients will notice an increase in the latency of some transactions, but the latency smoothes out as the system stabilizes. Since partition migration introduces additional cost in terms of the lost work of aborted transactions and warming up the cache at the newly assigned OTM, a high penalty is associated with partition reassignment, and the load balancer decides on adding (or removing) OTMs only if a change in load and usage patterns is observed over a period of time. This prevents the master from reacting immediately on short duration load spikes.

4.8 Parallel Analysis Queries

One of the design rationales of ElasTraS was to limit interactions of update transactions to a single machine. This design allows the system to scale out horizontally. We assume that nodes in ElasTraS are connected through a low latency and high throughput network. In such a scenario, a distributed transaction involving multiple OTMs and using 2 Phase Commit (2PC) [19] style of atomic commitment protocol might be realistic in terms of latency. But we do not support it in the present implementation primarily due to the following two reasons: (i) guaranteeing global serializability would require a common 2PC coordinator which can become a scalability bottleneck, and (ii) the blocking nature of 2PC on failure of the transaction coordinator can lead to performance degradation and reducing the availability of the system until the state of the master is recovered. We rather provide support for distributed ad-hoc read-only queries which can be executed at the *Read Committed* isolation level [5]. This lower level of isolation obviates the need for a 2PC type of protocol and global validation of the transactions, and it also gels well with the update transactions validated through OCC. When the client receives such a query, it dispatches the query in parallel to all the partitions relevant to the query. The partitions independently execute the queries, and the client merges the results from all the OTMs before returning them to the client. Since partitioning using the *tree schema* ensures that rows from all tables with related keys are collocated in a partition, the most frequent joins, where two tables are joined on the foreign keys, can be computed local to a partition. Note that we do not support across partition joins which includes joins on non-key attributes which are extremely infrequent.

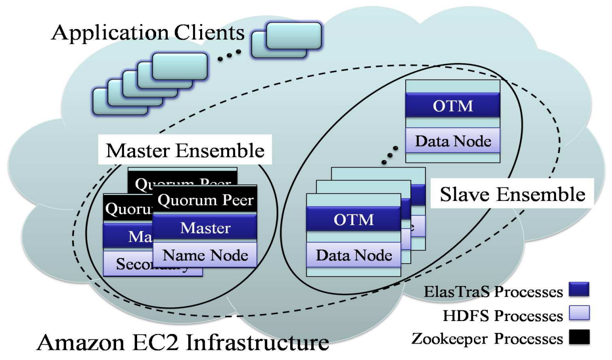


Figure 4: Cluster setup for the experiments.

5. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the performance of ElasTraS on a cluster of machines in the Amazon Web Services Elastic Compute Cloud (EC2) infrastructure. We evaluate the performance of ElasTraS using the TPC-C benchmark [34] which models the transactional workload of an online e-commerce store. We evaluate scalability in terms of latency of individual transactions and the transaction processing throughput of the entire system as the number of concurrent client transactions, the size of the database, as well as the number of OTMs increases. We also measure the elasticity of the system with changes in the load characteristics and its impact on the cost of operation. We evaluate ElasTraS for a single large TPC-C database instance. Note that the implementation and operation of ElasTraS is not dependent on the single-tenant or multi-tenant configuration; ElasTraS views both configurations as a set of partitions and client transactions update only a single partition. Therefore, a similar trend for the experimental results is expected to follow for the multi-tenant configuration as well. In the TPC-C benchmark, the `Warehouse` table is the scaling factor, and its size determines the amount of data in the database as well as the maximum possible number of clients. We configured ElasTraS with static partitioning with 10 warehouses per partition. The number of warehouses is varied from 1000 to 3000 as we scale up the number of OTMs from 10 to 30, and therefore, the number of partitions also scale up from 100 to 300. Each warehouse has about 280 MB of data; therefore with 3000 warehouses, the ElasTraS deployment manages about 850 GB of data (about 2.5TB data in DFS with 3X replication).

5.1 Experimental Setup

The experiments were performed on a cluster of machines in Amazon EC2. All machines in the cluster were "High CPU Extra Large Instances" (`c1.xlarge`) with 7 GB of memory, 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each), 1690 GB of local instance storage, and a 64-bit platform. Figure 4 provides an illustration of the experimental cluster setup. ElasTraS is implemented in Java and uses a number of open-source components from the Hadoop project. We used HDFS version 0.21.0-SNAPSHOT [22] (checked out from the 0.21 branch, which corresponds to the next slated release version of HDFS with durability guarantees on appends) as the distributed storage layer, and Zookeeper version 3.2.2 [37] for distributed lease and metadata management. Since the health of Zookeeper is critical for the proper functioning of ElasTraS, an ensemble of servers is used for running Zookeeper so that the Zookeeper state can be replicated on multiple nodes. In an ensemble of $(2n + 1)$ nodes, Zookeeper can tolerate up to n node failures. In our configuration, we used a 3 node Zookeeper ensemble; we refer to these machines as the *Master Ensemble*. The nodes in the master ensemble also host the ElasTraS Master(s), as well as the NameNode and Secondary Namenode of HDFS. A separate ensemble of nodes, referred to as the *Slave Ensemble*, is used to host the slave processes of ElasTraS and HDFS: namely the ElasTraS OTMs which serve the partitions, and the Data Nodes of HDFS which store the filesystem data. The number of nodes in the slave ensemble varied from 10 to 30 during various stages of the the experiments. At the peak capacity, the ElasTraS deployment has 30 OTMs, which amounts to about 210 GB of main memory, about 48.5 TB of disk space, and 240 virtual CPU cores. Together the Master and the Slave Ensembles constitute the data management infrastructure. Application client processes were executed on different nodes, but under the same infrastructure. The number of application client nodes used varied based on the load being simulated and the size of the database.

The latency of a transaction is measured as the time taken from submitting the transaction by the client to the time when the response was received, averaged over all the transactions. Transactions are executed as stored procedures, and the clients parameterize these transaction invocations. Throughput is measured as the overall end-to-end throughput of the system for executing the workload. We take an average of the total execution times of each of the clients, and use this as the time taken to complete the workload. The total number of transactions completed by all the clients and the total time taken to compute the workload together gives the overall system throughput. For our measurement, we first load the data into the system, perform an initial un-timed execution of the workload to warm-up the OTMs, and then perform the final timed run whose results are reported.

5.2 TPC-C Benchmark

In our experiments, we evaluated the performance of ElasTraS by executing the TPC-C benchmark [34] workload representing a standard OLTP workload. This workload is a mixture of read-only and update intensive transactions that simulate the activities found in complex OLTP application environments. The benchmark portrays a wholesale supplier with a number of sales districts and associated warehouses. There are nine tables in the TPC-C database (refer to Figure 2(b) for a description of the tables, and to the specifications [34] for details and schema for the tables). The benchmark suite consists of five transactions representing various business needs and workloads: (i) the NEWORDER transaction which models the placing of a new order through a single database transaction and represents a mid-weight, read-write transaction; (ii) the PAYMENT transaction which simulates the payment of an order by a customer which also reflects on the district and warehouse statistics, and represents a light-weight, read-write transaction; (iii) the ORDERSTATUS transaction representing a customer query for checking the status of the customer's last order and represents a mid-weight read-only database transaction; (iv) the DELIVERY transaction representing deferred batched processing of orders for delivery and consists of one or more transactions; and (v) the STOCKLEVEL database transaction which queries for the stock level of some recently sold items and represents a heavy read-only database transaction. A typical transaction mix consists of approximately 45% NEWORDER transactions, 43% PAYMENT transactions, and 4% each of the remaining three transaction types. All aspects of the TPC-C specifications were implemented, except for a few minor modifications. TPC-C requires that a very small fraction of orders be placed at one warehouse, and fulfilled by another warehouse. Since the database is partitioned by the warehouse id (the attribute `w_id` which is the primary key for the `Warehouse` table), implementation of this clause does not guarantee that update transactions update only a single partition. In our implementation, all orders are processed by the same warehouse where it was placed.

5.3 Scalability

In this experiment, we vary the number of OTMs in the cluster from 10 to 30. As the number of OTMs is increased, the size of the database is also increased proportionally. For a cluster with 10 OTMs, the database was populated with 1000 warehouses and the number of clients was increased from 100 to 600 concurrent clients in steps of 100, while for a cluster with 30 OTMs, the database size was set to 3000 warehouses, and the number of concurrent clients was varied from 300 to 1800 in steps of 300 (similarly scaling rules used for intermediate steps of scaling). The goal of this experiment was to verify the scalability of the system with increasing load (client transactions), number of OTMs, and data sizes. We report the latencies of the NEWORDER and PAYMENT transactions which constitute about 90% of the workload in the TPC-C transaction mix, and have stringent latency constraints. Transaction throughput is for the entire transaction mix, and includes all the transactions. The throughput is expressed in transactions-per-minute-C (tpmC) which is used as the performance metric for the TPC-C benchmark [34]. We used two variants of transactional workload: (i) the clients do not wait between submitting transaction requests, and (ii) the clients wait for a configurable amount of time (10–50ms) between two consecutive transactions. Variant (i) is used for stress testing the system, while variant (ii) represents a more realistic workload. For (ii), the reported latency and throughput computation does not include the wait times. As per TPC-C specifications for generating the workload, each client is associated with a warehouse, and issues requests only for that warehouse.

Figure 5 plots the latency of individual transactions (NEWORDER and PAYMENT) for different cluster sizes as the number of concurrent client transactions are increased. Along the x-axis we plot the number of concurrent clients, and along the y-axis, we plot the latency (in ms) for the transactions. The sub-figures correspond to the different cluster sizes. Figure 6 plots the system throughput of the transaction mix. Along the x-axis we plot the number of concurrent clients, and along the y-axis, we plot the transaction execution throughput (in tmpC).

As is evident from Figure 5, the system scales almost linearly in terms of transaction latencies as the cluster size and number of concurrent clients increase. For instance, the latencies of transactions for a 10 node cluster serving 100 clients is almost similar to the latency of the transactions with 200 clients on a 20 node cluster, and 300 clients on a 30 node cluster respectively. In terms of transaction throughput, as the cluster size increases from 10 to 30 nodes, the peak throughput is almost doubled. Furthermore, with an increase in the number of concurrent clients, an almost linear transaction throughput is observed till the system is saturated, at which point the throughput hits a plateau. Stressing the system beyond this point gives diminishing returns and the clients observe a steep increase in the transaction execution latency. In summary, these experiments demonstrate the scalability of ElasTraS in serving thousands of concurrent clients, and sustaining throughput of more than 0.2 million transactions per minute on a 30 node commodity cluster executing software components (like HDFS and Zookeeper) which are still at their infancy.

5.4 Elasticity and Cost of Operation

Figure 7 illustrates the effect of elastic load balancing as new OTMs are added to (or removed from) the system. Along the x-axis, we plot the time progress in seconds, the primary y-axis (left) plots the execution latency (in ms) of the NEWORDER transaction, and the secondary y-axis (right) plots the operating cost of the system (as per the cost of `c1.xlarge` EC2 instances). To reduce the noise in the latency measurements, we plot the moving average of latency averaged over a 10 second window. We keep the size of the database a constant at 2000 warehouses, and gradually increase the load on the system. At the start of the experiment, the cluster has 10 OTMs. As the load on the system increases, five new OTMs are added to the system (at around the 1300 sec mark). This causes re-balancing of the partitions amongst the new OTMs which results in a slight increase in transaction latencies during a period of flux (due to partitions which are taken offline for moving, and resulting in client timeouts and retries). Transaction latency stabilizes gradually as partition re-assignments are completed. As the load on the system decreases, OTMs are removed and partitions are consolidated into lesser number of OTMs (at about the 3200 sec mark, 8 OTMs are removed, resulting in a cluster size of 7 OTMs). Again, a period of flux is observed which gradually stabilizes. Since the number of OTMs is reduced, the operating cost of the system also reduces proportionately. This shows the benefits of an elastic system in adapting to increased load by adding more OTMs, thus reducing transaction latency, as well as reducing the operating cost leveraging from troughs in utilization. In the present implementation, we borrow the partition migration technique from HBase, and some instability in load balancing and migration resulted in a flux window spanning up to 60-80 seconds. A prudent partition migration and load balancing technique will reduce the flux window and the spike in latencies observed by the clients. We leave this for future work.

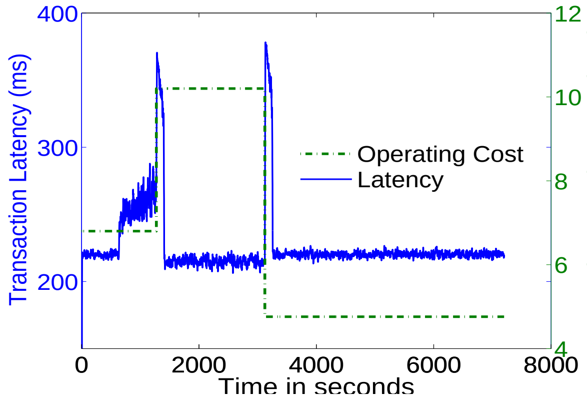
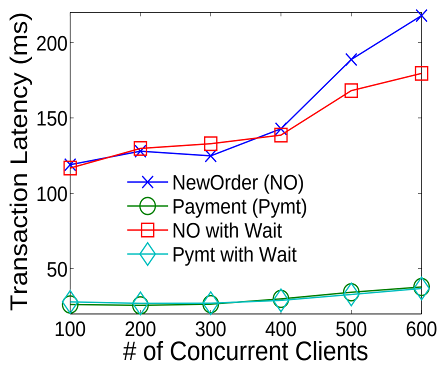


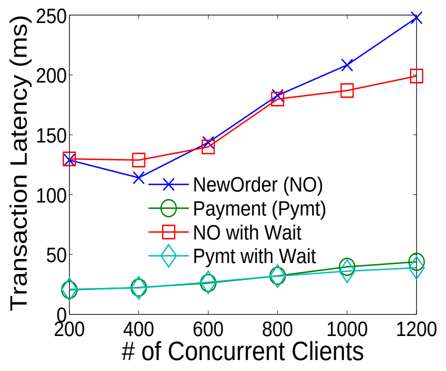
Figure 7: Effect of elastic load balancing on transaction execution latency and the cost of operation.

6. RELATED WORK

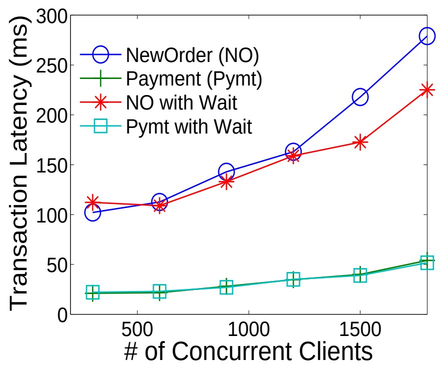
Scalable and distributed data management has been the vision of the database research community for more than three decades. As a result, a plethora of systems and techniques for scalable data have been documented in the literature. Owing to space limitation, we limit our survey to systems which are very closely related to our design. This thrust towards scaling out to multiple nodes resulted in two different types of systems: distributed database systems such as R* [28] and SSD-1 [33], which were designed for update intensive workloads; and parallel database systems such as Gamma [16] and Grace [18], which allowed updates but were predominantly used for analytical workloads. These systems were aimed at providing the functionality of a centralized database server, while scaling out to a number of nodes. Even though parallel database systems have



(a) 10 nodes, 1000 warehouses.

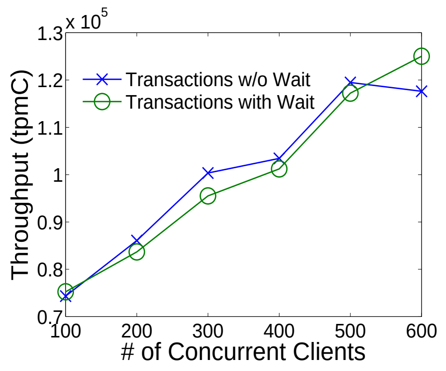


(b) 20 nodes, 2000 warehouses.

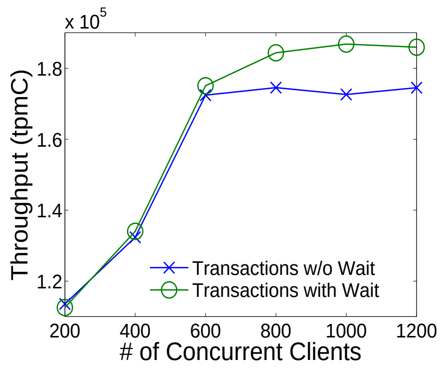


(c) 30 nodes, 3000 warehouses.

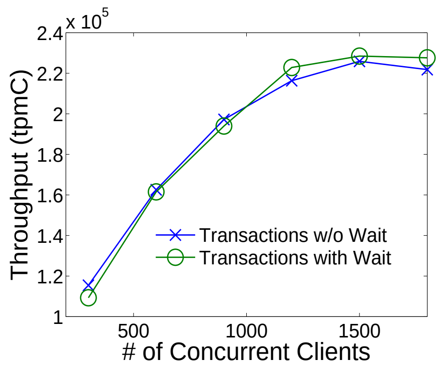
Figure 5: Latency of transactions for different cluster sizes and varying number of clients.



(a) 10 nodes, 1000 warehouses.



(b) 20 nodes, 2000 warehouses.



(c) 30 nodes, 3000 warehouses.

Figure 6: System throughput of transaction execution for different cluster sizes and varying number of clients.

been extremely successful, distributed databases have remained as research prototypes primarily due to the overhead of distributed transactions, and the fact that guaranteeing transactional properties in the presence of various kinds of failures limiting scalability and availability of these systems. The database community therefore resorted to *scaling up* database servers rather than *scaling out*. Changes in data access patterns resulting from a new generation of web-applications and the impossibility of scaling while providing high consistency, availability, and tolerance to network partitions [8] called for systems with high scalability and availability at the expense of weaker consistency guarantees. The result was the concept of *Key-Value* stores with accesses at the granularity of *single keys* or rows to ensure high scalability, and availability [1, 23]. These principles form the foundation of a number of very large distributed data systems [10–12, 15, 21].

A number of systems have recently been proposed for scalable data management in the cloud or similar cluster computing environments. Brantner et al. [7] propose a design of a database which utilizes a cloud based storage service like Amazon S3. Kraska et al. [25] build on this design to evaluate the impact of the choice of consistency of data when compared to the operational cost in terms of money and performance, and propose a framework where consistency guarantees can be specified on data rather than on transactions. Lomet et al. [29, 30] propose a radically different approach towards scaling databases in the cloud by "unbundling" the database into two components, the transaction component and the data component. Bernstein [6] proposes a scalable transactional system which relies on a flash based high performance distributed log. Transactions optimistically and atomically append to the log, and a replay of the transaction log on all transaction managers validates transactions for conflicts, and subsequently aborts transactions whose validation fails. Another interesting design was proposed by Aguilera et al. [2] where a two phase commit protocol (an optimization of the original 2PC protocol [19]) is used to support a class of operations referred to as *minitransactions* which allows limited but scalable distributed synchronization primitives such as test-and-set. Das et al. [14] propose a mechanism for providing transactional access over a dynamic group of keys using a *Key-Value* store like Bigtable [11] as a substrate. Yang et al. [36] propose a novel design of a data management system for supporting a large number of small applications in the context of various social networking sites which allow third party applications in their sites. Armbrust et al. [3] propose a scale independent data storage layer for social computing applications which is dynamically scalable, supports declarative consistency, while providing a scale aware query language. Another system H-Store [24], though not directly designed for the cloud, provides an example of a system where partitioning in a main memory database and replication of partitions at a number of nodes has been used to scale to large amounts of data in an enterprise cluster with huge memories and fast network interconnects. In summary, a large number of systems and designs have been proposed to address various issues of cloud data management, but none of them propose a system which is *elastic, scalable, fault-tolerant, self-managing* and provides *transactional access* at larger granularities beyond a single row – and this is the goal of the design of ElasTraS.

7. CONCLUSIONS AND FUTURE WORK

An elastic, scalable, fault-tolerant, self-managing, and transactional DBMS is critical to ensure the effective migration of data intensive applications to the cloud. We presented ElasTraS a transactional DBMS that is designed to meet the requirements of a cloud bound DBMS. The design of ElasTraS is inspired by the design principles of scalable *Key-Value* stores [11], and leverages from the decades of research in transaction management. We articulated the design principles of ElasTraS and presented the design and implementation of a prototype. Our design is suitable for both classes of DBMS applications in the cloud: large single-tenant database instances, and a large number of small multi-tenant databases. We further demonstrated how certain properties of the database schema can be used for effectively partitioning large database instances that allow update transactions to be restricted to a single partition, while being able to build meaningful and practical applications. We used the industry standard TPC-C benchmark to demonstrate this schema level partitioning, and evaluated performance of the system for the TPC-C workload. Our ElasTraS deployment on a 30 node cluster with over 1 TB of data served thousands of concurrent clients while sustaining a throughput of more than 0.2 million transactions per minute. In the future, we would like to make various extensions to the present design, the most notable ones include: designing an effective partition migration scheme to minimize the unavailability window of partitions during migration, and devising generalized schema level partitioning techniques that will limit update transactions to a single partition, while being able to cater to more diverse set of applications. Furthermore, we would like to investigate model based elasticity rather than rule based elasticity currently being used.

Acknowledgements

The authors would like to thank Aaron Elmore and Nagender Bandi for their insightful comments on the earlier versions of the paper which has helped in improving this paper. This work is partially supported by NSF Grants IIS-0744539 and IIS-0847925, and an AWS in Education grant.

8. REFERENCES

[1] D. Agrawal, A. El Abbadi, S. Antony, and S. Das. Data Management Challenges in Cloud Computing Infrastructures. In *DNIS*, 2010.

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

[3] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale Independent Storage for Social Computing Applications. In *CIDR Perspectives*, 2009.

[4] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD*, pages 1195–1206, 2008.

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

[6] P. Bernstein. Scaling Out without Partitioning. In *HPTS*, 2009.

[7] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, pages 251–264, 2008.

[8] E. A. Brewer. Towards robust distributed systems (Invited Talk). In *PODC*, page 7, 2000.

[9] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI*, pages 335–350, 2006.

[10] Cassandra: A highly scalable, eventually consistent, distributed, structured key-value store, 2009. `http://incubator.apache.org/cassandra/`.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[13] S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *USENIX HotCloud*, 2009.

[14] S. Das, D. Agrawal, and A. El Abbadi. G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud. In *ACM SOCC*, 2010.

[15] 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. In *SOSP*, pages 205–220, 2007.

[16] D. J. Dewitt et al. The Gamma Database Machine Project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990.

[17] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[18] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *VLDB*, pages 209–219, 1986.

[19] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[20] J. Hamilton. I love eventual consistency but... `http://perspectives.mvdirona.com/2010/02/24/ILoveEventualConsistencyBut.aspx`.

[21] HBase: Bigtable-like structured storage for Hadoop HDFS, 2009. `http://hadoop.apache.org/hbase/`.

[22] HDFS: A distributed file system that provides high throughput access to application data, 2009. `http://hadoop.apache.org/hdfs/`.

[23] P. Helland. Life beyond Distributed Transactions: An Apostate's Opinion. In *CIDR*, pages 132–141, 2007.

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

[25] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

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

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

[28] B. G. Lindsay, L. M. Haas, C. Mohan, P. F. Wilms, and R. A. Yost. Computation and communication in R*: A distributed

database manager. *ACM Trans. Comput. Syst.*, 2(1):24–38, 1984.

[29] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR Perspectives*, 2009.

[30] D. B. Lomet and M. F. Mokbel. Locking Key Ranges with Unbundled Transaction Services. *PVLDB*, 2(1):265–276, 2009.

[31] D. Obasanjo. When databases lie: Consistency vs. availability in distributed systems. `http://www.25hoursaday.com/weblog/2007/10/10/WhenDatabasesLieConsistencyVsAvailabilityInDistributedSystems.aspx`, 2009.

[32] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *LADIS*, pages 1–6, 2008.

[33] J. B. Rothnie Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. L. Reeve, D. W. Shipman, and E. Wong. Introduction to a System for Distributed Databases (SDD-1). *ACM Trans. Database Syst.*, 5(1):1–17, 1980.

[34] The Transaction Processing Performance Council. TPC-C benchmark (Version 5.10.1), 2009.

[35] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., 2001.

[36] F. Yang, J. Shanmugasundaram, and R. Yerneni. A scalable data platform for a large number of small applications. In *CIDR*, 2009.

[37] ZooKeeper: A high-performance coordination service for distributed applications, 2009. `http://hadoop.apache.org/zookeeper/`.

APPENDIX

A. CORRECTNESS GUARANTEES

In this section, we provide arguments for correctness of the individual components of the system as well as the entire system. Our implementation relies on two open source components (the metadata manager implemented using Apache Zookeeper [37], and the DFS implemented using HDFS [22]), and the correctness of the system is dependent on certain guarantees by these systems. We first list these guarantees and then use them to reason about correctness. ElasTraS is not designed to tolerate network partitions, but it should guarantee data safety in the presence of a network partition. We further assume that there is no malicious behavior. All communication uses reliable communication protocols like TCP, and hence there are no message related failures such as lost, duplicated, or reordered messages. We refer to *safety* as the *correctness* of the system, i.e., data is not lost, corrupted, or left inconsistent. *Liveness* on the other hand is the *ability to make progress*.

A.1 Guarantees by the MM and DFS

The metadata manager (MM), implemented using Zookeeper [37], has a replicated state to tolerate failures and increased availability, and uses a variant of the Paxos protocol [27], referred to as ZAB [32], which provides the following guarantees:

GUARANTEE 1. ***Replica Consistency.*** *Replicas of the MM's state are strongly consistent.*

GUARANTEE 2. ***Safety.*** *The state of the MM is safe even in the presence of arbitrary failures, including network partitions.*

GUARANTEE 3. ***Liveness.*** *The MM is live and can make progress if a majority of replicas are non-faulty and can communicate.*

GUARANTEE 4. ***Mutual exclusion.*** *At any point of time, only a single process can own a lease (or a znode in Zookeeper).*

Guarantees 1, 2, and 3 are provided by Paxos [27] (and also ZAB [32]). Guarantee 4 follows directly from the Paxos protocol [27]. At a very high level: if a lease (or znode) is available, the request to acquire a lease results in a new Paxos propose phase being instantiated by the present leader of the replicas, and the proposal is accepted and the lease granted only when a majority of replicas have acknowledged the proposal. This majority approval of a proposal ensures that any other concurrent proposal for granting the same lease is rejected and thus only a unique process is granted the lease. The TM master and OTMs can operate only if they hold the corresponding lease, and the processes terminate themselves if the lease could not be renewed and is lost.

The DFS layer must provide the following guarantee:

GUARANTEE 5. ***Durability of appends.*** *For scenarios of single site failure, appends to the DFS are never lost once a "flush" or a "sync" has been acknowledged.*

This is guaranteed by the DFS layer by acknowledging a flush only after the appends have been replicated in memory to at least two or more replicas.

A.2 Correctness of the Master

CLAIM 1. *At any instant of time there is only a single acting TM master in one ElasTraS installation.*

Claim 1 follows directly from Guarantee 4 which ensures that the ownership of the znode corresponding to the master is granted to at most one process, which now acts as the TM master.

CLAIM 2. *At any instant, at most one OTM has read/write access to a database partition.*

Only an acting TM master owning the master lease can perform partition assignment. On startup, the TM master scans the catalog for unassigned partitions and assigns them to the OTMs. The TM master assigns a partition only once, and since there is only a single TM master, a partition is assigned to only a single OTM during normal operation. To re-assign partition for load balancing and elasticity, the ownership of the partition is handed over from the OTM to the TM master and finally to a new OTM. The failure of the master during this transfer would at most render the partition unassigned, and the new master will recover the state of the failed master and complete this assignment. On the failure of an OTM, the TM master performs reassignment if and only if it was able to successfully delete the znode corresponding to the OTM. This ensures that the old OTM can no longer serve any of the partitions assigned to it. This again ensures that the newly assigned OTM is the only OTM which owns those partitions.

A.3 Correctness of the OTM

CLAIM 3. *Transactions on a single partition are guaranteed to be atomic, consistent, durable, and serializable.*

A transaction commit is acknowledged by an OTM only after the log records for a transaction, including the `COMMIT` record, has been forced to the DFS. Guarantee 5 ensures that the log entries can be read even if the OTM fails after forcing the log records.

This ensures that the transaction state is recoverable and guarantees durability. On failure of an OTM, the REDO recovery operation on a partition from the commit log applies all updates of a transaction whose `COMMIT` record is found in the commit log (i.e. the transaction successfully committed) and discards all updates of a transaction whose `COMMIT` record is missing (meaning either the transaction aborted and was rolled back, or it did not commit before the failure and hence is considered as aborted). This ensures atomicity of the transactions. Consistency and Serializability is guaranteed by the concurrency control protocol [26].

A.4 Overall Correctness of the System

CLAIM 4. ***Safety.*** *Data safety is guaranteed in the presence of arbitrary failures, including network partitions.*

Claim 4 follows from the requirement that an OTM or TM master can operate only if they own the leases, and Claims 1 and 2.

CLAIM 5. ***Liveness.*** *Liveness is guaranteed even in the presence of site failures, provided the metadata manager is alive and not partitioned.*

Liveness of the system is never threatened by the failure of an OTM, the master recovers the state of the OTM. If the MM is alive and is able to communicate with the rest of the nodes, then ElasTraS can also tolerate the failure of the TM master and operate unhindered without a functioning master unless an OTM fails. This is because the TM master is not on the data path. This allows a standby master to acquire the lease on the master's znode and take over as the new TM master, without interrupting any client transactions in progress.