Chimera: Data Sharing Flexibility, Shared Nothing Simplicity

Umar Farooq Minhas [*]
University of Waterloo
Waterloo, Canada
ufminhas@cs.uwaterloo.ca

David Lomet
Microsoft Research
Redmond, WA
lomet@microsoft.com

Chandramohan A. Thekkath
Microsoft Research
Mountain View, CA
thekkath@microsoft.com

Abstract

The current database market is fairly evenly split between shared nothing and data sharing systems. While shared nothing systems are easier to build and scale, data sharing systems have advantages in load balancing. In this paper we explore adding data sharing functionality as an extension to a shared nothing database system. Our approach isolates the data sharing functionality from the rest of the system and relies on well-studied, robust techniques to provide the data sharing extension. This reduces the difficulty in providing data sharing functionality, yet provides much of the flexibility of a data sharing system. We present the design and implementation of Chimera – a hybrid database system, targeted at load balancing for many workloads, and scale-out for read-mostly workloads. The results of our experiments demonstrate that we can achieve almost linear scalability and effective load balancing with less than 2% overhead during normal operation.

1 Introduction

1.1 Different Database Architectures

A number of different architectures for parallel database systems have been explored in the evolution of current commercial database systems. The "big picture" categorization is the division into *shared nothing* and *data sharing*. We begin by clearly defining what we mean by these terms. A more complete description of the technical issues of database architectures is given in Section 2.

Shared Nothing DBMS: A database server or a node, whether a single node system, or a member of a multi-node cluster, has exclusive access to the data that it manages. Data for a node is usually stored on a disk that is locally attached to the node. A database is usually divided into mutually exclusive partitions and distributed among the nodes of a cluster, where each node is responsible for only its own partition. Only the node responsible for a partition can cache data from its partition. This node can hence provide concurrency control and recovery for the data that it manages without the need to coordinate with any other node. This data can never be in the cache of a different node. This simplicity is a great advantage and in many settings leads to higher performance and scalability, because there is no need for such coordination. However, defining a good data partitioning is hard, partitions are usually statically defined, and repartitioning (which is necessary for load balancing) requires a reorganization of the entire database, and may require moving data between nodes.

[*]Work done while at Microsoft Research.

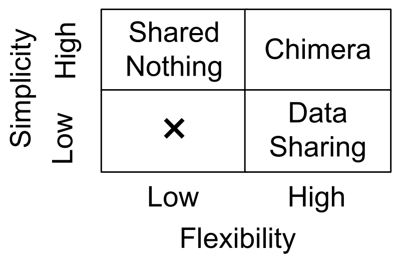


Figure 1: Chimera: Best of both worlds.

Data Sharing DBMS: More than one node of a cluster can cache the same data (hence the name "data sharing") from the database stored on disk. All nodes in the cluster require access to the storage devices, which may be provided by using storage area networks (SAN) that enable shared access to disks. A data sharing DBMS cluster can respond to changing system load by deploying additional nodes for "hot" data to scale beyond single node performance. As load changes, nodes can be redeployed to accommodate the work, balancing load without repartitioning the data. A data sharing system is more complex to implement, needing distributed concurrency control and recovery (CC&R) and distributed cache coherence, which impose additional load on the system.

The current database market is fairly evenly split between data sharing systems and shared nothing systems. In the data sharing camp are such long-lived commercial products as Oracle [17], and IBM mainframe DB2 [2]. In the shared nothing camp are the more recently implemented database systems such as IBM DB2/UDB [19] and Microsoft SQL Server [21].

In the TPC benchmarking wars, shared nothing systems usually win the price/performance battles, while data sharing systems have, with some frequency, won on peak performance. An interesting aspect is that when set up to run the benchmark, data sharing systems are frequently careful about partitioning **access** to the data to reduce the burden of CC&R and cache management.

Shared nothing systems are easier to implement because they are less complex than data sharing systems [25]. For example, shared nothing systems do not face the problems of distributed locking and complex failure scenarios for recovery. However, data sharing provides benefits such as *increased responsiveness to load imbalances*, which is highly desirable for dynamically changing workloads.

We would like a system with the flexibility of data sharing for load balancing and scale-out, but with the simplicity of shared nothing, as shown in Figure 1. An ideal solution would combine the advantages of data sharing with shared nothing without any of its disadvantages e.g., without the added complexity that arises from the use of a distributed lock manager, global buffer manager, complex logging, and recovery mechanisms. In this paper, we present the design and implementation of Chimera whose purpose is to achieve this objective by adding a data sharing "extension" to a shared nothing DBMS. Chimera is built using off-the-shelf components, providing effective scalability and load balancing with less than 2% overhead during normal operation.

1.2 Our Approach

We enable sharing of databases among all the nodes of a cluster i.e., each node, acting as a **remote** node, can access a database hosted at any other node, the **local** node in the cluster, if the local node chooses to share that database. We start with a shared nothing cluster of low-cost desktop machines that can each host a stand-alone shared nothing DBMS, with one or more databases stored on their respective local disks. We carefully extend it with data sharing capability while minimizing the amount of new code, and simultaneously limiting the frequency of its execution. The techniques we use are applicable, in principle, to any traditional shared nothing database system. Our contribution is in (1) the design of a database architecture that is a hybrid of shared nothing and data sharing, (2) the implementation of Chimera – a system based on the hybrid architecture, and (3) the experimental confirmation of the value of the approach.

"Full blown" data sharing can lead to both complexity and cost in lost performance. Thus, our goal is to provide carefully chosen data sharing that provides load balancing and scale-out benefits to a selected set of applications that are nonetheless of great practical importance since they occur frequently in modern data centers. We have built Chimera that provides:

1. **load balancing at table granularity.** It can offload the execution cost of database functionality in units of tables. A remote node can become the database server for accesses to one, several, or all tables, the node hosting the data simply serving pages belonging to the shared table(s) from the disk. Our experiments confirm this very effective load balancing, which is impossible with a shared nothing system.

2. **scale-out for read-mostly workloads.** Read-mostly workloads represent a wide range of internet based workloads (e.g., a service hosted at Microsoft, Yahoo, or Google) and thus are important. In such an environment, data is sometimes hosted on a centralized database because an application finds it difficult to partition the data, which limits its scalability. Our experiments confirm effective scale-out for these kinds of applications.

3. **close to shared nothing simplicity.** The key innovations are that only a single node can update a database at a time, while the node doing the updating may change to enable load balancing. This means that the updating node has exclusive access to both the data and the transaction log. Thus, while working on tables, including those delegated to it by other nodes, it executes as a shared nothing system. Only in transitioning ownership need it be aware of data sharing, taking active steps to invalidate cached data at other nodes.

Our architectural goal is to minimize the number of system components impacted by our extensions. Most of the shared nothing base system is unaffected by the addition of data sharing. Chimera relies on three main components: **(1)** the data is stored in a shared file system such as Common Internet File System (CIFS) [10] or Network File System (NFS) [23], so all nodes can access it, **(2)** a distributed lock manager such as Boxwood [18] or Chubby [3] provides ownership control, and **(3)** code added in the shared nothing DBMS coordinates data access and sharing among the nodes.

We present experiments demonstrating that Chimera enables both load balancing generally and scale-out for a read-mostly workload. Given the benefits of our approach and the modest size and architecture of our extension to "shared nothing" systems, this may gradually erase the hard boundary between data sharing and shared nothing.

1.3 Organization of the Paper

The paper is organized as follows. Section 2 describes data sharing and shared nothing database architectures. It details the complexities of the data sharing systems found in the database and systems literature. In Section 3, we provide an overview of our approach and talk about system components impacted when providing data sharing. Our implementation is described in some detail in Section 4, showing how our goals shaped our implementation approach. We report experimental results in Section 5 that demonstrate the value of the approach. Related work is discussed in Section 6. The final section summarizes the paper.

2 DBMS Architectures

2.1 The Two Architectures

A shared nothing database system (SN) deals with an inherently simpler situation than does a data sharing system (DS). The SN provides concurrency control and recovery techniques where it has exclusive ownership of and complete information about the data it manages. A DS must coordinate access to a dynamic set of resources that can change inflight, and provide recovery for the data that it manages when multiple nodes are updating the same data and committing transactions. Parts of database system are unaffected by whether a DBMS is DS or SN. For example, whether a system is DS or SN has little impact on query optimization or query execution – the architectural impact is mostly to the database kernel.

Some of the issues in databases systems apply also to distributed or cluster file systems, e.g., [1, 6, 8, 26] to cite a few early examples. Each metadata operation initiated by the file system is typically treated as an atomic operation requiring locking and logging techniques analogous to database kernels. However, distributed file systems do not typically allow arbitrary client transactions.

The complexity of the task that DS systems face makes it inevitable that they are more complex with longer execution paths. DS must provide distributed concurrency control, not only for access to relational data, but also for distributed cache coherence as well. More complex and costly handling of system crashes can also result. To provide recovery, a DS cannot simply write to a private log. Rather, it either needs a shared log or must ensure that its log be correctly merged with other logs. In the next two subsections, we delve more deeply into the challenges that DS systems face in accessing "shared data" from multiple nodes in a cluster of such nodes.

2.2 Concurrency Control

When data is shared by nodes of a cluster and lock based concurrency control is used, as commonly done, the locks on shared data must be visible to all accessing nodes. These are called distributed (or global) locks and the lock manager involved is usually called a distributed lock manager (DLM). Any node accessing data must request an appropriate distributed lock on the data at the DLM, which is costly. The DEC Rdb [16] experience was that distributed locks required almost 50% of the code path required for executing an I/O, and also introduced significant latency as well.

Achieving good performance required a relentless effort to reduce the number of distributed locks. These locks arise more frequently than one might expect. Because a DS has a distributed cache, costly distributed locks, not cheap latches, are required to protect pages and keep the cache coherent. In Rdb, locks are held by software processes, with each transaction assigned to a single process. Rdb 7.0 introduced lock retention across transactions to avoid extra trips to the DLM to re-acquire locks for the next transaction. The expectation was that frequently (and especially with effective application partitioning) the lock would be needed again soon by another transaction executing in the same process. A similar technique, called leases [9], is used in distributed file/storage systems as well [3, 6, 26, 24].

When a node of a DS cluster crashes and locks are not persistent, its distributed locks are lost. A very complex recovery process is needed to sort things out (see below). Making the locks persistent avoids this, but introduces extra cost. Writing the locks to disk results in appalling performance. IBM mainframe DB2, when executing on a cluster that includes a "Sysplex", specialized hardware that makes distributed locks stable, improves performance though at a not insignificant cost. Another way to make locks persistent is to build the DLM as a replicated state machine [12], an approach found in some distributed systems [3, 18, 26]. This requires no special hardware support and the communication and disk bandwidth performance costs may be acceptable in some cases.

2.3 Recovery

Data sharing systems can use a variety of techniques to ensure that locks persist, as indicated above, but they all add extra code path during normal execution. When distributed locks are volatile, execution cost is reduced but a node crash loses locks. This "partial" failure undermines a claimed advantage for data sharing clusters, i.e., that single node failures do not necessarily make data inaccessible, though this requires that multiple nodes can access each disk. (Such "shared disk" hardware is sometimes confused with data sharing databases because DS systems have frequently been built using shared disks.)

To cope with lost locks, Rdb instituted a "database freeze" [22]. This makes data briefly unavailable. All transactions are aborted, both on crashed nodes and on still up nodes. Transaction undo is guaranteed because sufficient locks are taken during forward execution so that no additional locks (which may block) are needed during undo. Undo returns the system to a state where no transactions are active and it is known that **NO** locks are held. At that point, access to data can resume (and very quickly at the nodes that did not crash) while the crashed node itself is recovered. This is both complicated, and briefly turns a partial failure into at least a brief complete interruption of service. Some transaction aborts might be avoided were Rdb to track, outside of the DLM, the locks held by transactions. But it does not.

Another complication is how logging is handled. Two different ways to approach this problem have been pursued.

1. Provide a shared log with all nodes posting log records to the shared log. Coordinating high performance shared logging is subtle. One cannot afford to access this log at every update as this involves interprocess communication. Rdb's shared log manager hands out log blocks to processes, which pass them back when they are filled. At a crash, extra effort is needed to determine the exact end of the log since the log tail typically had a number of noncontiguous blocks.

2. Provide separate logs for each node. This changes the problem to how to merge the logs during recovery. With separate logs, the usual LSN ordering no longer is guaranteed to match the time-sequence of updates. So the notion of LSNs and how they are assigned to log records and pages needs to be modified [14, 20, 26] so that, at recovery time, the separate logs can be effectively merged. Finally, to make it possible for a node to recover without involving other nodes in its recovery, one needs to make sure that recovery for any single data item is confined to a single log. When the node caching a data item changes, an up-to-date copy of the data must be on stable storage (disk) before the data is cached at the new node. This is coordinated via the DLM and often requires a disk write to flush dirty data from the old node, adding a performance burden.

3 Simplified Data Sharing

As we note in the previous section, providing full-fledged data sharing is hard owing to the complexity of CC&R and cache management. However, we note that both shared nothing and data sharing systems can be implemented using the same shared nothing hardware architecture. Only the system software needs to be different. A key challenge of a data sharing system is to maintain data integrity without sacrificing performance. Careful design is needed to keep the overhead low. Although, software-only techniques for implementing shared disk systems have been proposed (e.g.,

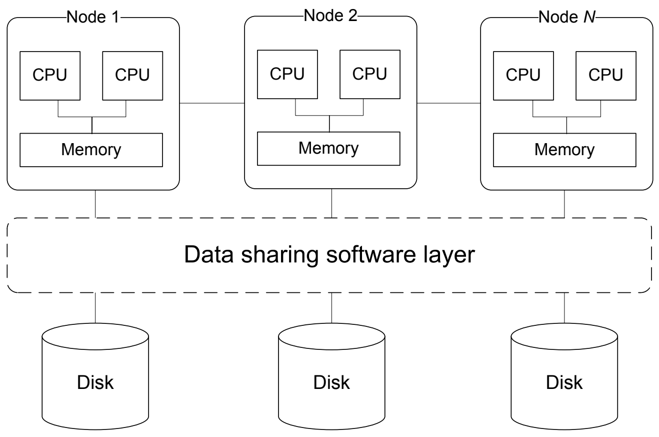


Figure 2: Shared nothing architecture with data sharing software layer.

[13]), specialized hardware has frequently been used to implement data sharing systems efficiently. This leads to specialized designs and increased costs since this hardware is not commodity and is priced accordingly.

In this work, we take a stand-alone SN database architecture running on commodity hardware and change only the software to implement a system that exhibits many of the characteristics of a data sharing system. We illustrate the resulting architecture for such an approach in Figure 2. The added *data sharing software layer*, which is carefully partitioned from the remainder of the system, distinguishes this architecture from a shared nothing architecture. We present the design of this layer below while the implementation details are presented in Section 4.

We exploit three fundamental simplifications in designing our solution.

1. All metadata and associated structures of the database are accessible by each node of a cluster as if it owned the database.

2. Only one node at a time can update a database, while multiple nodes can share read access.

3. Nodes have ownership locks on tables that are separate from transactional locking in order to share data access.

These simplifications definitely impact the data sharing functionality that our system can provide. Table granularity limits the number of locks and frequency of lock requests. While one node may be updating a table, other nodes can be sharing read access to the other tables of a database. However, very useful data sharing capability is supported, and at a very modest implementation cost. Importantly, load balancing by off-loading data from a "local node" to a "remote node" can be accomplished without moving the data from the local node's disks to the remote node's disk, and can easily be done on a time scale of seconds, not hours. The following subsections provide an overview of different components required to build a data sharing system. We show how one can use off-the-shelf components and make minor changes to the DBMS to build a hybrid system such as Chimera.

3.1 File-system Access

For data sharing, the database files need to be accessible from multiple nodes. Data sharing systems traditionally solve this problem by using SANs, or shared disks. We posit that software-only techniques, for example, distributed or network file systems, can be used for storing data and can provide adequate performance for a data sharing system. We can use any network or distributed file system (see [10, 23, 26] for some representative examples) to share access to the database files from multiple nodes. Allowing a DBMS to use a DFS or NFS to store database files requires simple changes to the DBMS. In Section 4 we present details of these changes for one particular DBMS. Similar changes can be made to any existing shared nothing DBMS.

By sharing the database files (using a DFS or NFS), we encapsulate all data and metadata into a single package (design consideration (1)). When we provide access to the data, we simultaneously provide access to the metadata. This permits the remote (sharing) node to receive queries, compile, and optimize them as if the database were local. The local node for the database acts as a disk controller for the remote nodes, reading its disk to provide database pages requested by the remote nodes. The result of this activity is invisible to the local node database system. Each remote node caches data as required by the queries it is executing.

3.2 Distributed Locking

We need some form of global locking with a data sharing system in order to synchronize access to common data. It is very important to keep the overhead of locking low. In the past, this has led to a highly complex distributed lock manager usually paired with specialized hardware for maximum performance. However, it is important to note that we do not need a distributed lock manager in its full glory for our data sharing solution. We simply need a lock manager that is able to handle lock requests on remote resources (e.g., in our case, a remote table) in addition to the local ones. Such a "global" lock manager dealing with ownership locking is far less complex than a full fledged distributed lock manager more typical of data sharing systems.

Given the wide spread deployment of internet scale services, several highly-scalable distributed coordination services have been proposed [3, 11, 18]. Chubby [3] offers a coarse-grained locking service that is able to provide strong synchronization guarantees for distributed applications. Chubby is used by Google File System [7] and Bigtable [4] to coordinate between master servers and their clients as well as to store metadata. ZooKeeper [11] offers a simple and efficient wait-free coordination service for large scale distributed applications. ZooKeeper is not a lock manager but the basic coordination primitives that it provides can be used to implement a distributed lock manager. Boxwood [18] implements a fault-tolerant distributed locking service that consists of a single master server and multiple slave servers. It implements leases which are cached at each lock client. A lease on a lock is released only if a conflicting request for the same lock is issued by another lock client. Failures of lock servers and/or clients are detected and safety is guaranteed by the use of a recovery procedure.

Our system exploits a global lock manager adopted from Boxwood [18]. While distributed fine grained locking, both for resources and for lock owners, has been complex and costly, simplified global locking at a coarse granularity has been successfully exploited [3, 11, 18], as mentioned above. Locking in our system goes on at two levels, globally for ownership sharing of tables among nodes and locally for transactional isolation within each node. This global/private distinction and its advantages were highlighted in [15]. We call the duration of ownership an ownership period. Importantly, once an ownership period is activated, no changes to the shared nothing system are required to mediate access during transaction execution within the period.

Locking granularity for ownership sharing among nodes is at the table level. This is a practical limitation designed to reduce the number of locks and hence locking overhead. We assume that reader and writer nodes (not transactions) hold ownership table locks for durations involving multiple transactions. Recall that we are using data sharing as a way to do load balancing in a low cost manner. Our data sharing is not intended as a continual mode of tightly coupled execution, but to inexpensively respond to load changes.

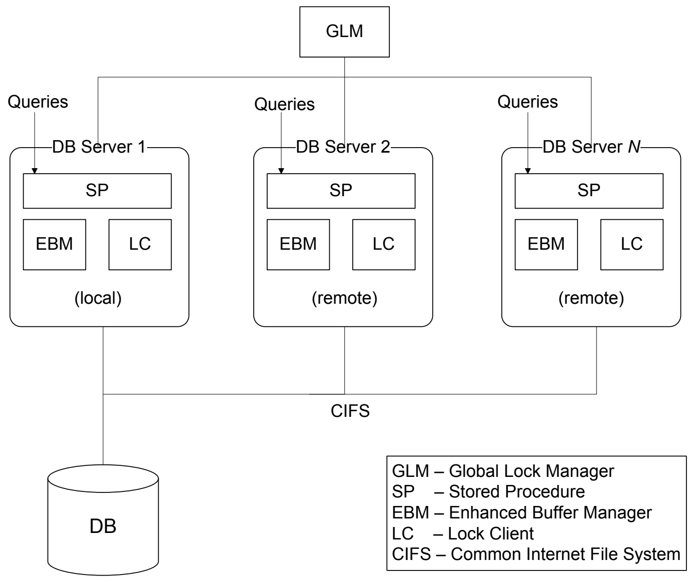


Figure 3: Chimera: Overall system architecture.

3.3 Distributed Buffer Management

Since data from the stable database can be cached at multiple nodes simultaneously, we need a distributed buffer manager to maintain consistency in face of updates to the stable database. A surprisingly modest extension to the existing buffer manager in any shared nothing database system today can provide the capability of "selective cache invalidation" which is sufficient to maintain global buffer consistency.

We limit the impact on readers of a writer updating a table via selective cache invalidation. Readers cannot access pages of a table while it is being updated by the writer. But pages of such a table can remain cached. Each writer, at the point when it releases its exclusive lock on a table does two things:

1. it flushes updated pages to disk so that readers, when re-accessing changed pages, find the latest page versions.

2. it sends invalidation notices to all reader nodes, notifying them of the changed pages. Readers invalidate these pages by removing them from their caches, requiring them to re-read these pages from disk when they are accessed again.

3.4 Logging & Recovery

Updates to the stable database can be executed from any node in the cluster so distributed logging and recovery mechanisms are needed for a data sharing system as detailed in Section 2. However, by making careful design decisions we can work with existing unmodified shared nothing logging and recovery protocols.

By providing exclusive access on the database for updates (design consideration (2)), we avoid dealing with multiple logs or with complex and costly interactions using a shared log. An updating node treats the log as if it were a local log. Hence, there are no logging changes required of the shared nothing system. Ownership of the log changes hands when exclusive access for updates to the database changes hands. This restriction also simplifies how we deal with system crashes as well. (1) Only one node needs to be involved in recovery. (2) Reader nodes can, so long as the

Algorithm 1: Read Sequence

```
Acquire S lock on the abstract resource "ServerName.DBName.TableName".
if lock granted then
    execute the read query.
else
    retry lock request.
Release S lock.
```

local node remains up, continue to execute queries on their read-only tables, unaffected by an updater crash. There is no ambiguity about which data is unavailable, it is the tables locked by the updater.

4 Implementation

In this section we present the implementation of a simple data sharing system – Chimera, that uses the techniques presented in Section 3. It is an extension to an existing shared nothing DBMS i.e., MS SQL Server. We use a cluster of PC-based commodity machines each running a stand-alone instance of MS SQL Server hosting one or more databases on local storage. Chimera allows a **remote** SQL Server instance to access a database hosted on a **local** instance. We share the database files among the nodes using CIFS – a network file sharing protocol [10]. CIFS is a commonly used protocol for file sharing over the network, any other similar protocol like NFS may be used as well. This enables multiple nodes to access the same database concurrently over the network, without physical data movement or repartitioning. Since this is not the default behavior, a minor tweak in the server allows the same database to be accessed remotely by multiple SQL Server instances. This setting instantly gives us a read-only data sharing cluster. However, in order to enable write data sharing, we need locks to synchronize access to common data. In addition, we also need to implement mechanisms to maintain global buffer consistency. Chimera achieves this by implementing an enhanced buffer manager (EBM) and a global lock manager (GLM) with corresponding local lock clients (LC). A stored procedure (SP) coordinates locking and buffer management while executing user queries against shared tables. A high-level architecture of a *N*-node data sharing cluster running Chimera is shown in Figure 3.

4.1 Stored Procedure

We implement most of the required changes in a user defined stored procedure that can be invoked like a standard stored procedure. An instance of this stored procedure is installed at all the nodes of the cluster and accepts queries that need to be executed against one of the shared tables. Based on whether it is a read or an update query, the procedure does appropriate locking and buffer management to execute the query. As part of metadata, we maintain the information about other nodes in the system and the databases (or tables) that they host in a SQL table stored locally on each node. Maintaining this information can pose a scalability challenge for a fairly large cluster. However, this is not a new problem in the domain of distributed file systems and we believe that same techniques can be applied here. For the scale of systems that we deal with (a few tens of nodes), any of these techniques are sufficient.

Algorithm 2: Write Sequence

```
Acquire X lock on the abstract resource "ServerName.DBName".
if lock granted then
    Acquire X lock on the abstract resource "ServerName.DBName.TableName".
    if lock granted then
        execute write request
    else
        retry X lock.
else
    retry X lock.
Flush updated pages to disk.
Do selective cache invalidation.
Release all X locks.
```

4.2 Enhanced Buffer Manager

We extend the existing buffer manager in the database engine to implement a cross-node *selective cache invalidation* scheme necessary to maintain global buffer consistency. After an update period, dirty pages are forced to disk at the updating node. The updating node also captures a list of dirty pages and broadcasts an invalidation message to all the readers instructing them to evict these pages from their buffer pool. Only after these pages are evicted at all nodes will the updating node release the write locks, allowing other readers of the same table to proceed. After an update, readers are required to re-read the updated pages from disk. Note that the cache invalidation granularity is at the page-level which is finer than our table-level sharing granularity. This means that after an update only the pages that got dirtied will be evicted. All other pages belonging to the shared table remain in-cache and immediately accessible.

4.3 Global Lock Manager

Chimera uses a global lock manager with local lock clients integrated with each database server instance. Our extended locking scheme is adopted from Boxwood [18] and complements the usual shared nothing locking protocol at each node when data sharing capability is used. Instead of locking a physical entity (e.g., a data or an index page), Boxwood lock server grants or releases a lock in shared (S) or exclusive (X) mode on an abstract resource represented by an arbitrary string. Use of Boxwood lock mechanism is ideal for Chimera as it does not require any changes to the existing lock manager. Also note that any other distributed coordination service such as Chubby [3] or ZooKeeper [11] would work equally well.

Locking Protocol

The locking protocol that we use is as follows: Assuming that a local server has enabled sharing on a table named `TableName` in a database named `DBName`, a reader always acquires a global shared (S) lock on an abstract resource created as "`ServerName.DBName.TableName`", where `ServerName` is the name of the local server hosting a database named `DBName` containing a shared table `TableName`. This naming scheme for abstract resources ensures that shared table names are unique among all nodes of the cluster. On the other hand, writers of a shared table acquire global exclusive (X) locks on resources "`ServerName.DBName`" and "`ServerName.DBName.TableName`" to block other writers of the database and other readers of the table, respectively. This two level locking scheme for writes prevents multiple writers of a database. Readers and writers of the same table always block each other and vice versa. Furthermore, locks are cached at each local lock client until the GLM revokes it, which can happen when there is a conflicting lock request for the same resource.

The pseudocode for read and write sequence implemented in the stored procedure is summarized in Algorithm 1 and Algorithm 2, respectively.

4.4 Recovery Issues

Section 2 describes the complexity of recovery in a data sharing system. The combination of our implementation technique and our restrictions on concurrency permits us to avoid that complexity. Our ownership locking protocol permits only a single writer at a time. That writer has access to the unique log associated with the database for the duration of its update period. The single updater simply uses LSNs associated with that log, its exclusive access to that log, and the "vanilla" recovery of the shared nothing DBMS to provide recovery. This is the "common shared log" approach implemented very simply and, because of our workload restrictions, its overhead for gaining access to that log is tolerable.

5 Performance Results

This section describes our experimental goals, the setup of physical machines used for experiments, and a discussion of the experimental results in detail.

5.1 Experimental Goals

We focus our attention on a small number of issues for which we want to provide quantitative results.

- Scalability: A key advantage of Chimera is that work can be spread among a set of nodes during periods of high load. We want to understand the limits of its scalability and the impact on shared nothing performance.
- Overhead: Chimera can introduce overhead arising from several sources: (1) The cost of executing database functionality from a remote node is likely to be more than that of local execution of the same functionality. (2) The latency of execution depends on the effectiveness of the cache. At startup, a cold cache results in significant overhead due to cache misses, while we expect steady state behavior, with warmed up caches, to be more reasonable. (3) The code changes we introduce into the common case execution path of a shared nothing DBMS may add excessive cost when used.
- Update Cost: Writes introduce a penalty in our system as they impact caching behavior. We wish to quantify this write penalty as well as understand the read/write mix for which we can provide acceptable performance.

5.2 Experimental Setup

We use a 16 node cluster to conduct our experiments. Each node in the cluster has identical hardware and software configurations which include: 2x AMD Opteron CPU @ 2.0GHz, 8GB RAM, running Windows Server 2008 Enterprise with SP2, and Microsoft SQL Server 2008 (SP1) with our modifications.

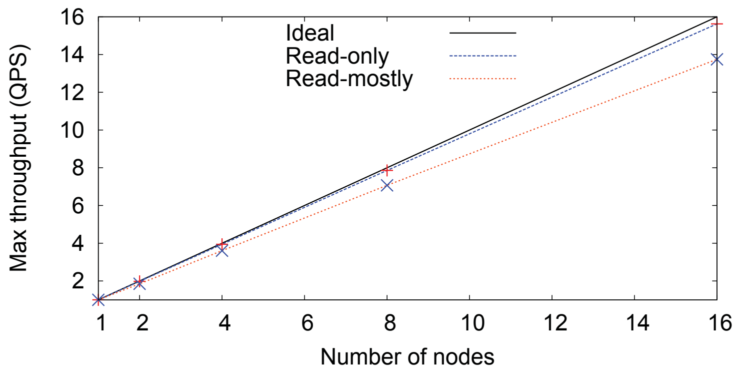
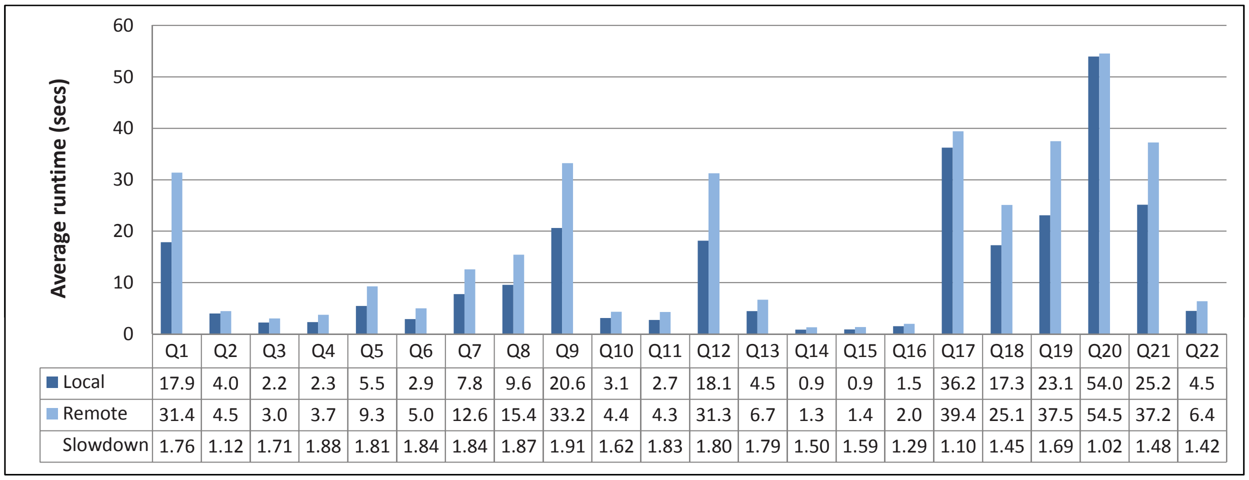


Figure 4: Scalability with an increasing number of nodes.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
■ Local	17.9	4.0	2.2	2.3	5.5	2.9	7.8	9.6	20.6	3.1	2.7	18.1	4.5	0.9	0.9	1.5	36.2	17.3	23.1	54.0	25.2	4.5
■ Remote	31.4	4.5	3.0	3.7	9.3	5.0	12.6	15.4	33.2	4.4	4.3	31.3	6.7	1.3	1.4	2.0	39.4	25.1	37.5	54.5	37.2	6.4
Slowdown	1.76	1.12	1.71	1.88	1.81	1.84	1.84	1.87	1.91	1.62	1.83	1.80	1.79	1.50	1.59	1.29	1.10	1.45	1.69	1.02	1.48	1.42

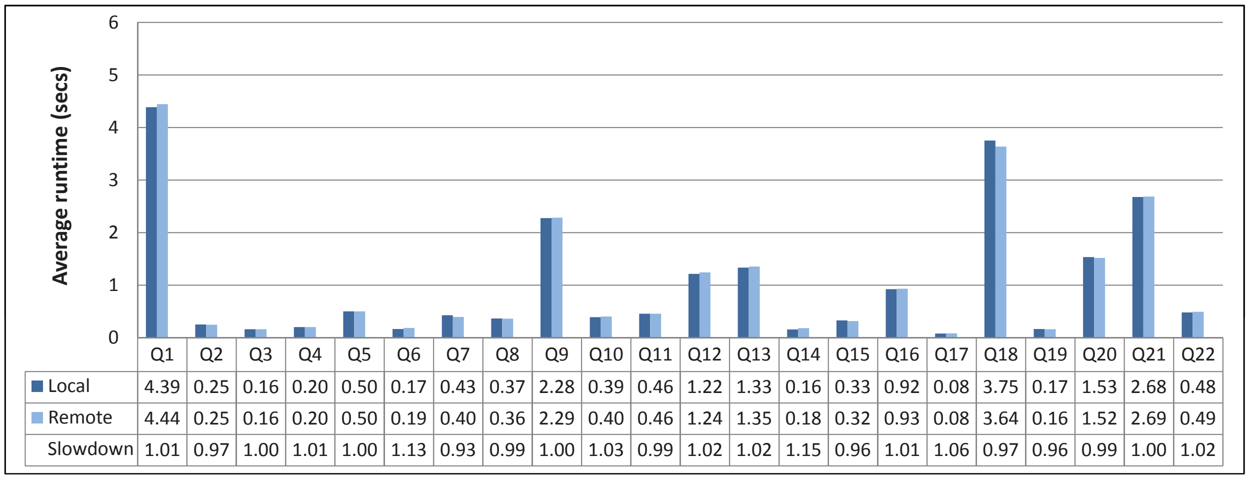
Figure 5: Remote execution overhead: Start up (cold cache).

We use a widely accepted benchmark i.e., TPC-H [27] for our experimental evaluation. TPC-H is a decision support benchmark that consists of a database containing 8 tables and a set of 22 read-only queries modeled after the long running compute intensive queries typical of a data warehousing environment. We use a TPC-H database with scale factor 1. Total size of the database on disk including indexes is approximately 2 GB. We configure SQL Server's buffer pool size to 2 GB so that when the cache is warm the entire database can fit in main memory.

5.3 Scalability

In the first experiment, we measure how performance scales with an increasing number of nodes in the cluster having only a single local node. We run concurrent TPC-H streams first using only a single node (the local node), and then incrementally add more (remote) nodes up to a maximum of 16 nodes. The number of concurrent TPC-H streams is also increased with the number of nodes adding increasingly more load to the cluster. We compare the peak performance in each case. The results for a read-only and a read-mostly workload are presented in Figure 4. These results have been normalized with respect to the maximum throughput achieved using a single node.

Focusing on the read-only case, with two nodes, we achieve a maximum throughput of about 1.95 QPS, which is



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
■ Local	4.39	0.25	0.16	0.20	0.50	0.17	0.43	0.37	2.28	0.39	0.46	1.22	1.33	0.16	0.33	0.92	0.08	3.75	0.17	1.53	2.68	0.48
■ Remote	4.44	0.25	0.16	0.20	0.50	0.19	0.40	0.36	2.29	0.40	0.46	1.24	1.35	0.18	0.32	0.93	0.08	3.64	0.16	1.52	2.69	0.49
Slowdown	1.01	0.97	1.00	1.01	1.00	1.13	0.93	0.99	1.00	1.03	0.99	1.02	1.02	1.15	0.96	1.01	1.06	0.97	0.96	0.99	1.00	1.02

Figure 6: Remote execution overhead: Steady state (warm cache).

nearly twice as much as the single node case. For four nodes, the performance peaks at about 3.81 QPS, which is about twice as that of the two nodes case. We see a similar trend moving forward with 8, and 16 node cluster sizes achieving a max throughput of 7.72, and 14.35 QPS, respectively. For read-mostly case, the TPC-H database is being updated every 15s. In this case, we have the additional overhead of locking and selective cache invalidation which causes the throughput curve to drop below the read-only curve. However, note that Figure 4 shows *almost linear* scalability. In both cases, there is a slight loss of performance due to increased contention with higher number of nodes, but this is to be expected.

Nodes can be added or removed to/from the cluster in an on-demand, and seamless fashion allowing the database to scale-out or shrink on a need basis. This allows for load balancing on a much finer time scale than what is possible with a pure shared nothing DBMS cluster.

5.4 Remote Execution Overhead

Our next experiment measures the overhead of remote execution i.e., overhead of executing a query from a remote node against a shared database hosted on a local node. We run each of the 22 TPC-H queries 5 times, measure the query execution time, and then take the average. To get startup costs, we flush the SQL Server buffer cache between each run. Figure 5 presents the times to run TPC-H queries locally and remotely, as well as the slowdown i.e., the ratio of the remote to the local running time. Most queries experience an overhead ranging from moderate (about 2%) to relatively high overhead (about 91%) when run with a cold database cache. On average it takes about 1.6 times longer to run a query on a remote node as compared to a local node. The worst case occurs for Q9 where the remote execution time is 1.91 times that on the local node. We note that all the high overhead queries access a significant portion of the TPC-H database. For example, Q9 is an aggregation query over more than half the tables of the TPC-H database including the *lineitem* and *orders* table, the two largest tables in the database. This causes the remote node to request a large number of pages across the network from the local node resulting in a high overhead at startup.

We believe that most of the overhead observed in query runtimes presented in Figure 5 can be attributed to the network latency of shipping database pages to the remote system. Next, we conduct an experiment where we repeat the same experimental runs except that we do not clear database caches between each run i.e., the system is already

TPC-H Query	Runtime Without Chimera (ms)	Runtime With Chimera (ms)	Slow Down Factor
Q1	4477	4389	0.98
Q2	279	253	0.91
Q3	182	163	0.90
Q4	199	202	1.02
Q5	504	502	1.00
Q6	166	165	0.99
Q7	356	429	1.21
Q8	329	367	1.12
Q9	2257	2275	1.01
Q10	381	390	1.02
Q11	470	459	0.98
Q12	1209	1217	1.01
Q13	1350	1333	0.99
Q14	155	157	1.01
Q15	315	331	1.05
Q16	922	923	1.00
Q17	76	79	1.04
Q18	3763	3753	1.00
Q19	160	168	1.05
Q20	1524	1533	1.01
Q21	2673	2677	1.00
Q22	460	481	1.05

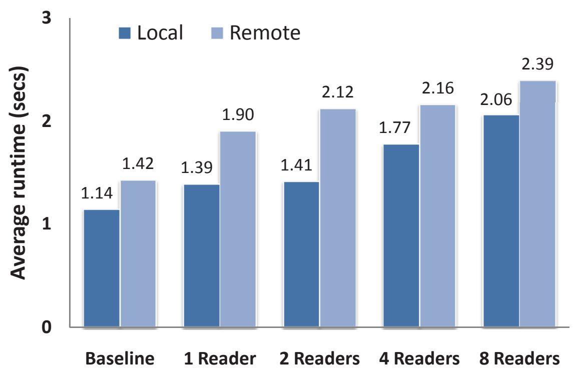
Table 1: Overhead of prototype execution.

in a steady state. Results are presented in Figure 6. In this case queries have about the same runtime on both local and remote nodes with an average overhead of less than 1%. We note that this is the common case for such a data sharing system when executing a read-mostly workload. It is very likely that the active pages of even a fairly large table will entirely fit in the memory. Thus high overhead as presented in Figure 5 should only be incurred at startup, leading to subsequent execution times with small overhead similar to the results presented in Figure 6. In subsequent experiments, unless stated otherwise, the system is in a steady state.

5.5 Prototype Overhead

Next we present an experiment where we measure the overhead of running Chimera. The goal is to show how vanilla SQL Server performs as compared to Chimera. Table 1 shows the runtime and slowdown factor when we run the 22 TPC-H queries on the local node with and without Chimera. Each value presented in Table 1 is an average over 5 runs.

In most of the cases the query runtimes with and without Chimera are about the same. In some cases (e.g., Q8, Q9), query runtime with Chimera is higher than without, indicating a small overhead. Yet in some other cases (e.g., Q2, Q13), queries seem to be running faster with Chimera. However, most of these differences are very small and fall within the boundaries of experimentation error. Averaged over all runs, the overhead is less than 2%. The conclusion that we draw from these numbers is that Chimera adds a negligible overhead to the query runtimes. Moreover, this



Slowdown (local)	1.21	1.24	1.55	1.81
(remote)	1.34	1.49	1.52	1.68

Figure 7: Cost of updates.

extra overhead is incurred only when our data sharing capability is exercised.

5.6 Cost of Updates

Even though Chimera has been designed to provide scalability for read-mostly workloads, it is important to characterize how costly updates will be when readers are present in the system. In this experiment, we run a simple update on the *lineitem* table on the local (host) node and measure its runtime, this serves as a baseline. We consider four different scenarios where 1, 2, 4, or 8 (remote) nodes are reading the shared TPC-H database (including the *lineitem* table) in an infinite loop, while the simple update query is executed on the host node. The results are presented in Figure 7. Each value reported is an average over 5 runs. Slowdown in this case is calculated as the ratio of the runtime in *n* reader case to the baseline runtime.

In the local case, our baseline takes an average of 1.14 seconds. With one reader in the system, the same query takes an average of 1.39 seconds, which is about 1.21 times the baseline runtime. With two readers, the runtime is about 1.24 times the baseline. We see a similar trend for 4, and 8 readers, with 1.55 and 1.81 slowdown factors, respectively. At the updating node, each additional read node adds to the overhead of acquiring X locks, as required by our locking protocol. The global lock server has to wait for each reader node of the *lineitem* table to release the S lock on the shared resource before it can grant the X lock to the writer node. This adds some extra overhead with increasing number of nodes, as shown in Figure 7 – local case.

In the next experiment we repeat the previous experiment except that the update is performed on one of the remote nodes – this is our new baseline. The results, also presented in Figure 7 – remote case, show a similar pattern as the local case. The difference in this case is that the average time it takes to complete the update is slightly higher in all scenarios as compared to the local case. This increase is a result of the additional cost of accessing data from the local node because the update is executed on a remote node.

Results presented in this section show that the cost for acquiring X locks in our system increases sub-linearly with the cluster size. This experiment illustrates that there is a moderate cost to execute updates with conflicting reads, which is acceptable for our target read-mostly workloads.

	Update Freq. (sec)	Queries Completed	Avg Runtime (ms)	Steady State Avg (ms)	QPS
Q6	60	1455	206	187	4.85
	30	1363	215		4.54
	15	1269	234		4.23
	5	1130	264		3.77
Q13	60	220	1376	1353	0.73
	30	220	1375		0.73
	15	216	1388		0.72
	5	218	1376		0.73
Q20	60	185	1616	1518	0.62
	30	180	1654		0.60
	15	148	1991		0.49
	5	146	2039		0.49
Q21	60	107	2783	2687	0.36
	30	104	2853		0.35
	15	100	3011		0.33
	5	83	3597		0.28

Table 2: Cost of reads with updates.

5.7 Cost of Reads with Updates

In our next experiment, we quantify the cost of reads while at least one of the nodes (the local node in our case) is updating the database at varying intervals of 60, 30, 15, and 5 seconds. We run one of the TPC-H read queries for a fixed duration of 300 seconds and measure average response time, the number of queries completed, and the number of queries executed per second (QPS). Results are presented in Table 2 for TPC-H query 6, 13, 20, and 21. Query 6, 20, and 21 conflict with the update query (i.e., read the same table) while query 13 performs a non-conflicting read. We have chosen this set to present a mix of queries with short to relatively long runtimes and with conflicting/non-conflicting reads to show how various update frequencies affect each group. Results with other TPC-H queries (not presented here) follow the same general trend.

The results for Q6 show that if updates are $\geq$ 60s apart, we are able to achieve performance very close to the quiescent state when updates are entirely absent. As we decrease the update interval from 60s down to 5s, we see a gradual decrease in performance resulting in a slightly higher average runtime (or low QPS), which is to be expected. However, note that the performance is adequate even at high update frequencies (i.e., every 5s). We see a similar trend for Q20 and Q21. Note that the overhead in these results includes the overhead of locking and selective cache invalidation. For Q13, the performance remains constant at all update frequencies given that it performs a non-conflicting read.

This experiment proves that it is possible to provide adequate performance for scenarios where reads run for relatively long intervals between writes – our target read-mostly workload.

6 Related Work

There is much work in the literature that is aimed at facilitating data sharing, much like the goals of our system. Chimera technology has some similarity to the work in the areas of distributed file systems (DFS), global lock servers, replicated databases, and scalable cloud stores.

Parts of Chimera are reminiscent of distributed file systems (DFS), e.g., [26, 24]. In particular, our locking strategy is similar, although there are differences in our system to provide the requisite database functionality. Our lock service is similar in spirit to modern file system lock managers (e.g., [3, 11]). Furthermore, the caching of pages in the buffer pool in Chimera shares aspects with the management of data in a DFS file cache. However, the details of the design are considerably different due to the complexities imposed by the database semantics compared to the simpler file and directory abstractions supported by a DFS.

Our approach to load balancing is superficially similar to replication used for the same purpose by some database vendors [17, 19, 21]. A replicated database can be used effectively to balance load by sending read requests to either or both replicas, making it similar in spirit to our system. However, write requests have to be propagated to both replicas and does not really benefit load balancing. Chimera is complementary to replicated databases in that it can be used in conjunction with replicated databases to balance load. In particular, Chimera can spread load to all server nodes, even those that do not hold a replica. This is in contrast to systems that use replication alone because requests can only be handled by the nodes that hold the replica.

Finally, providing scalability for large data sets is a property of cloud infrastructures like Big Table [4] and PNUTS [5]. One important difference here is that these infrastructures, while supporting atomic operations, do not support transactions or database functionality. Chimera fully supports transactions and provides database functionality. Furthermore, of course, Chimera is not a cloud-based system, and hence can be instantiated at existing customer sites. And customers can use it to better support their user base on their existing infrastructure under heavy or varying load.

7 Summary

Chimera demonstrates the design and implementation of data sharing as an extension to a shared nothing system to get the best of both worlds – data sharing flexibility with shared nothing simplicity. Chimera uses a hybrid database architecture to enable sharing at the granularity of tables, and has been designed as a simple extension to a shared nothing database system. It uses low-cost, coarse-grained global locks to control how data is shared among nodes and selective cache invalidation for global buffer consistency. Chimera demonstrates the ability to move database functionality from a local host node to a remote node for load balancing without moving tables between the disks of these nodes. This permits load balancing to respond in a matter of minutes not hours. Using a TPC-H database and 22 read-only queries, we show that Chimera is able to serve read-mostly workloads with acceptable performance while providing nearly linear scalability with increasing number of nodes.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y.Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] P. Bruni, R. Cornford, R. Garcia, S. Kaschta, and R. Kumar. *DB2 9 for z/OS Technical Overview*. IBM Redbooks, 2007.

[3] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, September 2006.

[4] 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 *Proceedings of the 7th Conference on Symposium on Operating Systems Design & Implementation (OSDI)*, pages 205–218, November 2006.

[5] 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. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*, pages 1277–1288, August 2008.

[6] M. Devarakonda, B. Kish, and A. Mohindra. Recovery in the Calypso file system. *ACM Transactions on Computer Systems*, 14(3):287–310, August 1996.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, October 2003.

[8] A. C. Goldstein. The design and implementation of a distributed file system. *Digital Technical Journal*, 1(5):45–55, September 1987.

[9] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th Symposium on Operating Systems Principles (SOSP)*, pages 202–210, December 1989.

[10] C. R. Hertel. *Implementing CIFS: The Common Internet File System*, chapter Introduction. Prentice Hall, 2003.

[11] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the USENIX annual technical conference (USENIXATC)*, pages 11–11, June 2010.

[12] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, February 1998.

[13] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92, October 1996.

[14] D. Lomet. Recovery for shared disk systems using multiple redo logs. Technical Report 4, Digital Cambridge Research Lab, 1990.

[15] D. Lomet. Private locking and distributed cache management. In *Proceedings of the 3rd International Conference on Parallel and distributed information systems (PDIS)*, pages 151–159, September 1994.

[16] D. Lomet, R. Anderson, T. K. Rengarajan, and P. Spiro. How the Rdb/VMS data sharing system became fast. Technical Report 2, Digital Cambridge Research Lab, 1992.

[17] K. Loney. *Oracle Database 11g The Complete Reference*. McGraw-Hill, 2008.

[18] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI)*, pages 8–8, December 2004.

[19] R. B. Melnyk and P. C. Zikopoulos. *DB2: The Complete Reference*. McGraw-Hill, 2001.

[20] C. Mohan and I. Narang. Efficient locking and caching of data in the multisystem shard disks transaction environment. In *Proceedings of the 3rd International Conference on Extending Database Technology (EDBT)*, pages 453–468, 1992.

[21] Microsoft SQL Server 2008. Microsoft. [online] http://www.microsoft.com/sqlserver/2008/en/us/default.aspx.

[22] T. Rengarajan, P. Spiro, and W. Wright. High availability mechanisms of VAX DBMS software. *Digital Technical Journal*, 1(8):88–98, February 1989.

[23] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network file system. In *Proceedings of Summer UNIX*, June 1985.

[24] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st Conference on File and Storage Technologies(FAST)*, pages 231–244, January 2002.

[25] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9:4–9, 1986.

[26] C. A. Thekkath, T. P. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP)*, pages 224–237, October 1997.

[27] TPC-H: An Ad-hoc, Decision Support Benchmark. [online] http://www.tpc.org/tpch/.