Citrusleaf: A Real-Time NoSQL DB which Preserves ACID

V. Srinivasan
Citrusleaf, Inc.
444 Castro Street, Suite 703
Mountain View, CA 94041
+1 650-336-5323
srini@citrusleaf.com

Brian Bulkowski
Citrusleaf, Inc.
444 Castro Street, Suite 703
Mountain View, CA 94041
+1 650-336-5323
brian@citrusleaf.com

ABSTRACT

In this paper, we describe the Citrusleaf real-time distributed database platform that is built using the core principles of traditional database consistency and reliability while also being fast and flexible enough for use in high-performance applications like real-time bidding. In fact, Citrusleaf is unique among NoSQL databases for its ability to provide immediate consistency and ACID while still being able to consistently exceed the high performance and scalability standards required by demanding real-time applications. This paper describes how the Citrusleaf system achieves the marriage of traditional database reliability, including immediate consistency and ACID, with flexibility and operational efficiency.

Citrusleaf scales linearly at extremely high throughput while keeping response time in the sub-millisecond range as demonstrated by the test results presented here. This kind of performance has enabled Citrusleaf to become the underlying component of some of the world's largest real-time bidding networks.

1. INTRODUCTION

Over the past several years, there has been an explosion in the growth of internet applications for the Real Time Web [15] (e.g., real-time advertising, mobile, location based apps, real-time feeds from social networks, etc.). The limitations of the existing databases to handle the high write loads inherent in many of these applications was initially encountered by major internet companies like Amazon, Facebook, Google, Yahoo!, etc., but these limitations are now being routinely encountered in many small and medium size companies that desire to build a meaningful real-time internet service to leverage the relentlessly explosive growth in internet usage. Here are a few examples:

- Real-time bidding (RTB) platforms for display advertising require an extremely scalable and cost effective high performance transaction system.
- Real-time campaign management for advertising and marketing campaigns requires minute by minute analysis of data that is changing as users navigate these campaigns.
- Social networking applications, especially social gaming require real-time exchange of data between users that are taking part in various games

Of late, several new databases [11], [20], [16], etc. have emerged out of several independent efforts to provide a scalable, flexible database alternative that can effectively address the needs of these high volume internet applications. Citrusleaf is also in this category of products.

Many of these new databases are built on extremely solid networking and distribution technologies but have diverged significantly from traditional database techniques. E.g., some of these systems support a technique called "eventual consistency" [1], where an update that was completed in the system could occasionally disappear from the view of other readers until it eventually reappears at some time in the future (hence the name, eventual consistency). While eventual consistency may be sufficient for certain applications (e.g., shopping carts on a web site), this sort of non-deterministic behavior creates enormous challenges for application developers who now have to handle complex failure cases themselves.

Our premise, however, is that it is possible (and imperative) to build clustered database systems that incorporate the best in networking technologies [6] (these are "table stakes" in the new world) while also retaining the robust concurrency and recovery practices used in traditional databases. Therefore, Citrusleaf is a product focused on maintaining the high performance and scalability of NoSQL solutions while also sticking to the time tested DBMS fundamentals like ACID, immediate consistency, backup and restore, high availability, etc. Most of the techniques described in this paper have been validated in mission-critical, internet scale deployments over a twelve month period.

The rest of the paper is organized as follows. In Section 2, we provide an overview of system architecture. Section 3 describes the key technology behind the system. In Section 4, we present results to demonstrate the linear scalability of Citrusleaf. In Section 5, we compare and contrast Citrusleaf with the various other products in this space. We describe future directions in Section 6 and present the conclusions in Section 7.

Proceedings of the VLDB Endowment, Vol. 4, No. 12

2. System Architecture

The Citrusleaf database platform (Figure 1) is based on the classic shared-nothing database architecture [7]. The database cluster consists of a list of commodity server nodes, each of which has CPU, DRAM, rotational disk (HDD) and optional flash storage (SSD). These nodes are connected to each other using a standard TCP/IP network.

In Citrusleaf, as in traditional databases, there is strict separation at the network level between the client and the server. The Citrusleaf client typically runs on the same node as the application and is usually tightly integrated with the application. One of the fundamental ways in which Citrusleaf differs from other comparable systems is its ability to use client-side load balancing to vastly increase transaction performance and achieve smooth linear scalability.

We will first describe the database cluster architecture and then the client layer.

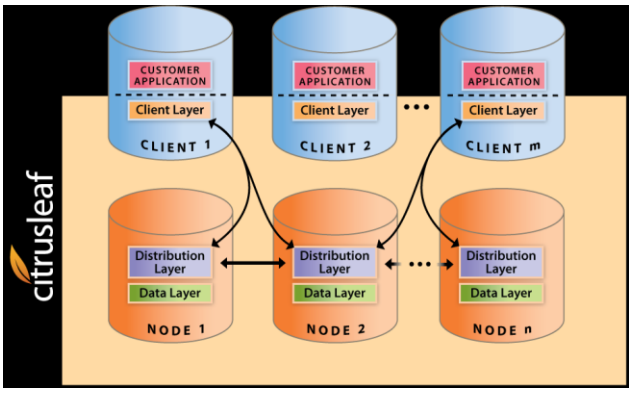

Figure 1: Citrusleaf Architecture.

2.1 Database Cluster Architecture

Each node in the database cluster comprises of two layers, the distribution layer and the data layer. These are explained in more detail below.

2.1.1 Distribution Layer

The Distribution Layer (Figure 2) is responsible for both maintaining the scalability of the Citrusleaf clusters, and for providing many of the ACID reliability guarantees. The implementation of the Distribution Layer is 'shared nothing' [7]. This means that there are no centralized 'managers' of any sort, eliminating bottlenecks, inefficient resource usage and single points of failure such as those often created by master/slave relationships.

Citrusleaf uses standard network components and therefore all communication in the system happens via TCP/IP. We have found that in modern Linux environments, TCP/IP requests can be coded in a way that allows many thousands of simultaneous connections at very high bandwidths. We have not found the use of TCP/IP to impact system performance when using Gigabit-Ethernet connections between components.

There are three major modules within the Distribution Layer – the Cluster Administration Module, the Data Migration Module, and the Transaction Management Module. These are discussed in more detail below.

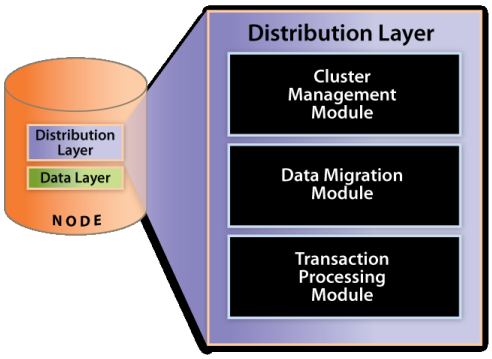

Figure 2: Distribution Layer.

2.1.1.1 Cluster Administration Module

The Cluster Administration Module is a critical piece of both the scaling and reliability infrastructure, since it determines which nodes are currently in the cluster. Each node periodically sends out a heartbeat to all the other nodes, informing them that it is alive and functional. If any node detects a new node, or fails to receive heartbeats from an existing node, that node's Cluster Administration Module will trigger a Paxos [8,9] consensus voting process between all the cluster nodes. This process determines which nodes are considered part of the cluster, and ensures that all nodes in the cluster maintain a consistent view of the system. The Cluster Administration Module can be set up to run over multicast IP (preferred for zero-config cluster management) or unicast IP (requiring slightly more configuration).

To increase reliability in the face of heavy load - when heartbeats could be delayed - the system also counts any transactional requests between nodes as secondary heartbeats.

Once membership in the cluster has been agreed upon, the individual nodes use a distributed hash algorithm to partition the primary index space into 'slices' and subsequently assign read and write masters and replicas to each of the slices. Because this partitioning is purely algorithmic, the system scales without a master and there is no need for additional configuration at the application level that is required in a non-clustered environment (e.g., sharding [19]). After cluster reconfiguration, data migration between the slices is handled by the Data Migration Module, below.

2.1.1.2 Data Migration Module

When a node is added or removed from the cluster, the Data Migration Module is invoked to rebalance the data within the cluster as determined by the Cluster Administration Module described in the previous Section. The Data Migration Module is responsible for ensuring that the multiple copies of every data item eventually reaches the correct cluster nodes. Note that Citrusleaf supports keeping more than two copies of a data item but most installations keep just two. This data migration process is completely transparent to both the Client and the Application.

Note that a naïve data migration scheme could take a lot of system resources. For example, adding a fourth node to a three

node cluster results in 50% of the data in the cluster moving to the new node. If all the existing nodes start to send the data at full throttle to the new node, it is quite likely the new node could go down as soon as it comes up! Therefore, the Data Migration Module, like much of the Citrusleaf code, has been carefully constructed to ensure that loading in one part of the system does not cause overall system instability. Transactions and heartbeats are prioritized above data migration, and the system is capable of fulfilling transactional requests even when the data has not been migrated to the 'correct' node (this is explained in more detail below in Section 2.1.1.3). This prioritization ensures that the system stays 100% available to application level transactions even while nodes are being added to or removed from the cluster.

Using a stateless deterministic hashing algorithm for assigning slices to nodes is not necessarily optimal and it could result in more rebalances than those that would result by using other algorithms (e.g., linear hashing). However, we have found that rebalancing is relatively rare in production systems that have been running for over a year and, therefore, the stateless approach has worked very well in practice.

2.1.1.3 Transaction Processing Module

The Transaction Processing Module provides many of the consistency and isolation guarantees of the Citrusleaf system. This module processes the transaction requests from the Client, including resolving conflicts between different versions of the data that may exist when the system is recovering from being partitioned.

In the most common case, the client has correctly identified the node responsible for processing the read or write transaction. In this situation, the Transaction Processing Module looks up the data, applies the appropriate operation (read or write) and returns the result to the client. If the request modifies data, the Transaction Processing Module also co-ordinates the changes to multiple copies of this data item thus ensuring immediate consistency.

Note that any node in the cluster has the ability to execute transactions for any data item using an intra cluster proxy mechanism. This is because, occasionally, the node that receives the transaction request will not contain the data needed to complete the transaction. This typically happens during the brief period after the arrival or departure of a node, when the client's routing tables to the cluster may briefly be out of date. In this situation, the Transaction Processing Module from the first node forwards the transaction to the Transaction Processing Module of the node that is responsible for the data item referenced by the client. Once the transaction is completed, the node contacted by the client fetches the transaction result from the node that actually executed the transaction and returns it to the client. The fact that the transaction was actually executed on a different cluster node is completely transparent to the client and is handled by the cluster itself.

Finally, the Transaction Processing Module is responsible for resolving conflicts that are created after cluster nodes rejoin after a network partitioning event that can cause a cluster to split (partition) into two (or more) separate running sub-clusters. Multiple conflicting writes to the same data item in different clusters need to be resolved. More on this topic is discussed later in Section 3.1.2.

2.1.2 Data Layer

The Data Layer (Figure 3) holds the indexes and data stored in each node, and handles interactions with the physical storage. It also contains modules that automatically remove expired data from the database (Citrusleaf supports an optional time-to-live, ttl, setting for each data item that can be set by the Application), and defragment the physical storage to optimize disk usage. Before discussing these components, let's first take a look at the Citrusleaf Data Model.

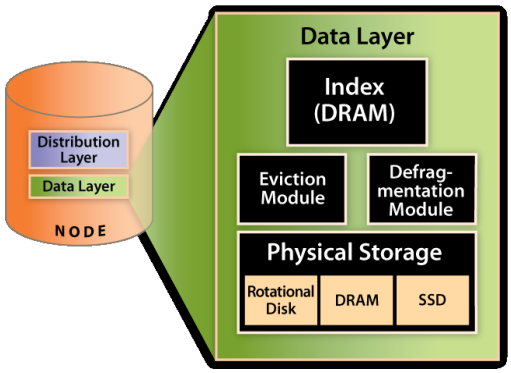

Figure 3: Data Layer.

2.1.2.1 Data Model

The Citrusleaf system is fundamentally a key-value store where the keys can be associated with a set of named values (similar to a 'row' standard RDMBS terminology.)

At the highest level, data is collected into policy containers called 'namespaces', semantically similar to 'databases' in an RDBMS system. Namespaces are configured when the cluster is started, and are used to control expiry, replication, and storage settings for a given set of data. For example, keeping more copies of the data allows you to trade increased storage requirements for improved availability during unexpected hardware failures that take out one more nodes in a cluster.

Within a namespace, the data is subdivided into 'sets' (similar to 'tables') and 'records' (similar to 'rows'). Each record has an indexed 'key' that is unique in the set, and one or more named 'bins' (similar to columns) that hold values associated with the record. Values in the bins are strongly typed, and can include strings, integers, and binary data, as well as language-specific binary blobs that are automatically serialized and de-serialized by the system. Note that although the values in the bins are typed, the bins themselves are not – the same bin value in one record may have a different type than the bin value in different record.

Although these structures may seem at first glance to be very similar to the familiar RDBMS structures, there are important differences. Most importantly, unlike RDBMS systems, the Citrusleaf system is entirely schema-less. This means that sets and bins do not need to be defined up front, but can be added

during run-time thus providing maximum flexibility for applications. Note that having arbitrary schema will result in increased run-time overhead for maintaining indexes. Citrusleaf, therefore, provides special optimization in cases where specific schema simplifications are present – e.g., single column namespaces have been used widely in several deployments due to their enormous efficiencies in both storage and performance.

Each record also has hidden fields like generation, ttl, etc. that enables the system to efficiently implement CAS (check and set) [10] and data expiry.

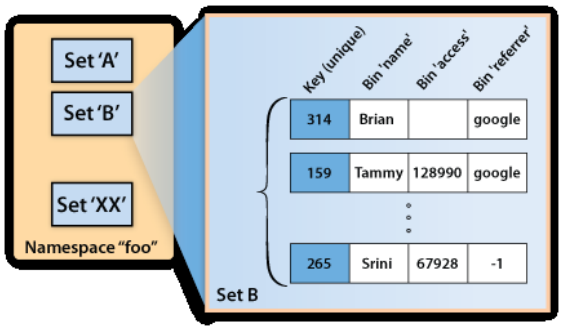

Figure 4: Data Model.

2.1.2.2 Data Storage

Citrusleaf can store data in DRAM, traditional rotational media, and SSDs, and each namespace can be configured separately. This configuration flexibility allows an application developer to put a small namespace that is frequently accessed in DRAM, but put a larger namespace in less expensive storage such an SSD. Significant work has been done to optimize data storage on SSDs, including bypassing the file system to take advantage of low-level SSD read and write patterns.

Citrusleaf's data storage methodology is optimized for fast transactions. Indices (via the primary Key) are stored in DRAM for instant availability, and data writes to disk are performed in large blocks to minimize latencies that occur on both traditional rotational disk and SSD media. The system also can be configured to store data in direct format – using the drive as a low-level block device without format or file system – to provide an additional speed optimization for real-time mission critical systems.

Because storing indices in DRAM impacts the amount of DRAM needed in each node, the size of an individual index entry per data item has been painstakingly minimized. Citrusleaf's indexing scheme allows keys of arbitrary sizes while storing only a fixed length digest as part of the index. At present, Citrusleaf can store indices for 100 million records in 7 gigabytes of DRAM.

2.1.2.3 Defragmenter and Evictor

Two additional processes – the Defragmenter and the Evictor – work together to ensure that there is space both in DRAM and disk to write new data. The Defragmenter tracks the number of active records on each block on disk, and reclaims blocks that fall below a minimum level of use.

The Evictor is responsible for removing references to expired records and for reclaiming memory if the system gets beyond a set high water mark. When configuring a namespace, the administrator specifies the maximum amount of DRAM used for that namespace, as well as the default lifetime for data in the namespace. Under normal operation, the Evictor looks for data that has expired, freeing the index in memory and releasing the record on disk. The Evictor also tracks the memory used by the namespace, and releases older, although not necessarily expired, records if the memory exceeds the configured high water mark.

By allowing the Evictor to remove old data when the system hits its memory limitations, Citrusleaf can effectively be used as an LRU cache.

Note that the age of a record is measured from the last time it was modified, and that the Application can override the default lifetime any time it writes data to the record. The Application may also tell the system that a particular record should never be automatically evicted.

2.2 Client Architecture

Citrusleaf provides a 'smart client' layer between the application and the server. This 'smart client' handles many of the administrative tasks needed to manage communication with the node – it knows the optimal server for each transaction, handles retries, and manages any cluster reconfiguration issues in a way that is transparent to the application. This is done to improve the ease and efficiency of application development – developers can focus on key tasks of the application rather than database administration. The Client also implements its own TCP/IP connection pool for further transactional efficiency.

The Client Layer itself consists only of a linkable library, the 'Client', which talks directly to the cluster. This again is a matter of operational efficiency – there are no additional cluster management servers or proxies that need to be set up and maintained.

Note that Citrusleaf Clients have been optimized for speed and stability; however, developers are welcome to create new clients, or to modify any of the existing ones for their own purposes. Citrusleaf provides full source code to the Clients, as well as documentation on the wire protocol used between the Client and servers. Clients are available in many languages, including C, C#, Java, Ruby, PHP and Python.

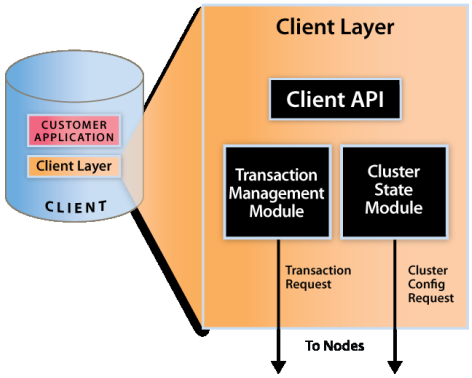

Figure 5: Client Architecture.

The Client Layer has the following responsibilities:

- Providing an API for the Application
- Tracking cluster configuration
- Managing transactions between the Application and the Cluster

Each of these responsibilities is discussed in more detail below.

2.2.1 Providing an API

Citrusleaf provides simple and straightforward interface for reading and writing data. The underlying architecture is based around a key-value store where the 'value' may actually be a *set* of named values, similar to columns in a traditional RDBMS. Developers can read or write one value or multiple values with a single API call. In addition, Citrusleaf implements optimistic locking to allow consistent and reliable read-modify-write cycles without incurring the overhead of a lock. Additional operations available include batch processing, auto-increment, and reading or writing the entire contents of the database. This final operation – reading and writing the entire database – is used for online backup and restore.

The APIs also provide several optional parameters that allow application developers to modify the operation of transaction requests. These parameters include the request timeout (critical in real-time operations where transactions are only valid if they can be completed within a specified time) and the policy governing automatic retry of failed requests.

For more information on the data model underlying the APIs, see Section 2.1.2.

2.2.2 Tracking Cluster Configuration

To ensure that requests are routed to the optimal cluster node, the Client Layer tracks the current configuration of the server cluster using the *info* protocol. To do this, the Client communicates periodically with the cluster, maintaining an internal list of server nodes. Any changes to the cluster size or configuration are tracked automatically by the Client, and such changes are entirely transparent to the Application. In practice, this means that transactions will not fail during the transition, and the Application does not need to be restarted during node arrival and departure.

2.2.3 Managing transactions

When a transaction request comes in from the application, the Client formats that request into an optimized wire protocol for transmission to the servers and sends it to the server that is most likely to contain the requested data.

As part of transaction management, the Client maintains a connection pool that tracks the active TCP connections associated with outstanding requests. It uses its knowledge of outstanding requests to detect transactional failures that have not risen to the level of a server failure within the cluster. Depending on the desired policy, the client will either automatically retry failures or immediately notify the Application of the transaction failure. If transactions on a particular node fail too often, the Client will attempt to route requests that would normally be handled by that node to a different node that also has a copy of the requested data. Note that read requests can be satisfied by any node that has a copy of the data, while write requests require access to the node that contains the master copy of the specific record. This strategy provides an additional level of reliability and resilience for reads when dealing with transient connectivity issues.

2.3 Summary

The Citrusleaf architecture is derived from three core principles – NoSQL scalability and flexibility, traditional database consistency and reliability, and cluster self-management. These principles are demonstrated in the shared-nothing distribution architecture, the schema-less data framework, the insistence on immediate consistency and atomicity, and system-wide fault-tolerance. In addition, the architecture is geared to operational efficiency, both in its ease of use for application developers and system administrators, and in its speed and low resource overhead requirements when running on off-the-shelf Linux environment.

3. Technology

Citrusleaf technology combines classic DB techniques with the latest in networking and distributed technology while still providing extremely high performance. We will describe here how Citrusleaf implements ACID and how it supports scalability and high performance.

3.1 ACID

Citrusleaf is intended to outperform traditional databases by an order of magnitude in the mission-critical environments where that performance is most needed: e.g., the high volume of frequently updated data that drives the front end of a business. We will briefly describe how Citrusleaf is optimized to squeeze as much transaction throughput as possible while still guaranteeing strong consistency (ACID) to make application development easy.

3.1.1 Atomicity

For read/write operations on a single record, Citrusleaf makes strict guarantees about the atomicity of these operations as follows:

- Each operation on a record is applied atomically and completely. For example, a read from or a write to multiple bins in a record is guaranteed a consistent view of the record.
- After a write is completely applied and the client is notified of success, all subsequent read requests are guaranteed to find the newly written data: there is no possibility of reading stale data. Therefore, Citrusleaf transactions provide immediate consistency.

In addition to single record operations, Citrusleaf supports distributed multi-key transactions using a simple and fast iteration interface where a client can simply request all or part of the data in a particular set. This mechanism is currently used for database backup and basic analytics on the data and delivers extremely high throughput. Single key transactions are serialized with respect to a multi-key transaction but two multi-key operations may not be serialized with each other.

3.1.2 Consistency

For operations on single keys, Citrusleaf provides immediate consistency using synchronous replication.

Multi-key transactions are implemented as a sequence of single key operations and do not hold record locks except for the time required to read a clean copy. Thus the multi-key transaction provides a consistent snapshot of the data in the database (i.e., no "dirty reads" are done).

Citrusleaf's support for relaxing consistency models gives operators the ability to maintain high performance during the cluster recovery phase after node failure. E.g., read and write transactions to records that have unmerged duplicates in the cluster can be sped up by bypassing the duplicate merge phase.

In the presence of failures, the cluster can run in one of two modes - Partition Tolerant or High Consistency. The difference between the two modes is seen only for a brief period during cluster recovery after a node failure. Citrusleaf is highly consistent when the cluster does not split.

3.1.2.1 Partition Tolerance

In Partition Tolerant mode, when a cluster splits, each faction of the cluster continues operating. One faction - or the other - may not have all of the data, so an application reading data may have successful transactions stating that data is not found in the cluster. Each faction will be in the process of obeying the replication factor rules, thus replicating data, and may accept writes from clients. Application servers which read from the other faction will not see the applied writes, and may write to the same primary keys. If, at a later point, the factions rejoin, data which has been written in both factions will be detected as inconsistent. Two policies may be followed. Either Citrusleaf will 'auto-merge' the data by favoring the last write (the write with the latest server timestamp), or both copies of the data will be retained. If two copies - versions - of the data are available in the cluster, a read of this value will return both versions, allowing the application to resolve the inconsistency. The client application - the only entity with knowledge of how to resolve these differences - must then re-write the data in a consistent fashion.

3.1.2.2 High Consistency

In High Consistency mode, when the cluster splits, the cluster with a minority quorum could be made to halt. This action prevents any client from receiving inconsistent data, but will reduce availability.

3.1.3 Isolation

Citrusleaf implements distributed isolation techniques consisting of latches and short-term record locks to ensure isolation between multiple transactions. Therefore, when a read and a write operation for a record are pending simultaneously, they will be internally serialized before completion, though their precise ordering is not guaranteed.

For enabling simple multi-record transactions, Citrusleaf supports an optimistic concurrency control scheme based on atomic conditional operations (CAS - Check and Set [10]), making the very common read-modify-write cycle safe -- without the often-crippling overhead of explicit locking. In many simplistic data storage systems, reading a data element, modifying it, and then writing it back exposes a race condition that could lead to data corruption during highly concurrent access to the data item.

For multi-key operations, one of the cluster nodes anchors the iteration operation and requests data from all the other nodes in parallel. Snapshots are taken of the indexes at various points to allow minimal lock hold times. As data is retrieved in parallel from the working nodes, it is forwarded to the client. Care is taken that the client is not overwhelmed by a flood of responses from multiple cluster nodes. Therefore, the client-server protocol has flow control features that the client uses to regulate the responses from the multiple nodes in the cluster that are working on the distributed transaction.

Any client-server system suffers from the client being potentially disconnected from the server at any time. This can result in the client being unable to distinguish whether a transaction in flight has been applied or not. Recovery from this situation may be quite complex. Citrusleaf supports mechanisms for retrying writes and using a client generated unique persistent transaction identifier that enables clients to properly determine if the transaction has completed properly.

3.1.4 Durability

In order to keep your data always available, Citrusleaf provides multi-server replication of your data. The cluster is configured to contain multiple namespaces (like 'databases' within a RDBMs), and each namespace contains configuration of the storage system, and storage policies, for the data contained in that namespace.

The basic mechanisms for providing durability in Citrusleaf consist of replication to multiple nodes using both DRAM and persistent storage. Therefore, every transaction update is written to multiple locations on the cluster before returning to the client. For example, in a persistent namespace that stores data in both DRAM and disk with a replication factor 2, the record is stored in four locations, two copies in DRAM on two nodes and two additional copies on disk.

3.1.4.1 Resilience to hardware failures

In the presence of node failures, clients are able to seamlessly retrieve one of the copies of the data from the cluster with no special effort. This is because, in a Citrusleaf cluster, the virtual partitioning and distribution of data within the cluster is completely invisible to the client. Therefore, when client libraries make calls using a lightweight Client API to the Citrusleaf cluster, any node can take requests for any piece of data.

If a cluster node receives a request for a piece of data it does not have locally, it satisfies the request by internally creating a proxy request, fetching the data from the real owner using the internal cluster interconnect and subsequently replying to the client directly. The Citrusleaf client-server protocol also implements caching of latest known locations of client requested data in the client library itself to minimize the number of network hops required to respond to a client request.

During the period immediately after a cluster node has been added or removed, the Citrusleaf cluster automatically transfers data between the nodes to rebalance and achieve data

availability. During this time, Citrusleaf's internal "proxy" transaction tracking allows high-consistency to be achieved by applying reads and writes to the cluster nodes which have the data, even if the data is in motion. The proxy mechanism ensures that the client does not have to handle dynamic redirection of requests, i.e., once a client makes a request of any node of the cluster, it is guaranteed to receive the response from that node even if the data itself is located in a different node within the cluster.

3.1.4.2 Backup and Recovery

Citrusleaf provides online backup and restore, which, as the name indicates, can be applied while the cluster is in operation. Even though data replication will solve most real-world data center availability issues, an essential tool of any database administrator is the ability to backup and restore. A Citrusleaf cluster has the ability to iterate all data within a namespace (similar to a map/reduce). The backup and restore tools are typically run on maintenance machines with a large amount of inexpensive, standard rotational disk.

Citrusleaf backup and restore tools are made available with full source. The file format in use is optimized for high speed but uses an ASCII format, allowing an operator to validate the data inserted into the cluster, and use standard scripts to move data from one location to another. The backup tool splits the backup into multiple files to allow restores to occur in parallel from multiple machines during a rapid response to a catastrophic failure event.

3.2 High Performance Transaction Processing

Citrusleaf uses a distributed hash table with a two level hashing scheme. The key space is first separated out into a large number of partitions. The second hash function distributes the partitions among nodes in the cluster. This enables transactions to be concurrently executed at an extremely high rate.

Citrusleaf further speeds up transaction processing as follows:

- The basic server code is written in the C language, using the same principles underlying an operating systems kernel and this enables extremely fast dispatching of tasks within the Citrusleaf server.
- Using the info protocol, the Citrusleaf client is aware of the assignment of partitions within the cluster nodes. Therefore, every client is able to efficiently route requests to the appropriate node within the cluster that proceeds to handle the transaction with the minimal number of network hops. In fact, this optimization is one reason why Citrusleaf is able to support immediate consistency and still provide the extremely high throughput needed for NoSQL applications.
- Support for relaxing consistency models gives operators the ability to maintain high performance during the cluster recovery phase after node failure. E.g., read and write transactions to records that have unmerged duplicates in the cluster can be sped up by bypassing the duplicate merge phase.
- Multi-key transactions are implemented as a sequence of single key operations and do not hold record locks except for the time required to read a clean copy. Thus the multi-key transaction provides a loosely consistent snapshot of the data in the database, i.e., no "dirty reads" are done but the snapshot may not be serialized with respect to other concurrent multi-key transactions.
- A sophisticated real-time prioritization algorithm is used to balance long running tasks versus client transactions. This scheme prioritizes fast read and write transactions higher than long running tasks like data rebalancing, data expiry from namespaces, cluster-wide backup/restore, batch queries, etc. Note that the prioritization scheme ensures that the long running tasks continue to make reasonable progress so that they complete in due course while also ensuring that the response times of the short client transactions is within acceptable parameters (i.e., well under 1 millisecond)

3.3 Scalability

There are two noteworthy aspects of Citrusleaf scaling:

- Linear scaling - the Citrusleaf cluster capacity increases linearly as nodes are added to the cluster with per node throughput staying well over 200,000 TPS at under 1 millisecond response times in real-world configurations.
- Auto scaling - the system requires no operational intervention to add new nodes to the cluster.

3.3.1 Linear Scaling

Citrusleaf attains smooth linear scaling by implementing efficient clustering algorithms, balanced data partitioning and efficient transaction routing from the client to the nodes within the cluster.

3.3.1.1 Efficient Distributed Consensus

All of the nodes in the Citrusleaf system participate in a Paxos [8,9] distributed consensus algorithm, which is used to ensure agreement on a minimal amount of critical shared state. The most critical part of this shared state is the list of nodes that are participating in the cluster. Consequently, every time a node arrives or departs, the consensus algorithm runs to ensure that agreement is reached. This process takes a fraction of a second. After consensus is achieved, each individual node agrees on both the participants and their order within the cluster. Using this information the master node for any transaction can be computed along with the replica nodes. Since the essential information about any transaction can be computed, transactions can be simpler and use proven database algorithms. This results in low latency transactions which only involve a minimal subset of nodes.

3.3.1.2 Balanced Data Partitioning

In a Citrusleaf database cluster, the contents of a namespace are partitioned by key value and the associated data items are spread across every node in the cluster. Once a node has been added to or removed from a cluster, data rebalancing starts immediately after a cluster change is detected. The automatic data rebalancing is conducted across the internal cluster interconnect. Balanced data ensures that query volume is distributed evenly across all nodes, and the cluster stays robust in the event of node failure

happening during rebalancing itself. This provides Citrusleaf's most powerful feature: scalability in both performance and capacity can be achieved entirely horizontally. Furthermore, the system is designed to be continuously available, so data rebalancing doesn't impact cluster behavior.

If a cluster node receives a request for a piece of data that it does not have locally, it satisfies the request by creating an internal proxy for this request, fetching the data from the real owner using the internal cluster interconnect (see Figure 6), and subsequently replying to the client directly.

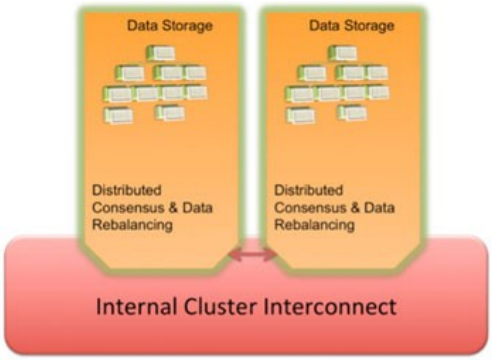

Figure 6: Cluster Interconnect.

3.3.1.3 Efficient transaction routing

Without efficient transaction routing from the client, a request would always require an extra network hop. Either a proxy would be placed in the middle of the transaction, increasing latency and decreasing throughput, or transactions would flow over the cluster interconnect. Therefore, the Citrusleaf client dynamically discovers the cluster's current partition state and routes transactions to the correct node in the cluster. As nodes are added to the cluster and the data is automatically rebalanced, the cluster's capacity to handle client throughput continues to increase linearly. Specifically, there is no added overhead introduced because of cluster interconnect. This fact is borne out in our benchmarks which demonstrate that the maximum number of transactions supported per node stays constant as we add nodes to the cluster.

3.3.2 Auto Scaling

The Citrusleaf database platform is self-organizing and scales elastically to fit your business needs in a "just in time" manner. Distributed consensus algorithms for automatic node addition and removal combined with automatic data rebalancing algorithms provide robust self-management of the system during node arrival and departure.

3.3.2.1 Automatic node addition and removal

The Citrusleaf algorithms for detecting node arrival and departure are robust.

- We use multiple independent paths for nodes to discover each other. Nodes can be discovered via an explicit heartbeat message and/or via other kinds of traffic sent to each other using the internal cluster interconnects.
- The algorithms to discover node departure need to avoid mistaken removal of nodes during temporary congestion. We again use failures along multiple independent paths to ensure high confidence in the event.
- Sometimes nodes can depart and then join again in a relatively short amount of time (router glitches). The system therefore avoids race conditions by uniquely identifying the order of node arrival and departure events using single threaded execution for this function.

The Citrusleaf consensus algorithm for admitting and removing nodes from the cluster is unique in that consistency votes are taken only during cluster reorganization. Once the cluster membership list is agreed upon, the rest of the data routing tables can be independently generated by the cluster members very quickly. Unlike many other clustered solutions, Citrusleaf clusters do not have a “master” during normal operation. All nodes are treated equal and data is distributed equitably among all nodes of the cluster.

3.3.2.2 Automatic Data Rebalancing

Citrusleaf’s data rebalancing mechanism ensures that query volume is distributed evenly across all nodes, and is robust in the event of node failure happening during rebalancing itself. The system is designed to be continuously available, so data rebalancing doesn't impact cluster behavior. The transaction algorithms are integrated with the data distribution system, and there is only one consensus vote to coordinate a cluster change. With only one vote, there is only a short period when the cluster internal redirection mechanisms are used while clients discover the new cluster configuration. Thus, this mechanism optimizes transactional simplicity in a scalable shared-nothing environment while maintaining ACID characteristics.

Citrusleaf allows configuration options to specify how much available operating overhead should be used for administrative tasks like rebalancing data between nodes as compared to running client transactions. In cases where slowing transactions temporarily is preferred, the cluster will heal more quickly. In cases where transactional speed and volume must be maintained, the cluster will rebalance more slowly.

In some cases, the replication factor cannot be satisfied with the remaining cluster resources. The cluster can be configured to either decrease the replication factor and retain all data, or begin evicting the oldest data that is marked as disposable. If the cluster can't accept any more data, it will begin operating in a read-only mode until new capacity becomes available - at which point it will automatically being accepting application writes.

By not requiring operator intervention, the cluster is able to self-heal even at demanding times. In one customer deployment, a rack circuit breaker blew, taking out one node of an 8 node cluster. No operator intervention was required. Even though the outage was at peak time for the data center, transactions continued with full ACID fidelity. In several hours, when the fault was corrected and the troublesome rack was brought back online, operators did not need to take special steps to maintain the Citrusleaf cluster.

4. System Scalability

This section provides the results of tests we ran to validate that the Citrusleaf cluster scales linearly at high performance. Citrusleaf scales linearly, handling **over 200,000 transactions per second per server node** of read-heavy load on commodity hardware, while providing **immediate consistency**.

Here is a description of the tests that we ran (the results are compiled in Table 1). The following apply to all the tests:

- We tested Citrusleaf's ability to scale to up to four server nodes.
- We ran with two, three, and four node Citrusleaf clusters, with 2-way replication featuring immediate consistency,.
- The number of records used was 25M records per server node, with 50 byte keys and a single 8 byte integer values (we also ran tests with larger values and got similar results).
- Citrusleaf was run with the configuration of keeping data in memory while also backing up the data persistently on rotational disk. In this configuration, every update transaction will synchronously write to both memory copies and queue the write to rotational storage, before returning success to the client. In a year of the system running in production with 100% uptime, we have found this scheme to provide robust durability with extremely high transaction throughput.
- Note that we also ran tests with data directly stored in flash based storage (Solid State Disk or SSD) that showed similar scalability characteristics but those results are omitted here due to lack of space. The key difference we noticed in the SSD based tests is that the response times are about four tenth of a millisecond higher and transaction throughputs are commensurately lower compared to the in-memory configuration.
- The tests report average response time values and, for both SSD and non-SSD configurations, we were able to consistently get response times of less than 1 millisecond for over 95% of both reads and writes.
- Note that in none of these tests, we attempted to push the system to its maximum throughput per node. One reason was that we had access to only 8 client nodes and 2 clients per server node was insufficient to push the node to its max throughput.

In all the tests, the Throughput is rounded to the nearest 1,000 and Response Time is rounded to the nearest 0.01 milliseconds.

4.1 Test #1: Throughput, 50/50 R/W

In the first test, we had a number of client nodes read and update simultaneously against these records. These reads and updates simulated a random update-heavy load, and were done in a 50/50 ratio, and did not use batching. The number of client nodes used was scaled linearly with the number of server nodes, at a rate of 2 client nodes per server node. For example, the test with 3 server nodes received input from 6 client nodes. Each client node ran exactly 6 client processes.

4.2 Test #2: Throughput 95/5 R/W

In the second test, we had a number of client nodes read and update simultaneously against the records loaded into the system. These reads and updates simulated a random read-heavy load, were done in a 95/5 ratio, and did not use batching. The number of client nodes used was scaled linearly with the number of server nodes, at a rate of 2 client nodes per server node. For example, the test with 4 server nodes received input from 8 client nodes. Each client node ran exactly 8 client processes.

Table 1: Scalability.

Test	N	Throughput (Server-Side[2])	Resp. Time (Read)	Resp. Time (Update)
#1 Throughput • 50/50 R/W	2	297K	0.26 ms	0.69 ms
	3	404K	0.28 ms	0.77 ms
	4	519K	0.29 ms	0.80 ms
#2 Throughput [6] • 95/5 R/W	2	433K	0.30 ms	0.39 ms
	3	636K	0.30 ms	0.41 ms
	4	839K	0.31 ms	0.42 ms
#3 Min Response Time • 50/50 ratio	2	100K	0.16 ms	0.34 ms
	3	150K	0.16 ms	0.33 ms
	4	200K	0.16 ms	0.34 ms

4.3 Test #3: Min Response Time, 50/50 R/W

In the third test, we had an appropriate number of client nodes, each running four Java clients, read and update simultaneously against these records which kept the response time as low as possible. These clients generated a total throughput of 50,000 transactions per server node of random update-heavy load. As in the other tests, these reads and updates simulated a random load, were done in a 50/50 ratio, and did not use batching. The servers in the two node test received input from 1 client node, the servers in the three node test received input from 2 client nodes, and the servers in the four node test received input from 3 client nodes. The results show that the system response time stays constant across nodes.

4.4 Hardware specification

4.4.1 Server Node

- Intel core i5-2400 Quad-Core @ 3.10GHz
- Asus P7P55 LE BIOS 1101
- 16GB 1600 MHz DDR3 memory
- 1 Gb/sec Realtek RTL8111B ethernet (single port)
- 7200 rpm system disk formatted as ext3
- CentOS 5.5 with 2.6.36.4 kernel
- Citrusleaf Version 2.0.23.11

4.4.2 Client Node

- Intel core i5-750 Lynnfield Quad-Core @ 2.80GHz
- Asus P7P55 LE BIOS 1101
- 16GB 1333 MHz DDR3 memory
- 1 Gb/sec Realtek RTL8111B ethernet (single port)
- 7200 rpm system disk formatted as ext3
- CentOS 5.5 with 2.6.36.4 kernel
- Citrusleaf Version 2.0.23.11
- Java 1.6.0_17

4.4.3 Networking Equipment

- Dlink DGS-1016D

5. Other Systems

As we mentioned earlier, there are a number of systems that have tackled the problem of high performance needed for applications. Here is a review of how some of the key systems differ from Citrusleaf in their approach.

5.1 Today's RDBMS

Today's RDBMs implementations provide a rich and well known set of interfaces for data storage. These systems are the primary means for reliable data storage within computer systems. Within the overall web application environment, RDBMs are either a strong starting point due to their ease of use and well known interfaces (example: MySQL and Postgres), or a more trusted alternative (Oracle, DB/2, MSSQL). In what is now seen as a traditional architecture, three external technologies are required to achieve scale: a key-value cache, read replicas, and write-shards. Some open-source frameworks have evolved to provide scalability wrappers to RDBMs, and some professional RDBMSs vendors sell cache products that have some overlap with Citrusleaf (Oracle's Coherence).

As we outlined earlier, the lack of cost-effectiveness of the enhanced RDBMS solutions for high traffic internet applications (and their virtual inability to scale horizontally without a lot of special hardware and configuration) had given rise to a number of new database technologies, the NoSQL and the high performance SQL alternatives. Citrusleaf is currently part of the NoSQL list (for lack of a better term) and is cost-effective for the most demanding data intensive applications like real-time bidding.

5.2 NoSQL

We briefly discuss a few NoSQL technologies and identify key differences between them and Citrusleaf.

5.2.1 Cassandra

Cassandra [1] is a clustered distributed database that is extremely scalable and in use in some very huge internet deployments today. A key difference between Citrusleaf and Cassandra is that Citrusleaf provides immediate consistency while Cassandra currently supports only eventual consistency.

5.2.2 Mongo DB

Mongo DB [11] has strong support for secondary indexes, map/reduce scan using JavaScript, integration with the newly-popular 'node.js' web programming model. Their use of server-side JavaScript provides an easy-to-use programming environment for those who are used to web programming. The key difference between MongoDB and Citrusleaf is that Mongo DB supports an automatic sharding scheme, Citrusleaf is natively clustered like Cassandra and clustered systems are inherently amenable to easy self-management. While MongoDB is high performance, it is not clear that it is intended for extremely high throughputs at low latency that real-time bidding systems demand.

5.2.3 Redis

Redis [16] is primarily used for real-time analytics today. Redis, like Citrusleaf, can easily be configured to store data on rotational disk for durability. Its strengths are internal support for complex data types, such as lists and counters. It does not support native clustering yet, but is very high performance.

5.2.4 Other NoSQL

Note that addition to the above products that are somewhat closely related to Citrusleaf are a number of other systems that are also present, e.g., CouchBase [4], Riak [17], etc., and also related high-performance SQL based solutions like VoltDB [20], Clusterix [2], Schooner [18], etc. Reviewing all the related products in this area is beyond the scope of this paper.

5.3 Scalable SQL

There have been a few key efforts in making SQL databases scale for real-time applications. Clustrix [2] and VoltDB [20] are two examples.

5.3.1 Clustrix

Clustrix [2] is a clustered SQL database appliance. It claims to support a rich amount of MySQL [12] functionality. The system depends of special network level interconnect like Infiniband unlike a software only solution like Citrusleaf.

5.3.2 VoltDB

VoltDB [20] is an effort to modernize and rewrite the processing of SQL based entirely on in-memory technology. VoltDB supports ACID transactions and traditional database level reliability. The key difference between Citrusleaf and VoltDB is VoltDB's support for a subset of SQL (Citrusleaf does not support SQL) and the lack of support for flash storage (Citrusleaf supports both main-memory and flash based configurations).

6. Future Work

There are two kinds of scenarios in which the new database technology can evolve.

One way is for specialized databases to be built based on the application domain. In this scenario, there will be specialized databases for Graph (e.g., Neo4J [13]), other specialized ones for ultra-high performance transaction processing (e.g., VoltDB, Citrusleaf), and yet another for Geo-location processing, and so on. In this model, there will be a specific DBMS that is well suited for every application area that is not very suitable for those in another area.

Another scenario of database evolution would be one where a single fast, reliable database platform would support multiple application level APIs. The advantage of this approach is that the database platform can be optimized to solve some of the hardest issues like real-time prioritization, clustering, storage management, concurrency, ACID, failure management, etc. Such a platform can then be used to support any and all application APIs. In fact, it is well known that the traditional Database systems that dominate the market place today ([14] [5]) were built on previously existing high performance storage platforms that included support for ACID and had a fast access path to the data. It was only later that SQL became popular and the SQL interface engine was simply built on top of these fast reliable storage platforms.

One way to think of Citrusleaf is that it is a "state of the art" version of an ultra high performance transactional storage platform that runs on commodity hardware. We believe that we should be able to provide access to such a platform from existing interfaces, both SQL and various kinds of NoSQL APIs. No doubt, this would be an extremely challenging endeavor but by no means impossible as demonstrated by the evolution of database products over the last 30 years.

Our position here logically leads to a list of future work in this area. A few possibilities are to add query processing support to handle real-time analytics, adding APIs for graph and geo-location, and even using Citrusleaf as the backend data platform with SQL as a front-end interface. Another dimension of future enhancements would be to support replication of data between geographically distributed clusters analogous to what is supported by PNUTS [3].

7. Conclusion

Citrusleaf is a distributed database platform that marries traditional database consistency and reliability with high performance distributed clustering and sophisticated self-management. Citrusleaf is unique among NoSQL databases for its ability to provide immediate consistency and ACID while still being able to consistently exceed the performance and scalability standards required by demanding real-time applications.

The Citrusleaf system architecture is such that there is no single master in the system and this creates extreme resiliency. The system can rebalance data on the fly while using real-time prioritization techniques to balance short running transaction s with several classes of long running tasks. Our technology is able to achieve high transactional throughput using a variety of techniques on the server as well as by smart routing done by the client.

The system scales linearly at extremely high throughput while keeping response time in the sub-millisecond range as seen by the results of tests presented above. This kind of performance has enabled Citrusleaf to be used by some of the world's largest real-time bidding networks in the area of display advertising.

Finally, it is noteworthy that the high availability and self management techniques outlined in this paper have been validated in the field by internet scale production deployments that have run for almost a year with no downtime whatsoever.

8. ACKNOWLEDGMENTS

Our thanks to IBM Fellow, Don Haderle for many illuminating discussions on how to build a high performance NoSQL database with traditional ACID properties, to Carolyn Wales for documenting the Citrusleaf architecture in a concise manner, to Joey Shurtleff for designing and running the scalability tests.

9. REFERENCES

[1] Cassandra, *http://cassandra.apache.org*

[2] Clustrix, *http://www.clustrix.com*

[3] Cooper, B. F., et al. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB, 1,* 2, 1277-1288, (2008).

[4] CouchBase, *http://www.couchbase.com*

[5] DB2, *http://www.ibm.com/software/data/db2*

[6] DeCandia, G., et al, Dynamo: Amazon's Highly Available Key-value Store, *Proceedings of the Symposium on Operation Systems Principles,* 205-220, (Oct. 2007).

[7] Dewitt, D., Gerber, B., Graefe, G., Heytens, M., Kumar, K., Muralikrishna, GAMMA – A High Performance Dataflow Database Machine. *Proceedings of the 1986 VLDB Conference,* 228-237, (Aug. 1986).

[8] Lamport, L. Paxos Made Simple, Fast, and Byzantine. *OPODIS,* 7-9, (2002).

[9] Lamport, L., The part-time parliament. *ACM Trans. On Computer Syst, 16,* 2, 133-169, (May 1998).

[10] Memcached, *http://memcached.org*

[11] MongoDB, *http://www.mongodb.org*

[12] MySQL, *http://www.mysql.com*

[13] Neo4J, *http://neo4j.org*

[14] Oracle, *http://www.oracle.com/us/products/database/index.html*

[15] Real-time Web, *http://en.wikipedia.org/wiki/Real-time_web*

[16] Redis, *http://redis.io*

[17] Riak, *http://wiki.basho.com*

[18] Schooner, *http://www.schoonerinfotech.com*

[19] Shard (database architecture), *http://en.wikipedia.org/wiki/Shard_(database_architecture)*

[20] VoltDB, *http://voltdb.com*