



White Paper

Citrusleaf 2.0 Architecture Overview

Citrusleaf is a distributed, scalable NoSQL database. When architecting Citrusleaf, we had a couple of key objectives: first, to create a flexible, scalable platform that would meet the needs of today's web-scale applications; and second, to provide the robustness and reliability (ie, ACID) expected from traditional databases. From our experiences developing applications with web-scale databases and our interactions with our customers, we developed a general philosophy of operational efficiency that further guided product development. It is these three principles - NoSQL flexibility, traditional database reliability, and operational efficiency - that drive the Citrusleaf architecture.

System Architecture

The Citrusleaf system architecture can be broken into three main layers - the Client Layer, the Distribution Layer, and the Data Layer. We'll very briefly describe the layers in this section, and in the later sections dive deeper into each layer's functionality.

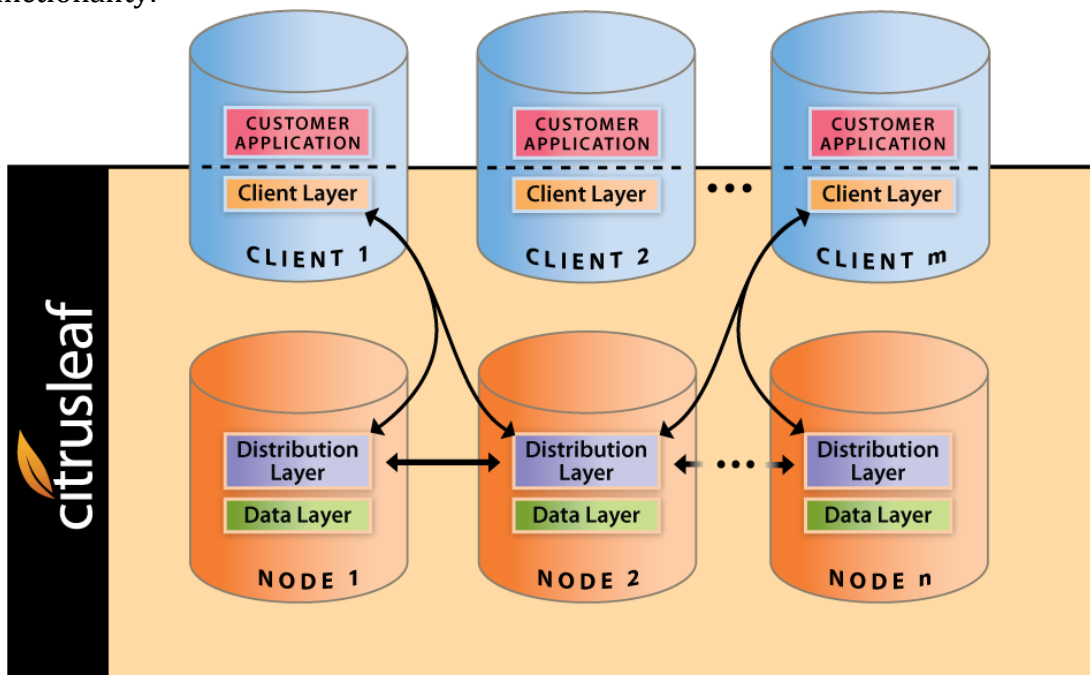


Fig 1: Citrusleaf Architecture Overview



Client Layer

The Citrusleaf Client libraries make up the Client Layer. To make application development easier and more efficient, Citrusleaf provides a 'smart client' – in addition to implementing the APIs exposed to the transaction, the Client also tracks cluster configuration and manages the transaction requests, making any change in cluster membership completely transparent to the Application.

Distribution Layer

In the Distribution Layer, server cluster nodes communicate to ensure data consistency and replication across the cluster. The system uses a shared-nothing architecture, which enables it to be linearly scalable. The Distribution Layer is also at the heart of most of the system's ACID guarantees.

In addition, the Distribution Layer ensures that the Citrusleaf cluster remains fully operational when individual server nodes are removed from or added to the cluster.

Data Layer

On each server node, the Data Layer handles management of stored data on disk and maintains indices corresponding to the data in the node. The data layer is optimized for operational efficiency – indices are stored in a very tight format to reduce memory requirements, and the system can be configured to use low level access to the physical storage media to further improve performance.

Each of these layers is explained in more detail below.

Client Layer

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') that 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.

To now dive into more technical detail, consider that Client Layer has the following responsibilities:

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

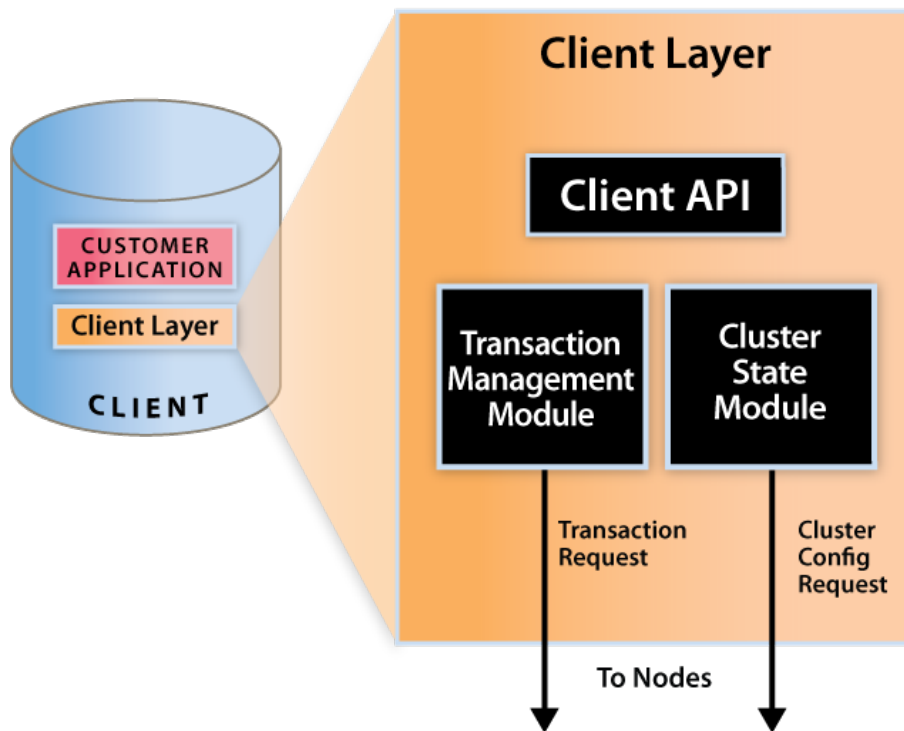


Fig 2 - Client Layer

Each of these responsibilities is discussed in more detail below.

Providing an API interface for the Application

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 backup and restore, and is



implemented in a way that allows the system to remain functional while the backup or restore is happening.

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 that underlies the APIs, see the Data Layer section.

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. 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 if nodes are brought on- or off- line.

Managing transactions between the Application and the Cluster

When a transaction request comes in from the application, the Client formats that request into an optimized wire protocol for transmission to the servers. The Client uses its knowledge of the cluster configuration to determine which server is most likely to contain the requested data and parcels out the request appropriately.

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. This strategy provides an additional level of reliability and resilience when dealing with transient issues.

Distribution Layer

The Distribution Layer 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' – there are no separate



'managers', eliminating bottlenecks and single points of failure such as those created by master/slave relationships.

All communication between both cluster nodes and Application machines 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 GigE 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.

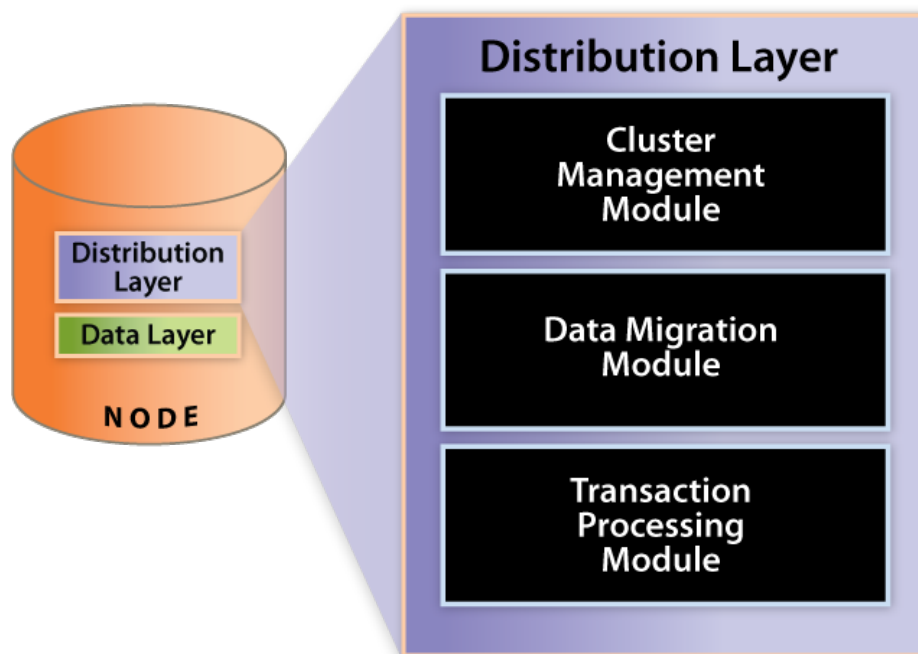


Fig 3 – Distribution Layer

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 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) or unicast (requiring slightly more configuration).



To increase reliability in the face of heavy loading - 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 divide the primary index space into data 'slices' and then to assign read and write masters and replicas to each of the slices. Because the division is purely algorithmic, the system scales without a master and eliminates the need for additional configuration that is required in a sharded environment. After cluster reconfiguration, data migration between the slices is handled by the Data Migration Module, below.

Data Migration Module

When a node is added or removed from the cluster, the Data Migration Module is invoked to transition the data to and from nodes as appropriate for the new configuration. The Data Migration Module is responsible for balancing the distribution of data across the cluster nodes, and for ensuring that each piece of data is duplicated across nodes as specified by the system's configured Replication Factor. The data migration process, by which data is moved to the appropriate node, is completely transparent to the both the Client and the Application.

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 prioritization ensures that the system is functional even while servers are coming on- or off- line, and that changes to the configuration of the cluster do not bring the system down.

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 correct node responsible for processing the transaction - a read replica in the case of a read request, or in the case of a write request, the write master for that data's slice. (Requests that involve simultaneous reads and writes go to the write master.) In this situation, the Transaction Processing Module looks up the data, applies the appropriate operation and returns the result to the Client. If the request modifies data, the Transaction Processing Module also ensures immediate consistency within the system - before committing the data internally and returning the result to the client, it first propagates the changes to all other replicas.



In some cases, the node that receives the transaction request will not contain the data needed to complete the transaction. This is a rare case, since the Client maintains enough knowledge of cluster state to know which nodes should have what data. However, if the cluster configuration changes, the Client's information may briefly be out of date. In this situation, the Transaction Processing Module forwards the request to the Transaction Processing Module of the node that is responsible for the data. This transfer of control, like data migration itself, is completely transparent to the Client.

Finally, the Transaction Processing Module is responsible for resolving conflicts that can occur when a cluster is recovering from being partitioned. In this case at some point in the past, the cluster has split (partitioned) into two (or more) separate pieces. While the cluster was partitioned, the Application wrote two different values to what should be the identical copies of the data. When the cluster comes back together, there is a conflict over what value that data should have.

The Transaction Processing Module is responsible for detecting this condition, and can resolve it in one of two ways. If the system has been configured for automatic conflict resolution, the Transaction Processing Module will consider the data with the latest timestamp to be canonical. On a read operation, that data will be returned to the Client. If the system has not been configured for automatic conflict resolution, the Transaction Processing Module will hand both copies of the data to the Client (where they are then forwarded to the Application). It becomes the Application's responsibility to decide which copy is correct.

Data Layer

The Data Layer holds the indexes and data stored in each node, and handles interactions with the physical storage. It also contains modules that automatically remove old data from the database, and defragment the physical storage to optimize disk usage.

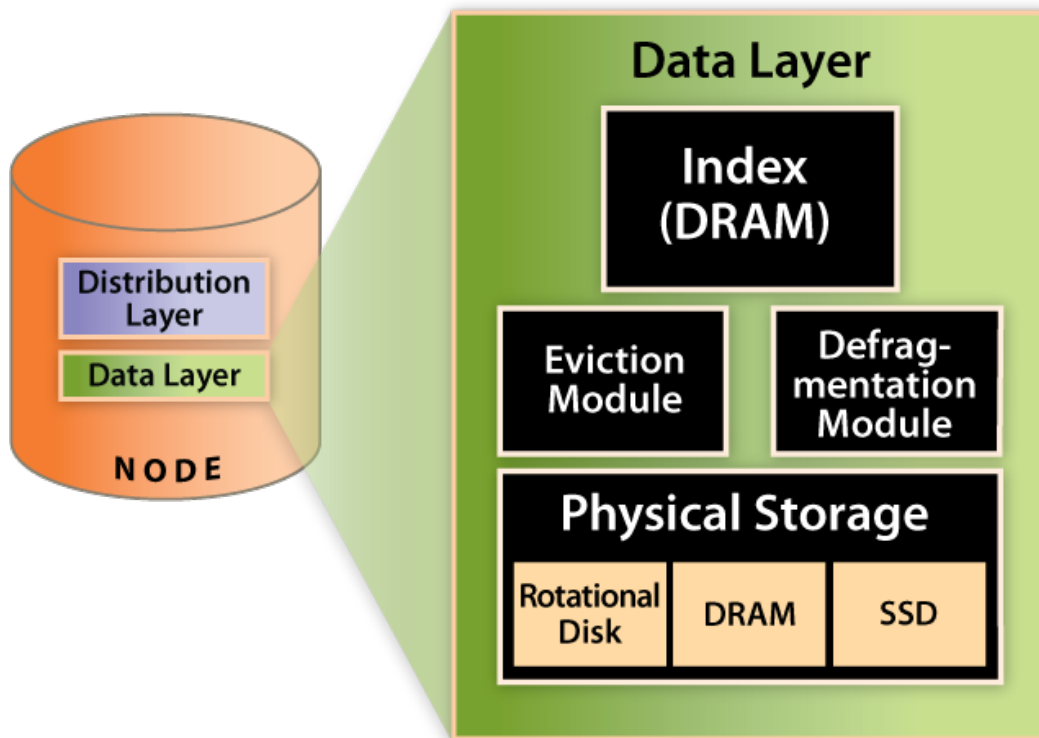


Figure 4 - Data Layer

Before discussing these components, let's first take a look at the Citrusleaf Data Model.

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 RDBMS 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 retention and reliability requirements for a given set of data. For example, one of the most important system configuration policies is the Replication Factor, which controls the number of stored



copies of every piece of data. Changing the Replication Factor allows you to trade increased storage requirements for improved resiliency to simultaneous hardware failures.

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.

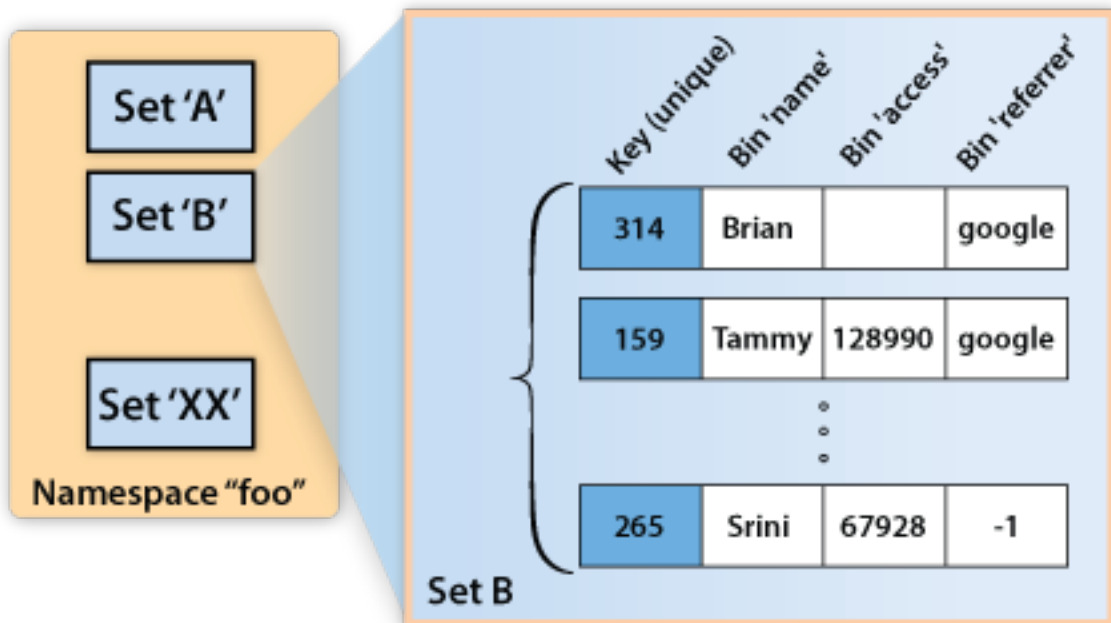


Fig 5 – Citrusleaf Data Model

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 if additional flexibility is needed.

Each record is also tagged with a Generation Count, which is the number of times that the record has been modified. This number is used to reconcile data if the cluster breaks apart (partitions) and rejoins, and can also be used by the clients in CAS operations to ensure that they are modifying a known copy of the data.



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 as 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 critical systems.

(Because storing indices in DRAM impacts the amount of DRAM needed in each node, the size of an individual index has been painstakingly minimized. At present, Citrusleaf can store indices for 100 million records in 7 gigabytes of DRAM.)

The contents of each namespace are spread evenly across every node in the cluster, and duplicated as specified by the namespace's configured Replication Factor. For optimal efficiency, the location of any piece of data in the system can be determined algorithmically, without the need for a stored lookup table.

Defragmentor 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.

Conclusion

The Citrusleaf architecture is derived from three core principles – NoSQL scalability and flexibility, and traditional database consistency and reliability. 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. Citrusleaf represents a best of all worlds solution for applications requiring fast transactional access to web-scale data.