VoltDB the NewSQL database for high velocity applications

VoltCache: Developing Key/Value Applications Using VoltDB

A memcached-compatible Key/Value store built on the VoltDB RDBMS

VoltDB, Inc.
www.voltdb.com
Twitter: @voltdb

About This Paper

Memcached is a popular Key/Value caching solution for high scaling web applications, known for its performance and programming ease. However, as applications become more complex, memcached's limitations create challenges for developers and operations teams.

This paper discusses VoltCache, an alternative to memcached. Built on the blazing-fast VoltDB RDBMS, VoltCache overcomes many of memcached's challenges without sacrificing performance or simplicity of programming.

Read this paper to learn about:

- Database throughput reaching millions of operations per second
- Why VoltDB's scale-out architecture is a perfect match for Key/Value caching
- How VoltCache can simplify programming and overcome common K/V challenges
- How VoltCache performs against memcached/MySQL and Membase
- What's inside the VoltCache Key/Value function library

Table of Contents

Getting the Best Performance From High-Scaling Web Apps 2

The Basics of Key/Value Caching 2

The Data Management Pitfalls of Key/Value Caching 3

VoltCache: Database, Meet Cache 4

An Ultra-Fast, Integrated Database and Cache Solution 4

Performance Comparisons 6

Single-Node Benchmarks 6

Multi-Node Benchmarks 7

Performance Analysis: Membase vs. VoltCache 8

Summary 9

Getting Started with VoltDB and VoltCache 10

VoltDB Demo Dashboard 10

Appendix A – VoltCache API 11

Getting the Best Performance From High-Scaling Web Apps

If you are building an application for the web, you are building for scale. Today's web apps have the opportunity to achieve global reach, offering a high degree of interactivity to millions of users.

As a result, speed matters – not only in terms of application performance, but also in your ability to deliver working systems to hungry users quickly. So to get out of the gate fast, many Web 2.0 developers begin building their application with a relational database management system (RDBMS) like MySQL. This data storage model is very popular, comes built into many modern coding frameworks, and has been proven across many applications.

However, once your app begins to experience heavy use, RDBMS performance often hits a wall. Successful web applications reach a demand profile where normal traffic stresses the data tier, and spikes simply cause things to stop working entirely.

A popular solution to this problem is to augment the RDBMS with a Key/Value (K/V) cache such as memcached. This optimization offloads most read operations from the database. Thus, reads are faster because they're serviced primarily from the cache, and writes (transactions) are faster because they're not contending with reads.

The Basics of Key/Value Caching

A traditional relational database is designed to protect the integrity of the data it holds, delivering it consistently and accurately to any client that requests it. It ensures that the relationships inherent in the data are preserved, even throughout the most complex, multi-step transactions. The database developer codes the rules of integrity through schema, transactional programming, and other RDBMS features.

By contrast, a Key/Value cache is a lightweight, easy-to-implement store that allows programs to store and access arbitrary structures, or blobs, of data, simply by providing a lookup key. K/V systems are often used to optimize performance. An application retrieves data from its database, stores it in the cache, and then the next time it needs that data it can get it right from the cache, avoiding the database altogether.

> The most popular Key/Value cache is memcached, an open source main memory K/V store.

K/V concepts are easily understood by software engineers trained in object-oriented programming. Common uses for K/V caches include:

- Dynamically-generated web pages
- User-based session data
- Images and videos to be served to web and mobile clients
- Top viewed or most recent posts, articles, news stories, and similar content

There are a number of proven design patterns for working with a K/V cache. (However, some complexities are hidden beneath the surface and don't become evident until a production database begins to scale).

A typical K/V cache implementation scenario is to pair memcached with a RDBMS like MySQL. The interplay between application, cache and database is roughly as follows:

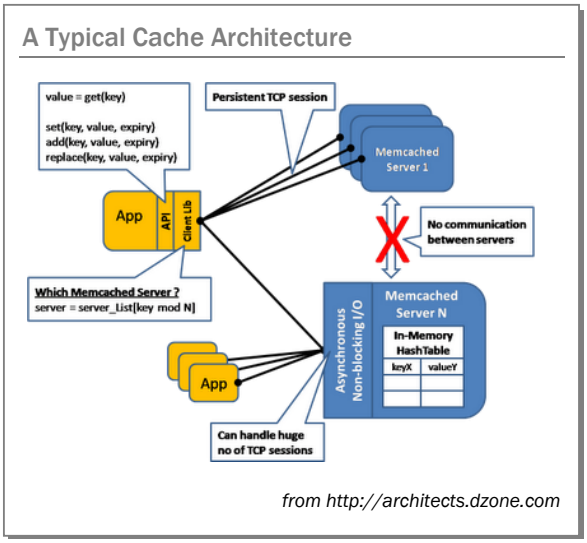
- Before the app performs a database query, it looks for the data in the cache.
- If it doesn't find the data there, it retrieves the data from the database, delivers it to the user and writes it to the cache.
- The next time that query is executed, the application will find the data in the cache and avoid the database read.
- When the application updates a database record, it must also invalidate the associated cache record, effectively removing the stale data from the cache.
- After the cache has been invalidated, the next time the application queries for that data, it will start the cycle all over again.

These mechanics, while fairly well understood, mean that a significant amount of data "housekeeping" code exists in the application tier to manage cache locations and invalidation logic. This increases the complexity – and brittleness – of the overall application, especially as its features and functions grow over time. It also does nothing to improve the raw write throughput of the underlying database.

The Data Management Pitfalls of Key/Value Caching

Although K/V caching can be an effective scaling solution, the benefits begin to break down as a database grows in size, complexity and workload. Common challenges include:

- **Applications must do their own data partitioning.** A typical memcached implementation requires the client application code to manage the cache partitioning scheme. Although the access patterns are pretty well understood, this code nonetheless represents additional code complexity, failure points and operational process.



- **Complexity increases with scale.** Sharding and managing a database (e.g., MySQL) that's front-ended by a caching tier (e.g., memcached) can be a house of horrors. Imagine a manually-sharded MySQL database with multiple memcached servers/shard. Not only does the application have to explicitly perform operations on this storage cocktail, operations teams must somehow work out how to monitor the infrastructure and manage its loosely-connected components. As the database grows, the complexity of managing this infrastructure is compounded.

- **De-normalization is burdensome.** As application usage grows, it's common for users to demand more flexible access to their data. To the degree that user queries are serviced by data stored in de-normalized structures (i.e., a BLOB in a K/V datastore), an application must decompose that structure, maintain its state and possibly even perform complex operations on the data.

VoltCache: Database, Meet Cache

The inspiration for VoltCache was user demand. Some developers simply prefer a K/V data model but need many of the features of a robust RDBMS, including ACID transactions. So we implemented VoltCache in response to our customers' requests for a caching system based on VoltDB. Essentially, *VoltCache is a K/V data model and stored procedures, implemented on VoltDB, and accessed via a memcached-compatible function library*.

We benchmarked the performance of VoltCache against Membase and memcached/MySQL. Our test results, which are presented later in this paper, reveal that VoltCache generally outperforms these technologies at scale. Like Membase, VoltCache delivers the throughput and simplicity of an in-memory database; like memcached/MySQL, VoltCache delivers the power of a full ACID compliant RDBMS behind the K/V cache, easily supporting high velocity mixed-workload applications.

An Ultra-Fast, Integrated Database and Cache Solution

VoltCache is based on the VoltDB database, a breakthrough RDBMS specifically designed to run on modern scale-out infrastructures – fast, inexpensive servers connected via high-speed data networks.

At its core, VoltDB delivers all the features you expect from a clustered RDBMS, including SQL support, ACID-level consistency, developer APIs, and data integration and expert interfaces.

However, VoltDB differs from traditional RDBMSs in some very important ways:

Ultra-fast, in-memory database that performs magnitudes faster than traditional RDBMS products on a per-node basis.

Shared-nothing scale-out architecture with transparent data partitioning for horizontal, massively parallel scalability and simplified client code.

Built-in fault tolerance and durability that use replication, single-threaded in-partition execution, and native transaction processing.

Designed to handle high-velocity data, playing nicely with Hadoop and OLAP data warehouse products.

Under the hood, the key to VoltDB's architecture is its natively partitioned data model. VoltDB uses multiple execution engines that process small work tasks in different data partitions. Unlike a traditional database operating a single execution engine, VoltDB allows you to easily increase the throughput of the database simply by increasing the number of execution engines. This model delivers massive scalability at a fraction of the cost, because execution engines can execute on commodity servers instead of expensive monoliths.

VoltCache: A Memcached Interface on VoltDB

VoltCache was implemented in two stages. First, in response to numerous user inquiries, we modeled a simple K/V store on VoltDB and created stored procedures that support standard PUT/GET/DELETE operations. The result is a fast, scalable K/V store that inherently leverages all of VoltDB's partitioning, fault tolerance and durability features. It's also easy to extend VoltKV – for example, you can easily add columns to a VoltKV application for custom data sub-filtering, to further improve performance.

For common K/V workloads, VoltKV to easily out-paces Cassandra and other K/V stores. More benchmarking information is available here on the Voltage blog.

Second, with basic K/V support in place, we implemented an API that's functionally equivalent to memcached (hence, the name VoltCache). This function call interface supports all common K/V operations and adds constructs for counting, expiration and update. And, again, because it is built on the VoltDB relational engine, you can easily expand VoltCache to support richer functionality, workloads and data objects.

Building client applications in VoltCache is semantically identical to building memcached applications. A summary of the VoltCache API is provided in Appendix A. Developers who have built systems using the combination of memcached/MySQL will find VoltCache to be easier to use (no more need for complex cache synchronization), easier to operate (no need to pre-warm your cache or hassle with multi-step back-ups), and much faster at runtime. Most importantly, VoltCache carries with it all the benefits of the VoltDB RDBMS, including:

- **Data partitioning built into the engine.** You no longer need to handle partitioning in the client application code. VoltCache leverages VoltDB's embedded partitioning capabilities, meaning your code is less complex, and scaling is transparent to your application.
- **Data consistency.** VoltDB is a relational database that delivers ACID level transaction consistency. VoltCache combines a popular K/V programming model with high throughput ACID writes. With VoltCache, K/V app developers no longer have to sacrifice performance and consistency.
- **Disk-based durability.** While most K/V solutions allow you to save transactions to disk, they do so asynchronously, which means there is a window during which a failure could result in the loss of data. VoltDB, and VoltCache by extension, allow you to persist data either synchronously or asynchronously. You can easily optimize latency and durability to suit the needs of your application.

Performance Comparisons

We benchmarked VoltCache against memcached/MySQL in a single-node configuration, and Membase in both single-node and clustered configurations. We chose memcached/MySQL because it's a popular combination for gaining optimizing the performance of Web application databases. We chose Membase because it's a NoSQL product based on memcached and offers integrated features not found in the memcached/MySQL combination.

During our testing, we found that Membase could run the benchmark comparisons with better throughput and lower average latency when we used the Xmemcached Java client instead of the Spymemcached Java client Membase recommends. Thus, our performance graphs show results for Membase's recommended configuration (using Spymemcached) and "Membase (Optimized)", for which we used Xmemcached.

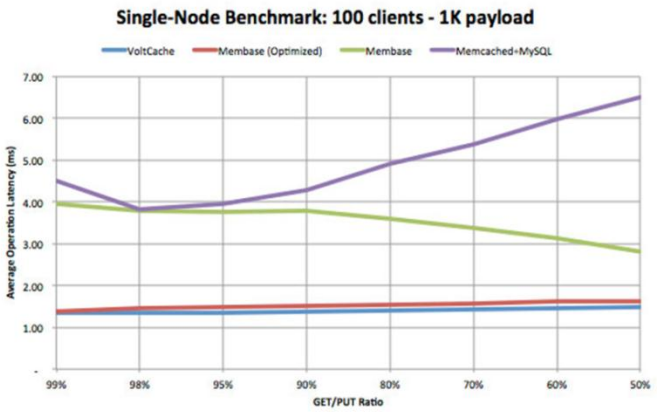
Single-Node Benchmarks

Our first test suite was designed to reveal how each system would perform on a single-server deployment. In this test, the client and server components were all run on a single machine. The client executed PUT and GET operations against the server in varying ratios, for ten minutes at a time. We measured overall throughput and average latency for each product configuration.

Benchmark System Specifications	Benchmark Test
• Dell Studio XPS 9100 Workstation • Single-Quad-core Intel i7 930 @ 2.80GHz • 4 Effective Cores, 8 Virtual cores • RAM: 12GB @ 1333MHz • HD: Standard 7,200RPM SATA Drives (No RAID) • OS: Ubuntu 11.04 x64 (Kernel version: 2.6.38-11-generic) • Java: OpenJDK 6 fully patched	• Everything local (server & client) • 10 minute runs @ different GET/PUT ratios and thread counts • 100 thread-count was "fairest" comparison

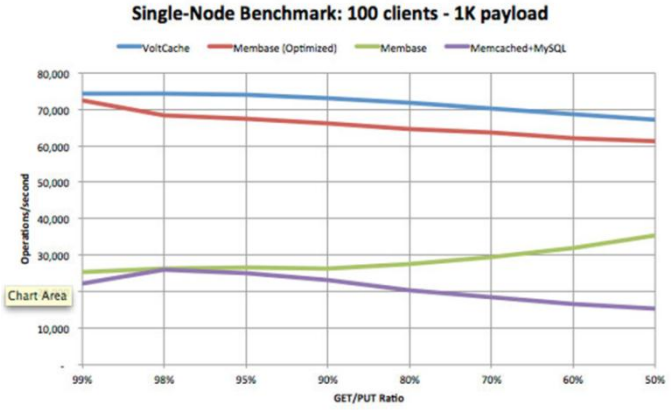
Latency (Lower is Better)

The latency of VoltCache compares very favorably with memcached/MySQL, particularly as write workloads increase. Average latencies of VoltCache and "optimized" Membase were virtually identical at all workload combinations.



Throughput (Higher is Better)

VoltCache leveraged VoltDB's in-memory transaction engine to sustain high write throughput. "Optimized" Membase sustained a similar throughput profile. It's noteworthy that VoltCache achieved these single-node throughput numbers while performing ACID transactions.



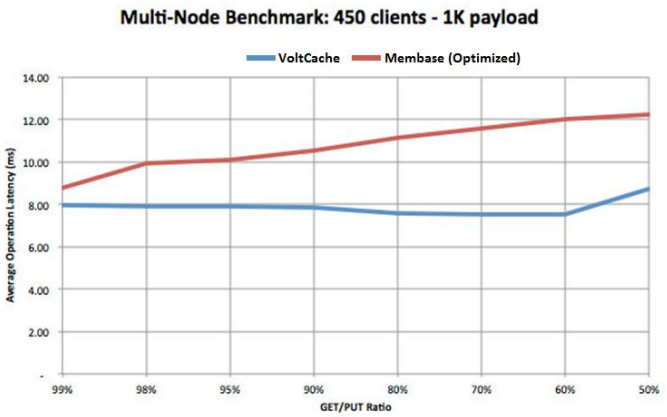
Multi-Node Benchmarks

Our second test suite was designed to compare the performance of small (3-node) Membase and VoltCache clusters. Note that we could not include memcached/MySQL in these tests because MySQL is a single-node database. Our multi-node test included three separate client systems running against three server nodes. Note that since Membase only provides asynchronous persistence, all tests were run in that mode. However, VoltDB also provides a synchronous durability option.

Benchmark System Specifications	Benchmark Test
• Dell PowerEdge R610 • Dual-Quad-core Intel Xeon X5550 @ 2.67GHz (Hyper-threaded; First Launched: Q1 2009) • 8 Effective Cores, 16 Virtual cores • RAM: 48GB @ 1333MHz • HD: Standard 7,200RPM SATA Drives (No RAID) • OS: CentOS 5 x64 (Kernel version: 2.6.18-194.3.1.el5) • Java: OpenJDK 6 fully patched	• 3 server nodes, 3 client nodes • 1 replica, and asynchronous persistence • 10 minute runs @ different GET/PUT ratios and thread counts

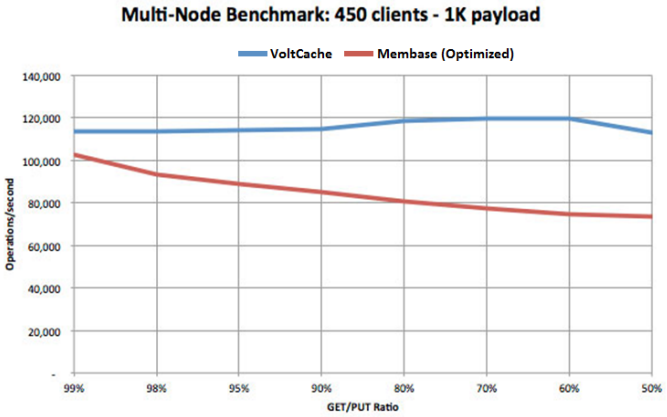
Latency (Lower is Better)

In terms of latency, Membase and VoltCache performed comparably. In these tests, clients were forced to hit the Membase cache to avoid data inconsistencies brought on by Membase's asynchronous replication.



Throughput (Higher is Better)

VoltCache's throughput measurements revealed a similar story. VoltCache achieved better performance because clients can reliably connect to multiple nodes without encountering inconsistent data across nodes.



Performance Analysis: Membase vs. VoltCache

Membase	VoltCache
Very good performance with low client count	Ability to connect over 1,500 clients from a single machine
Protocol sequencing ties requests to individual threads and limits throughput	No protocol sequencing limitation, resulting in better throughput
Must use Google's Xmemcached client (out-of-the-box Spymemcached client yields sub-optimal performance)	No workload duplication to synchronize nodes (memcached)
Multi-threaded client fails with > 150 threads	Performance almost constant with GET/PUT ratio
Performance degrades with increasing client count – both throughput and most noticeably latency	Performance improves with increasing client count
Significant throughput degradation in multi-node clusters with increasing PUT ratio	Performance maintains scalability profile on cluster deployment
Significant latency degradation in multi-node clusters with increasing client count	Lower performance with low client count (< 25)

Summary

Memcached is a popular memory caching technology. It allows developers to offload repetitive read operations from an underlying database (typically MySQL), thereby improving application scale. Unfortunately, a basic memcached infrastructure also requires the application to implement a lot of complex "housekeeping" between the cache and the underlying database, especially as the application scales. Further, memcached does nothing to improve the throughput and latency of the back-end database (MySQL) itself.

VoltDB is a main-memory RDBMS. It is designed specifically to handle high velocity read and write workloads. Where disk-centric RDBMSs struggle to support tens of thousands of ACID transactions per second, VoltDB can easily support millions per second.

At the request of several VoltDB users, we developed a reference implementation (called VoltCache) of memcached using the VoltDB RDBMS. This simple application models a simple Key/Value store on VoltDB, and provides an API that's functionally equivalent to memcached.

We ran a few performance tests using VoltCache, memcached/MySQL and Membase (an open source main memory alternative to memcached). Our tests revealed that VoltCache easily outperformed memcached/MySQL in all scenarios, and performed comparably to an optimized version of Membase in all scenarios.

VoltCache is freely available and prepackaged (including source code) with all distributions of VoltDB. It provides a great starting point for developers who prefer K/V data models. And because VoltCache is implemented on VoltDB, it also delivers the transactional consistency, HA and durability needed for high scaling production applications.

Getting Started with VoltDB and VoltCache

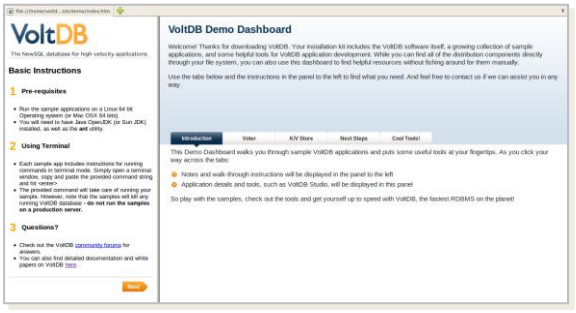
VoltDB is available in two distributions – Community Edition and Enterprise Edition. *The VoltCache reference application is prepackaged with all VoltDB distros*. Community Edition is open source (GPL3) and geared toward developers who are building, testing and tuning VoltDB applications. Enterprise Edition is provided under VoltDB's commercial license and is intended for organizations that are deploying VoltDB applications into production. VoltDB runs natively on supported 64-bit Linux systems. Developers can also build VoltDB applications on 64-bit Mac OSX. VoltDB's main software downloads page, including client libraries and documentation, can be found here. Pre-packaged evaluation versions of VoltDB are available for Amazon EC2 and VMWare:

VoltDB can be setup to run on a cluster of choice on Amazon EC2 within just a few minutes. VoltDB for Amazon EC2 includes the VoltDB Demo Dashboard, VoltDB Studio, sample applications and other developer resources. Get it here.

The VoltDB VMware image can be executed using a variety of VMware players. Although VoltDB is a 64-bit Linux application, VMware allows it to execute as a virtualized image on other platforms such as Windows. Get it here.

VoltDB Demo Dashboard

The VoltDB Demo Dashboard is bundled with all VoltDB distributions. It's a useful tool for product evaluators and developers who are new to VoltDB, providing convenient access to a variety of introductory resources, including sample apps, reference implementations and VoltDB Studio from a simple browser based interface. Source code for most resources in the Demo Dashboard is readily accessible in the VoltDB installation directory.



To learn more about VoltDB, visit www.voltdb.com. The links below also provide quick access to the most popular technical content on our website. To discuss your high performance database needs with a VoltDB engineer, please call Sales at +1.978.528.4660 or send us an email.

Appendix A – VoltCache API

Declarations

```
import net.rubyeye.xmemcached.*;
import net.rubyeye.xmemcached.utils.AddrUtil;
public class XMembaseStore implements IKeyValueStore
{
    private MemcachedClient Cache;
    public XMembaseStore(String servers, int port) throws Exception {
        final String addr = servers.replace(",", ":" + port + " ") + ":" + port + " ";
        this.Cache = new XMemcachedClient(AddrUtil.getAddresses(addr.trim()));
    }

    public void put(String key, byte[] value) throws Exception {
      this.Cache.set(key, 0, value);
    }
    public byte[] get(String key) throws Exception {
        return this.Cache.get(key);
    }
    public void clean() throws Exception {
        this.Cache.flushAll();
    }
    public void close() {
        try {this.Cache.shutdown();} catch(Exception x) {}
    }
}
```

Synchronous API

```
VoltCacheResult add(String key, int flags, int exptime, byte[] data, boolean noreply)

VoltCacheResult append(String key, byte[] data, boolean noreply)

VoltCacheResult cas(String key, int flags, int exptime, byte[] data, long casVersion, boolean
noreply)

VoltCacheResult delete(String key, int exptime, boolean noreply)

VoltCacheResult flushAll(int exptime, boolean noreply)

VoltCacheResult get(String key)

VoltCacheResult get(String[] keys)

VoltCacheResult incrDecr(String key, long by, boolean increment, boolean noreply)

VoltCacheResult prepend(String key, byte[] data, boolean noreply)

VoltCacheResult replace(String key, int flags, int exptime, byte[] data, boolean noreply)

VoltCacheResult set(String key, int flags, int exptime, byte[] data, boolean noreply)

PerfCounterMap getStatistics()
```

Asynchronous API

```
Future<VoltCacheResult> asyncAdd(String key, int flags, int exptime, byte[] data)

Future<VoltCacheResult> asyncAppend(String key, byte[] data)

Future<VoltCacheResult> asyncCas(String key, int flags, int exptime, byte[] data, long casVersion)

Future<VoltCacheResult> asyncDelete(String key, int exptime)

Future<VoltCacheResult> asyncFlushAll(int exptime)

Future<VoltCacheResult> asyncGet(String key)

Future<VoltCacheResult> asyncGet(String[] keys)

Future<VoltCacheResult> asyncIncrDecr(String key, long by, boolean increment)

Future<VoltCacheResult> asyncPrepend(String key, byte[] data)

Future<VoltCacheResult> asyncReplace(String key, int flags, int exptime, byte[] data)

Future<VoltCacheResult> asyncSet(String key, int flags, int exptime, byte[] data)
```