An Oracle White Paper
November 2012

Oracle NoSQL Database

ORACLE

Table of Contents

Introduction

NoSQL databases represent a development in enterprise application architecture, continuing the evolution of the past twenty years. In the 1990's, vertically integrated applications gave way to client-server architectures, and more recently, client-server architectures gave way to three-tier web application architectures. In parallel, the demands of web-scale data analysis added map-reduce processing into the mix and data architects started eschewing transactional consistency in exchange for incremental scalability and large-scale distribution. The NoSQL movement emerged out of this second ecosystem.

NoSQL is often characterized by what it's not – depending on whom you ask, it's either *not only* a SQL-based relational database management system or it's simply *not* a SQL-based RDBMS. While those definitions explain what NoSQL is not, they do little to explain what NoSQL is. Consider the fundamentals that have guided data management for the past forty years. RDBMS systems and large-scale data management have been characterized by the transactional ACID properties of Atomicity, Consistency, Isolation, and Durability. In contrast, NoSQL is sometimes characterized by the BASE acronym:

Basically **A**vailable: Use replication to reduce the likelihood of data unavailability and use *sharding*, or partitioning the data among many different storage servers, to make any remaining failures partial. The result is a system that is always available, even if subsets of the data become unavailable for short periods of time.

Soft state: While ACID systems assume that data consistency is a hard requirement, NoSQL systems allow data to be inconsistent and relegate designing around such inconsistencies to application developers.

Eventually consistent: Although applications must deal with instantaneous consistency, NoSQL systems ensure that at some future point in time the data assumes a consistent state. In contrast to ACID systems that enforce consistency at transaction commit, NoSQL guarantees consistency only at some undefined future time.

NoSQL emerged as companies, such as Amazon, Google, LinkedIn and Twitter dealt with unprecedented data and operation volumes under tight latency constraints. Analyzing high-volume, real time data, such as web-site click streams, provides significant business advantage by harnessing unstructured and semi-structured data sources to create more business value. Traditional relational databases were not up to the task, so enterprises built upon a decade of research on Distributed Hash Tables (DHTs) and either conventional relational database systems or embedded key/value stores, such as Oracle's Berkeley DB, to develop highly available, distributed key-value stores.

Although some of the early NoSQL solutions built their systems atop existing relational database engines, they quickly realized that such systems were designed for SQL-based access patterns and latency demands that are quite different from those of NoSQL systems, so these same organizations began to develop brand new storage layers. In contrast, Oracle's Berkeley DB product line was the original key/value store; Oracle Berkeley DB Java Edition has been in commercial use for over eight years. By using Oracle Berkeley DB Java Edition as the underlying storage engine beneath a NoSQL system, Oracle brings enterprise robustness, stability, and High Availability to the NoSQL landscape.

Furthermore, until recently, integrating NoSQL solutions with an enterprise application architecture required manual integration and custom development; Oracle's NoSQL Database provides all the desirable features of NoSQL solutions necessary for seamless integration into an enterprise application architecture.

Figure 1 shows how Oracle's NoSQL Database fits into a canonical *acquire-organize-analyze* data-cycle ecosystem. Oracle-provided adapters allow the Oracle NoSQL Database to integrate with a Hadoop MapReduce framework or with the Oracle Database in-database MapReduce, Data Mining, R-based analytics, or whatever business needs demand.

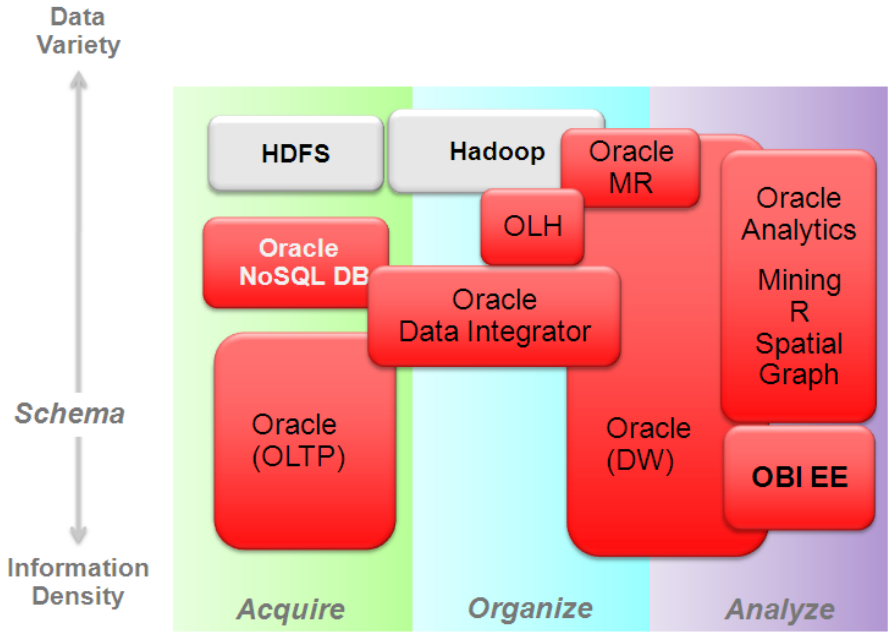


Figure 1: Oracle NoSQL Database integrates seamlessly into the data management ecosystem.

The Oracle NoSQL Database, with its "No Single Point of Failure" architecture, is the right solution when data access is "simple" in nature and application demands exceed the volume or latency capability of traditional data management solutions. For example, click-stream data from high volume web sites, high-throughput event processing and social networking communications all represent application domains that produce extraordinary volumes of simple keyed data. Monitoring online retail behavior, accessing customer profiles, pulling up appropriate customer ads and storing and forwarding real-time communication are examples of domains requiring the ultimate in low-latency access. Highly distributed applications such as real-time sensor aggregation and scalable authentication also represent domains well-suited to Oracle NoSQL Database.

Technical Overview

Oracle NoSQL Database leverages the Oracle Berkeley DB Java Edition High Availability storage engine to provide distributed, highly-available key/value storage for large-volume, latency-sensitive applications or web services. It also provides fast, reliable, distributed storage to applications that need to integrate with ETL processing.

Data Model

In its simplest form, Oracle NoSQL Database implements a map from user-defined keys (Strings) to opaque data items. Although it records internal version numbers for key/value pairs, it only maintains the single latest version in the store. Applications never need to worry about reconciling incompatible versions because Oracle NoSQL Database uses single-master replication; the master node always has the most up-to-date value for a given key, while read-only replicas might have slightly older versions. Applications can use version numbers to ensure consistency for *Read-Modify-Write (RMW)* operations. Avro support is also available and is the preferred option for providing schemas in the value portion of the records.

Oracle NoSQL Database hashes record keys to provide good distribution over a collection of computers that provide storage for the database. However, applications can take advantage of *subkey* capabilities to achieve data locality. A key is the concatenation of a *Major Key Path* and a *Minor Key Path*, both of which are specified by the application. All records sharing a Major Key Path are co-located to achieve data-locality. Within a co-located collection of Major Key Paths, the full key, comprised of both the Major and Minor Key Paths, provides fast, indexed lookups. For example, an application storing user profiles might use the user-name as a Major Key Path and then have several Minor Key Paths for different components of that profile such as email address, name, phone number, etc. Because applications have complete control over the composition and interpretation of keys, different Major Key Paths can have entirely different Minor Key Path structures. Continuing our previous example, one might store user profiles and application profiles in the same store and maintain different Minor Key Paths for each. Prefix key compression makes storage of key groups efficient.

While many NoSQL databases state that they provide eventual consistency, Oracle NoSQL Database provides several different consistency policies. At one end of the spectrum, an application can specify *absolute* consistency which guarantees that all reads return the most recently written value for a designated key. At the other end of the spectrum, an application capable of tolerating inconsistent data can specify weak consistency, allowing the database to efficiently return a value even if it is not entirely up to date. In between these two extremes, an application can specify *time-based consistency* to constrain how old a record might be or *version-based consistency* to support both atomicity for *Read-Modify-Write* operations and reads that are at least as recent as the specified version.

Figure 2: Consistency Policies

Figure 2 shows how the range of flexible consistency policies enables developers to easily create business solutions providing data guarantees while meeting application latency and scalability requirements.

Oracle NoSQL Database also provides a range of durability policies that specify what guarantees the system makes in the event of a crash.

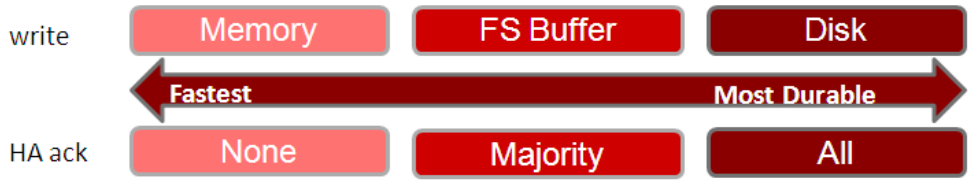


Figure 3: Durability Policies

At one extreme, applications can request that write requests block until the record has been written to stable storage on all copies. This has obvious performance and availability implications, but ensures that if the application successfully writes data, that data will persist and can be recovered even if all the copies become temporarily unavailable due to multiple simultaneous failures. At the other extreme, applications can request that write operations return as soon as the system has recorded the existence of the write, even if the data is not persistent anywhere. Such a policy provides the best write performance, but provides no durability guarantees. The default for the system is that an operation is considered committed when a majority of replicas have acknowledged the write, but not necessarily made it durable. By specifying when the database writes records to disk and what fraction of the copies of the record must be persistent (none, all, or a simple majority), applications can enforce a wide range of durability policies.

API

Incorporating Oracle NoSQL Database into applications is straightforward. APIs for basic Create, Read, Update and Delete (CRUD) operations and a collection of iterators are packaged in a single jar file. Applications can use the APIs from one or more client processes that access a stand-alone Oracle NoSQL Database server process, alleviating the need to set up multi-system configurations for initial development and testing.

Create, Remove, Update, and Delete

Data create and update operations are provided by several *put* methods. The *putIfAbsent* method implements creation while the *putIfPresent* method implements update. The *put* method does both by adding a new key/value pair if the key is not currently present in the database or updating the value if the key does exist. Updating a key/value pair generates a new version of the pair, so the API also includes a conditional *putIfVersion* method that allows applications to implement consistent Read-Modify-Write semantics.

The *delete* and *deleteIfVersion* methods unconditionally and conditionally remove key/value pairs from the database, respectively. Just as *putIfVersion* ensures read-modify-write semantics, *deleteIfVersion* provides deletion of a specific version.

The *get* method retrieves items from the database.

The code sample below demonstrates the use of the various CRUD APIs. All code samples assume that you have already opened an Oracle NoSQL Database, referenced by the variable *store*. Although support for storing data using Avro records is available and encouraged, we do not show this in the code samples.

CRUD Examples

```
// Put a new key/value pair in the database, if key not already present.
Key key = Key.createKey("Katana");
String valString = "sword";

store.putIfAbsent(key, Value.createValue(valString.getBytes()));

// Read the value back from the database.
ValueVersion retValue = store.get(key);

// Update this item, only if the current version matches the version I read.
// In conjunction with the previous get, this implements a read-modify-write
String newvalString = "Really nice sword";
Value newval = Value.createValue(newvalString.getBytes());

store.putIfVersion(key, newval, retValue.getVersion());

// Finally, (unconditionally) delete this key/value pair from the database.
store.delete(key);
```

Iteration

In addition to basic CRUD operations, Oracle NoSQL Database supports two types of iteration: unordered iteration over records and ordered iteration within a Major Key set.

In the case of unordered iteration over the entire store, the result is not transactional; the iteration runs at an isolation level of *read-committed*, which means that the result set will contain only key/value pairs that have been persistently written to the database, but there are no guarantees of semantic consistency across key/value pairs.

The API supports both individual key/value returns using several *storeIterator* methods and bulk key/value returns within a Major Key Path via the various *multiGetIterator* methods. The example below demonstrates iterating over an entire store, returning each key/value pair individually. Note that although the iterator returns only a single key/value pair at a time, the *storeIterator* method takes a second parameter of *batchSize*, indicating how many key/value pairs to fetch per network round trip. This allows applications to simultaneously use network bandwidth efficiently, while maintaining the simplicity of key-at-a-time access in the API.

Unordered Iteration Example

```
// Create Iterator.
Iterator<KeyValueVersion> iter = store.storeIterator(Direction.UNORDERED, 100);

// Now, iterate over the store.
while (iter.hasNext()) {
    KeyValueVersion keyVV = iter.next();
    Value val = keyVV.getValue();
    Key key = keyVV.getKey();

    System.out.println(val.toString() + " " + key.toString() + "\n");
}
```

Bulk Operation API

In addition to providing single-record operations, Oracle NoSQL Database supports the ability to bundle a collection of operations together using the *execute* method, providing transactional semantics across multiple updates on records with the same Major Key Path. For example, let's assume that we have the Major Key Path "Katana" from the previous example, with several different Minor Key Paths, containing attributes of the Katana, such as length and year of construction. Imagine that we discover that we have an incorrect length and year of construction currently in our store. We can update multiple records atomically using a sequence of operations as shown below.

Example of Wrapping a Sequence of Operations in a Transaction

```
// Create a sequence of operations.
OperationFactory of = store.getOperationFactory();
List<Operation> opList = new ArrayList<Operation>();

// Create major and minor path components.
List<String> majorComponents = new ArrayList<String>();
List<String> minorLength = new ArrayList<String>();
List<String> minorYear = new ArrayList<String>();

majorComponents.add("Katana");
minorLength.add("length");
minorYear.add("year");

Key key1 = Key.createKey(majorComponents, minorLength);
Key key2 = Key.createKey(majorComponents, minorYear);

// Now put operations in an opList.
String lenVal = "37";
String yearVal = "1454";

opList.add(of.createPut(key1, Value.createValue(lenVal.getBytes())));
opList.add(of.createPut(key2, Value.createValue(yearVal.getBytes())));

// Now execute the operation list.
store.execute(opList);
```

Administration

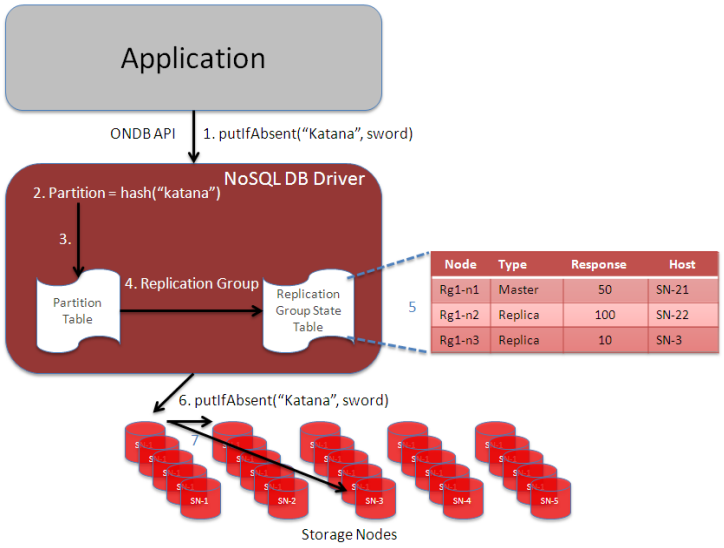
Oracle NoSQL Database comes with an *Administration Service*, accessible from both a Command Line Interface (CLI) and a web console. Using the CLI, administrators can configure a database instance, start it, stop it, and monitor system performance. The administrator can also expand the store by specifying additional nodes, in which case the system will automatically redistribute the data to the new systems without interrupting service..

The Administration Service is itself a highly-available service, but consistent with the Oracle NoSQL Database "No Single Point of Failure" philosophy, the ongoing operation of an installation is not dependent upon the availability of the Administration Service. Thus, both the database and the Administration Service remain available during configuration changes.

In addition to facilitating configuration changes, the Administration Service also collects and maintains performance statistics and logs important system events, providing online monitoring and input to performance tuning.

Architecture

We present the Oracle NoSQL Database architecture by following the execution of an operation through the logical components of the system and then discussing how those components map to actual hardware and software operation. We will create the key/value pair "Katana" and "sword". Figure 4 depicts the method invocation *putIfAbsent("Katana", "sword")*[1].



The application issues the *putIfAbsent* method to the Client Driver (step 1). The client driver hashes the key "Katana" to select one of a fixed number of partitions (step 2). The number of partitions is fixed and set by an administrator at system configuration time and is chosen to be significantly larger than the maximum number of storage nodes expected in the store. Storage nodes may be added to the system by the administrator and they are automatically populated – without stopping operations -- using data from existing storage nodes.

Figure 4: Request Processing

In this example, our store contains 25 storage nodes, so we might have configured the system to have 25,000 partitions. Each partition is assigned to a particular shard. The driver consults the partition table (step 3) to map the partition number to a *shard*.

A shard consists of some (configurable) number of *replication nodes*. Every shard consists of the same number of replication nodes. The number of replication nodes in a shard dictates the number of failures from which the system is resilient; a system with three nodes per shard can withstand two failures while continuing to service read requests. Its ability to withstand failures on writes is based upon the configured durability policy. If the application does not require a majority of participants to acknowledge a write, then the system can also withstand up to two failures for writes. A five-node

[1] Although the API takes byte[] and Avro records as Values, for convenience, we are showing the values as Strings.

group can withstand up to four failures for reads and up to two failures for writes, even if the application demands a durability policy requiring a majority of sites to acknowledge a write operation.

Given a shard, the Client Driver next consults the Shard State Table (SST) (step 4). For each shard, the SST contains information about each replication node comprising the group (step 5). Based upon information in the SST, such as the identity of the master and the load on the various nodes in a shard, the Client Driver selects the node to which to send the request and forwards the request to the appropriate node (step 6). In this case, since we are issuing a write operation, the request must go to the master node.

The replication node then applies the operation. In the case of a *putIfAbsent*, if the key exists, the operation has no effect and returns an error, indicating that the specified entry is already present in the store. If the key does not exist, the replication node adds the key/value pair to the store and then propagates the new key/value pair to the other nodes in the shard (step 7).

Implementation

An Oracle NoSQL Database installation consists of two major pieces: a *Client Driver* and a collection of *Storage Nodes*. As shown in Figure 3, the client driver implements the partition map and the SST, while storage nodes implement the replication nodes comprising shards. In this section, we'll take a closer look at each of these components.

Storage Nodes

A storage node (SN) is typically a physical machine with its own local persistent storage, either disk or solid state, a CPU with one or more cores, memory, and an IP address. A system with more storage nodes will provide greater aggregate throughput or storage capacity than one with fewer nodes, and systems with a greater degree of replication in shards can provide decreased request latency over installations with smaller degrees of replication. Storage nodes may be added to the system to improve capacity, decrease latency, and improve throughput.

A Storage Node Agent (SNA) runs on each storage node, monitoring that node's behavior. The SNA (a) receives configuration from, and (b) reports monitoring information to, the Administration Service which interfaces to the Oracle NoSQL Database monitoring dashboard. The SNA collects operational data from the storage node on an ongoing basis and then delivers it to the Administration Service when asked for it.

A storage node serves one or more replication nodes. Each replication node belongs to a single shard. The nodes in a single shard all serve the same data. Each shard has a designated master node that handles all data modification operations (create, update, and delete). The other nodes are read-only replicas, but may assume the role of master should the master node fail. A typical installation uses a replication factor of three in the shards, to ensure that the system can survive at least two simultaneous faults and still continue to service read operations. Applications requiring greater or lesser reliability can adjust this parameter accordingly.

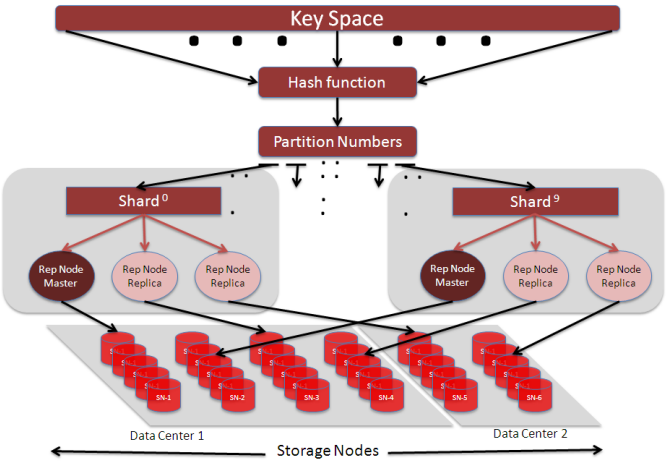


Figure5: Architecture

Figure 5 shows an installation with 10 shards (0-9). Each shard has a replication factor of 3 (one master and two replicas) spread across two data centers. Note that we place two of the replication nodes in the larger of the two data centers and the last replication node in the smaller one. This sort of arrangement might be appropriate for an application that uses the larger data center for its primary data access, maintaining the smaller data center in case of catastrophic failure of the primary data center. The 10 shards are stored on 30 storage nodes, spread across the two data centers.

Replication nodes support the Oracle NoSQL Database API via RMI calls from the client and obtain data directly from or write data directly to the log-structured storage system, which provides outstanding write performance, while maintaining index structures that provide low-latency read performance as well. The Oracle Berkeley DB Java Edition storage engine pioneered the use of log-structured storage in key/value databases since its initial deployment in 2003 and has been proven in several open-source NoSQL solutions, such as Dynamo, Voldemort, and GenieDB, as well as in Enterprise deployments.

Oracle NoSQL Database uses replication to ensure data availability in the case of failure. Its single-master architecture requires that writes are applied at the master node and then propagated to the replicas. In the case of failure of the master node, the nodes in a shard automatically hold a reliable election (using the Paxos protocol), electing one of the remaining nodes to be the master. The new master then assumes write responsibility. When multiple replication nodes reside on a storage node, the system will attempt to insure that no shard has more than one of its replication nodes on a single storage node.

Client Driver

The client driver is a Java jar file that exports the API to applications. In addition, the client driver maintains a copy of the *Topology* and the *Shard State Table* (SST). The Topology efficiently maps keys to partitions and from partitions to shards. For each shard, it includes the host name of the storage node hosting each replication node in the group, the service name associated with the replication nodes, and the data center in which each storage node resides. The client then uses the SST for two primary purposes: identifying the master node of a shard, so that it can send write requests to the master, and load balancing across all the nodes in a shard for reads. Since the SST is a critical shared data structure, each client and replication node maintains its own copy, thus avoiding any single point of failure. Both clients and replication nodes run a *RequestDispatcher* that use the SST to (re)direct write requests to the master and read requests to the appropriate member of a shard.

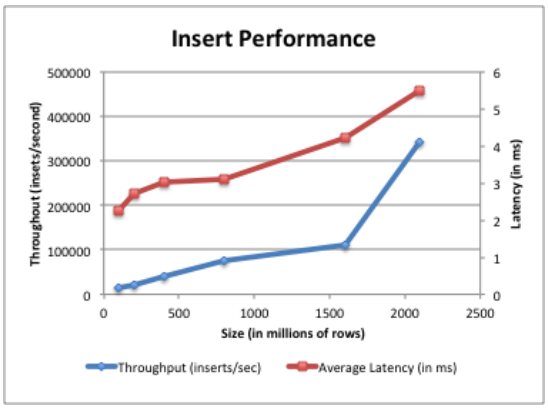
The Topology is loaded during client or replication node initialization and can subsequently be updated by the administrator if there are Topology changes like adding more storage nodes. The SST is dynamic, requiring ongoing maintenance. Each replication node runs a thread, called the *Replication Node State Update* thread that is responsible for ongoing maintenance of the SST. The update thread, as

well as the *RequestDispatcher*, opportunistically collect information on remote replication nodes including the current state of the node in its shard, an indication of how up-to-date the node is, the time of the last successful interaction with the node, the node's trailing average response time, and the current length of its outstanding request queue. In addition, the update thread maintains network connections and re-establishes broken ones. This maintenance is done outside the RequestDispatcher's request/response cycle to minimize the impact of broken connections on latency.

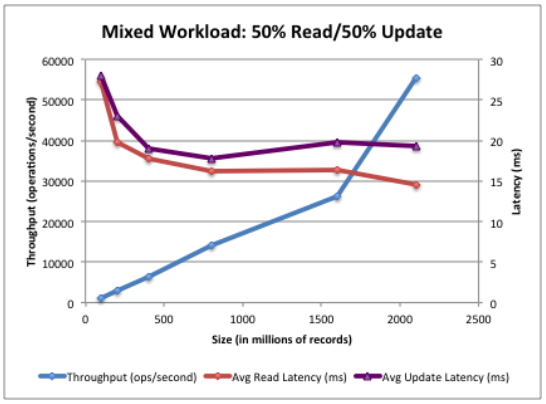
Performance

We have experimented with various Oracle NoSQL Database configurations and present the results of the Yahoo! Cloud Serving Benchmark (YCSB) against two Oracle NoSQL Database configurations, first demonstrating how the system scales with the number of nodes in the system and then demonstrating the upper end of the performance scale in mixed operations. As with all performance measurements, your own results may vary.

For the first test, we applied a constant YCSB load per storage node to configurations of varying sizes. Each storage node was comprised of a 2.93GHz Westmere 5670 dual socket machine with 6 cores/socket and 24GB of memory. Each machine had a 300GB local disk and ran RedHat 2.6.18-164.11.1.el5.crt1. At 300 GB, the disk size is the scale-limiting resource on each node, dictating the overall configuration, so we configured each node to hold 100M records, with an average key size of 13 bytes and data size of 1108 bytes.



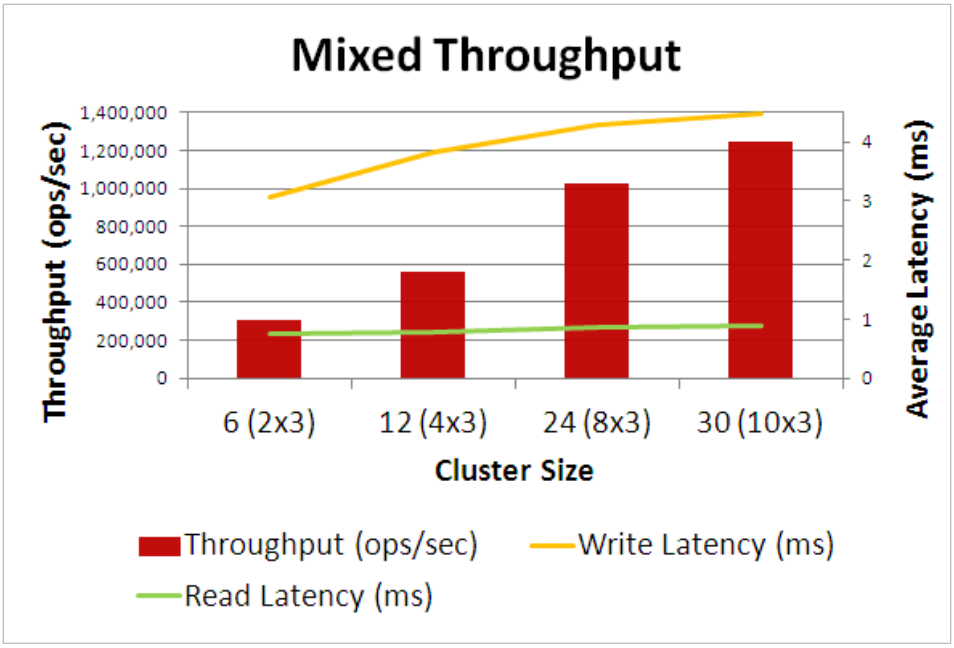
The graph to the left shows the raw insert performance of Oracle NoSQL Database for configurations ranging from a single shard system with three nodes storing 100 million records to a system with 32 shards on 96 nodes storing 2.1 billion records (the YCSB benchmark code used for this test was limited to a maximum of 2.1 billion records). The graph shows both the throughput in operations per second (blue line and left axis) and the response time in milliseconds (red line and right axis). Throughput of the system scales almost linearly as the database size and number of shards grows, with only a modest increase in response time.



The second graph shows the throughput and response time for a workload of 50% reads and 50% updates. As the system grows in size (both data size and number of shards), we see both the update and read latency decline, while throughput scales almost linearly, delivering the scalability needed for today's demanding applications.

The second test was designed to observe the upper end of the Oracle NoSQL Database performance scale with a stated target goal of 1 million mixed read/update operations per second.

We configured 15 storage nodes, each one a 2.9 GHz Xeon E5-2690 dual socket machine with 8 cores socket and 193GB of memory. Each machine had two 335GB PCI flash storage devices. We configured these nodes into 10 shards and each of the 30 replication nodes (either a master or a replica) used a single flash storage device. We tested with 2 billion records and a 95%/5% read/update YCSB mix.



The results were:

- 1.24 million mixed operations/sec
- 0.88 ms read latency (avg) and 2 ms in the 95'th percentile
- 4.3 ms update latency (avg) and 21 ms in the 95'th percentile

These results demonstrated that the system can meet the peak performance needs of many of the most demanding applications.

Conclusion

Oracle's NoSQL Database brings enterprise quality storage and performance to the highly-available, widely distributed NoSQL environment. Its commercially proven, write-optimized storage system delivers outstanding performance as well as robustness and reliability, and its "No Single Point of Failure" design ensures that the system continues to run and data remain available after any failure.

Oracle NoSQL Database
November 2012

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together