An Introduction to HyperDex and the Brave New World of High Performance, Scalable, Consistent, Fault-tolerant Data Stores

ROBERT ESCRIVA, BERNARD WONG, AND EMIN GÜN SIRER

Robert Escriva is a PhD student in computer science at Cornell University. He focuses on building infrastructure services for cloud computing.
escriva@cs.cornell.edu

Bernard Wong is an Assistant Professor in the School of Computer Science at the University of Waterloo. His research interests span distributed systems and networking, with particular emphasis on problems involving decentralized services, self-organizing networks, and distributed storage systems.
bernard.wong@uwaterloo.ca

Emin Gün Sirer is an Associate Professor of Computer Science at Cornell University. He works on infrastructure services for cloud computing and secure operating systems.
egs@systems.cs.cornell.edu

A new generation of data storage systems is now emerging to support high-performance, large-scale Web services whose demands are ill-met by traditional RDBMSes. Dubbed the NoSQL movement, this trend has produced systems characterized by data stores that provide weak consistency guarantees and limit the system interface. We argue that these systems have too aggressively capitulated, that much stronger consistency, availability, and fault-tolerance properties are possible, and, further, that it is possible to provide these properties while offering a rich API, although not as rich as full-blown SQL. We report on a recent system called HyperDex, describe the new techniques it uses to combine strong consistency and fault-tolerance guarantees with high-performance, and go through a scenario to see how the system can be used by real applications.

ACID and BASE

During the golden age of databases, when the canonical database users were banks and other financial institutions, providing strong guarantees of atomicity, consistency, isolation, and durability (ACID) were of paramount concern. More recently, however, the focus of data storage innovation has shifted away from supporting financial transactions to enabling Web services, such as Google, Facebook, and Amazon.com, that need to respond to queries efficiently, scale up to vast numbers of users, and tolerate the server failures that are inescapable at Web scale.

The flagship for this shift away from traditional RDBMS concerns towards properties that are better suited for Web services is a movement called NoSQL. This movement represents a constellation of new data storage systems that forego the traditional ACID guarantees of RDBMSs, along with their SQL interface, for improvements along the dimensions that matter to scalable Web applications. Although the NoSQL name suggests that the removal of SQL is the driving force behind the movement, it is really just the focal point for an overhaul of the storage system interface. For example, rather than having rigid schemas and support for complex search queries, most NoSQL systems have relaxed schemas and favor key-based operations whose implementation can be made scalable and efficient.

Yet the NoSQL movement has, in many ways, tossed the baby out with the bathwater. Most NoSQL systems subscribe to an alternative to ACID called the BASE approach, whose fundamental pillars are Basically Available service, Soft-State, and Eventually Consistent data. It is true that achieving Web scale will require hard tradeoffs between conflicting desires; yet the BASE approach represents a capitulation across *all* fronts. It provides no fault-tolerance guarantee and achieves no longevity for data, and typical BASE systems struggle to always return up-to-date results even with no failures. The name is catchy, but the resulting systems are quite weak and are useful only to a small niche of applications that can accept best-effort guarantees.

In this article, we provide a brief introduction to HyperDex, a second-generation distributed key-value store that is fast, scalable, strongly consistent, and fault-tolerant. By strongly consistent, we mean that a `get` will always return the latest value placed in the system by a `put`, not just eventually, but always, even during failures and reconfiguration. By fault-tolerant, we mean a system that can tolerate up to `f` failures, whether they are node (server) failures or network partitions affecting up to `f` hosts. And by fast, we mean a system with a streamlined implementation that, on the industry-standard YCSB benchmark, outperforms Cassandra [6] and MongoDB [1], two popular NoSQL systems, by a factor of 2 to 13. And above all, HyperDex supports a new lookup primitive by which objects stored in the system can be recalled by their attributes. Thus HyperDex combines the scalability and high performance properties of NoSQL systems with the consistency and fault-tolerance properties of RDBMSs, while providing a rich API. This unique combination of features is made possible by two novel techniques, *hyperspace hashing* and *value dependent chaining*, that determine the way HyperDex distributes its data.

Hyperspace Hashing

A key-value store, as its name suggests, provides users access to its data through key-based operations, such as `put` and `get`. Most large-scale key-value stores that support horizontal scaling either use a hashing function to map keys to nodes, such as Cassandra [6] and Dynamo [4], or partition the keyspace into contiguous regions that are assigned to different nodes by a centralized coordinator, such as BigTable [3] or HBase [2].

In contrast, HyperDex uses a new object placement method, called *hyperspace hashing*, that takes into account many object attributes when mapping objects to servers. Hyperspace hashing creates a multidimensional Euclidean space, where each dimension corresponds to one searchable attribute, that is, an attribute that may be used as part of a search query. An object's position in this space is specified by its coordinate, which can be determined by hashing the object's searchable attribute values. Objects' schemas are fixed, and different object types necessarily reside in different hyperspaces. Of course, nothing prevents a HyperDex deployment from having multiple spaces with the same hyperspace structure.

For example, a space of objects with "first name," "last name," and "phone number" searchable attributes corresponds to a three-dimensional hyperspace where each dimension corresponds to one attribute in the original object. Such a space is depicted in Figure 1. There are three objects in this space. The circular point is "John Smith" whose phone number is 555-8000. The square point is "John Doe" whose phone number is 555-7000. The diamond point is "Jim Bob" whose phone number is 555-2000. Anyone named "John" *must* map to somewhere in the plane labeled "John." Similarly, anyone with the last name "Smith" *must* map to somewhere within the plane labeled "Smith." Naturally, all people named "John Smith" must map to somewhere along the line where these two planes intersect.

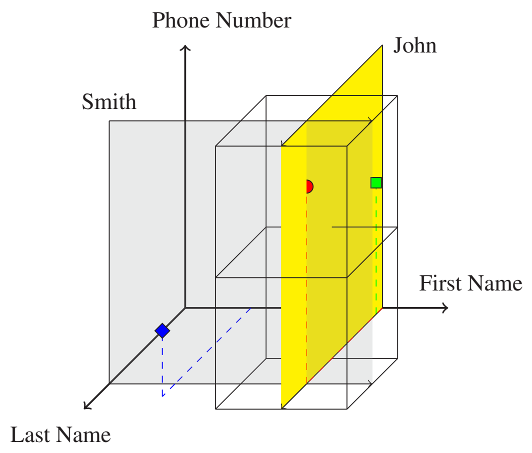

Figure 1: Simple hyperspace hashing in three dimensions. Each plane passes through all points corresponding to a specified query. Together the planes represent a line through all phone numbers for a given first name and last name pair. The cubes show two of the eight zones in this hyperspace each of which is handled by different servers.

For each space, HyperDex tessellates the hyperspace into disjoint pieces called *zones*, and assigns nodes (servers) to each zone. Figure 1 shows two of these assignments. Notice that the line for "John Smith" only intersects two out of the eight assignments. Consequently, performing a search for all phone numbers of "John Smith" requires contacting only two nodes. Furthermore, the search could be made more specific by restricting it to all people named "John Smith" whose phone number falls between 555-5000 and 555-9999. Such a search contacts only one out of the eight servers in this hypothetical deployment.

This simple object-mapping technique is not without pitfalls. Objects with many attributes translate to hyperspaces with many dimensions. The volume of the resulting hyperspace grows exponentially in the number of dimensions/attributes. A naïve approach would be to restrict the number of searchable attributes, and thus the size of the hyperspace. Such a technique limits the utility of hyperspace hashing. HyperDex avoids exponential growth of the hyperspace while maintaining the utility of hyperspace hashing by creating multiple independent and smaller hyperspaces, called *subspaces*. A large object may be represented in constant-size hyperspaces, the number of which is linear to the number of searchable attributes in the object. Here, HyperDex trades storage efficiency for search efficiency.

An additional pitfall with naïve hyperspace hashing is that key lookups would be equivalent to single attribute searches, which would likely be inefficient compared to key lookups in other key-value stores. Fortunately, using subspace partitioning, it is trivial to construct a subspace containing *just* the key of the object. This ensures that a `get` operation will always contact exactly one server in this subspace.

Value-Dependent Chaining

In addition to providing good performance and scalability, a distributed storage system must also provide fault tolerance. Much like other distributed storage systems, HyperDex achieves fault tolerance through data replication. However, HyperDex's use of hyperspace hashing and subspace partitioning introduce additional challenges, as the two features in combination force the same object to be stored at more than one server, which in turn presents problems of consistency between these replicas. As the location of an object in each subspace can change with every object update, the location of the replicas will also change. The replication scheme must therefore be able to manage replica sets that change frequently.

One replication approach, used in NoSQL systems that preceded HyperDex, would be to use an eventually consistent update mechanism. Such a mechanism would allow each replica to accept updates, and at a later point, the updates would be propagated to the rest of the replicas. However, changes to the replica set from multiple concurrent updates could result in inconsistency across subspaces. This type of inconsistency can accumulate over time and result in significant divergence between the contents of different subspaces. Furthermore, detecting such divergences is non-trivial and likely involves some form of all-to-all communication.

Instead, HyperDex introduces a new replication protocol called *value-dependent chaining* that efficiently provides total ordering on replica set updates. In value-dependent chaining, each update is propagated to the affected server nodes through a well-defined linear pipeline. Updates flow down the chain, while acknowledgments flow back up the chain. The head of the chain is the node responsible for that object's key, called a point leader. Because all value dependent chains for the same object have the same point leader, all updates to that object can be fully ordered with respect to each other. Node failures lead to broken chains, which are fixed automatically by shifting all nodes below the point of breakage up a spot and adding a new spare node at the tail of the chain to restore the desired level of fault tolerance. Failures of the point leader are handled the same way, with the backup point leader becoming the new node responsible for that zone. This linear ordering ensures the invariant that there is never any confusion about which nodes have seen the most fresh updates; consequently, there is no need for expensive mechanisms such as voting, leader election, or quorum writes.

Value-dependent chains also provide an additional property for free: all key operations are strongly consistent. The same chaining mechanisms that consistently update the replica set ensure consistent updates to the objects, without any overhead beyond what is required to maintain consistency of the replica set.

Tutorial

HyperDex has been fully implemented and is freely available for download. It includes all of the features we have described in this article. It is also being actively developed, with a small but growing development community that is eager to add developer-friendly features and additional language bindings. In this section, we will illustrate how a simple phonebook application uses HyperDex as its storage back-end.

Creating a HyperDex Space

A phonebook application needs to, at a bare minimum, keep track of a person's first name, last name, and phone number. In order to distinguish unique users, it might assign to each a user ID. We can instruct HyperDex to create a suitable space for holding such objects with the following command:

```
hyperdex-coordinator-control
        --host 127.0.0.1 --port 6970
        add-space << EOF
```

```
space phonebook
dimensions username, first, last
      phone (int64)
key username auto 1 3
subspace first, last, phone auto 3 3
EOF
```

This command creates a new space called phonebook that stores objects with the following four searchable attributes: username, first name, last name, and phone number. In this example, the space creation command instructs HyperDex to create a 1-dimensional subspace for the key, and a 3-dimensional subspace for the remaining attributes.

The replication level is specified by the "1 3" and "3 3" parameters at the end of the key and subspace line. This instructs HyperDex to divide the key subspace into 21 zones and the subspace for the remaining attributes into 23 zones, and to replicate each zone on to three nodes. As a general rule, a HyperDex administrator should configure HyperDex to not have significantly more zones per subspace than the number of nodes in the deployment.

Basic Operations

With a hyperspace defined, our phonebook application can connect to HyperDex and begin issuing basic get and put requests. We illustrate the HyperDex API using our Python client.

```
import hyperclient
c = hyperclient.Client('127.0.0.1', 1234)
```

This code snippet instructs the client bindings to talk to the HyperDex controller and retrieve the current HyperDex configuration. The controller ensures that the clients always receive the most up-to-date configuration. If the configuration changes, say, due to failures, the servers will detect that a client is operating with an out-of-date configuration and instruct it to retry with the updated HyperDex configuration.

Now that our phone application has created a client, it can insert objects in the system by issuing `put` requests:

```
c.put('phonebook', 'jsmith1',
      {'first': 'John', 'last': 'Smith', 'phone': 6075551024})
True
c.put('phonebook', 'jd',
      {'first': 'John', 'last': 'Doe', 'phone': 6075557878})
True
```

The client determines the unique location in the hyperspace for an object, contacts the servers responsible, and issues the `put` request to these servers. Similarly, our phone application can retrieve the jsmith1 object by issuing a get request.

```
c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075551024}
```

Our phone application can also use HyperDex's search primitive to retrieve objects based on one or more secondary attributes.

```
[x for x in c.search('phonebook',
            {'first': 'John', 'last': 'Smith', 'phone': 6075551024})]
[{'first': 'John', 'last': 'Smith',
             'phone': 6075551024,
             'username': 'jsmith1'}]
[x for x in c.search('phonebook', {'first': 'John'})]
[{'first': 'John', 'last': 'Smith', 'phone': 6075551024, 'username': 'jsmith1'},
{'first': 'John', 'last': 'Doe', 'phone': 6075557878, 'username': 'jd'}]
[x for x in c.search('phonebook', {'last': 'Smith'})]
[{'first': 'John', 'last': 'Smith', 'phone': 6075551024, 'username': 'jsmith1'}]
[x for x in c.search('phonebook', {'last': 'Doe'})]
[{'first': 'John', 'last': 'Doe', 'phone': 6075557878, 'username': 'jd'}]
```

Should the user decide to remove "John Doe" from his/her phonebook, the phonebook application can remove the object by issuing a `delete` request:

```
c.delete('phonebook', 'jd')
True
[x for x in c.search('phonebook', {'first': 'John'})]
[{'first': 'John', 'last': 'Smith', 'phone': 6075551024, 'username': 'jsmith1'}]
```

Finally, if the user wants to locate everyone named "John Smith" from Ithaca (area code 607), the phonebook application can issue the following range query to HyperDex:

```
[x for x in c.search('phonebook',
            {'last': 'Smith', 'phone': (6070000000, 6080000000)})]
[{'first': 'John', 'last': 'Smith', 'phone': 6075551024, 'username': 'jsmith1'}]
```

Atomic Read-Modify-Write Operations

HyperDex offers several atomic read-modify-write operations which are impossible to implement in key-value stores with weaker consistency guarantees. These operations, in turn, enable concurrent applications that would otherwise be impossible to implement correctly using non-atomic operations. For instance, using standard `get` and `put` operations, an application cannot ensure that its operations will not be interleaved with operations from other clients.

The canonical example for needing atomic read-modify-write operations involves two clients who are both trying to update a salary field. One is trying to deduct taxes—let's assume that they are hard-working academics being taxed at the maximum rate of 36%. The other client is trying to add a $1500 teaching award to the yearly salary. So one client will be doing:

```
v1=get(salary), v1 = v1 - 0.36*v1; put(salary, v1)
```

while the other client will be doing:

```
v2=get(salary), v2 += 1500; put(salary, v2)
```

where v1 and v2 are variables local to each client. Since these get and put operations can be interleaved in any order, it is possible for the clients to succeed (so both the deduction and the raise are issued) and yet for the salary to not reflect the results! If the sequence is get from client1, get from client2, put from client2, put from client1, the raise will be overwritten—a most undesirable outcome.

Atomic read-modify-write operations provide a solution to this problem. Such operations are guaranteed to execute without being interrupted by or interleaved with any other operation.

The word "atomic" is often associated with poor performance; however, HyperDex's atomic operations are inexpensive and virtually indistinguishable from a put, thanks to the use of value-dependent chains. The head of each object's value-dependent chain is in a unique position to locally compute the result of the atomic operation and, should it succeed, pass the operation down the chain as a normal `put`. Should the operation fail, the remainder of the value-dependent chain does not need to be involved at all.

HyperDex supports a few different atomic instructions, the most general of which is a conditional_put. A conditional_put performs the specified `put` operation if and only if the value being updated matches a specified condition.

Continuing with the sample phonebook application, consider extending the application for use in login authentication. The phonebook table must then be extended to include a password attribute. Intuitively, a user should only be able to change his/her password when it matches the password that he/she used to log in. The phonebook application can do this by using conditional_put:

```
c.conditional_put('phonebook', 'jsmith',
          {'password': 'currentpassword'},
          {'password': 'newpassword'})
True
c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075552048,
           'password': 'newpassword'}
```

Although this toy example omits certain implementation details relating to secure password storage, it is clear that the `conditional_put` operation enables behavior that is otherwise impossible to achieve with normal `get` and `put` operations. Any attempt to change the password without providing the previous password will fail:

```
c.conditional_put('phonebook', 'jsmith',
          {'password': 'wrongpassword'},
          {'password': 'newpassword'})
False
```

As expected, the `conditional_put` failed because the password is not, in fact, "wrongpassword".

HyperDex offers additional atomic operations. In many applications, the clients will want to increment or decrement a numerical field in the style of Google +1 and Reddit up/down votes. While implementing this is trivial with conditional_put, the implementation may require multiple attempts as the conditional_put operations fail in the face of contention. Atomic increment operations, in contrast, will not fail spuriously, and do not require the user to have retrieved the old value before starting the operation.

We further extend our sample phonebook application to track the number of times each user's information is viewed by adding a "lookups" attribute. The phonebook application can consistently manage this counter using the atomic_increment operation:

```
c.atomic_increment('phonebook', 'jsmith1', {'lookups': 1})
True
```

The atomic increment is as inexpensive as a put operation. This enables our application to log each lookup quickly and efficiently.

Asynchronous Operations

So far, we submitted synchronous operations to the key-value store, where the client had just a single outstanding request and waited patiently for that request to complete. In high-throughput applications, clients may have a batch of operations they want to perform on the key-value store. The standard practice in such cases is to issue asynchronous operations, where the client does not immediately wait for each individual operation to complete. HyperDex has a very versatile interface for supporting this use case.

Asynchronous operations allow a single client library to achieve higher throughput by submitting multiple simultaneous requests in parallel. Each asynchronous operation returns a small token that identifies the outstanding asynchronous operation, which can then be used by the client, if and when needed, to wait for the completion of selected asynchronous operations.

Every operation we've covered so far in the tutorials (e.g., get) has a corresponding version prefixed with async_ for performing that operation asynchronously. The basic pattern of usage for asynchronous operations is to initiate the asynchronous operation, do some work, perhaps issue more operations, and then wait for selected asynchronous operations to complete. This enables the application to continue to do other work while HyperDex performs the requested operations.

Here's how we could insert the "jsmith" user asynchronously:

```
d = c.async_put('phonebook', 'jsmith1',
       {'first': 'John', 'last': 'Smith',}
        'phone': 6075551024})
d
<hyperclient.DeferredInsert object at 0x7f2bbc3252d8>
do_work()
d.wait()
True
d = c.async_get('phonebook', 'jsmith1')
d.wait()
{'first': 'John', 'last': 'Smith', 'phone': 6075551024}
```

Notice that the return value of the first d.wait() is True. This is the same return value that would have come from performing c.put(...), except the client was free to do other computations while HyperDex servers were processing the put request. Similarly, the second asynchronous operation, async_get, queues up the request on the servers, frees the client to perform other work, and yields its results only when wait is called.

This allows for powerful applications. For instance, it is possible to issue thousands of requests and then wait for each one in turn without having to serialize the round trips to the server. Note that HyperDex may choose to execute concurrent asynchronous operations in any order. It's up to the programmer to order requests by calling wait appropriately.

Fault Tolerance

HyperDex provides a strong fault-tolerance guarantee to its clients. Anywhere during the preceding tutorial, feel free to kill off up to two of the nodes in the system. You will be able to continue the tutorial, as the value-dependent chains will detect the failures and route around them. If you bring up new nodes, they will be integrated into the chains seamlessly by the coordinator. The particular fault-tolerance level `f`, which determines the number of simultaneous failures a space can withstand, is entirely up to the application to determine. Of course, there are tradeoffs; while a large `f` will yield a more robust system, it will also increase operation latencies, and the improvement in actual reliability is subject to diminishing returns. The critical issue here is that this tradeoff is not part of the HyperDex substrate but is left up to applications to determine.

Performance

In an accompanying report [5], we carefully quantify HyperDex's performance using the industry-standard YCSB benchmark against Cassandra and MongoDB. While a similar performance study is beyond the scope of this introduction to HyperDex, we will report the major takeaway: HyperDex is very fast. It is approximately 2 to 13 times faster than the fastest of the other two NoSQL systems. There are two reasons for this huge gap in performance, which is even more striking because the other two systems are left in their preferred configurations, where they provide weak fault-tolerance and consistency guarantees. First, hyperspace hashing provides an enormous speedup for search-oriented operations. There is a qualitative difference between systems that enumerate objects by iterating through the keyspace and HyperDex, which can use the hyperspace to efficiently pick the desired items, so the 13x improvement could have been even higher if the benchmark's dataset had been larger. Second, HyperDex has a more streamlined implementation that is 2 to 4 times faster than Cassandra and MongoDB even at traditional `get`/`put` workloads. The precise details of the comparisons are in the technical report, and the beauty of open source is that there is tangible proof in a public repository that anyone can trivially check out and execute.

Summary

The emergence of large-scale Web applications has significantly altered the trajectory of distributed storage systems. From the radically different requirements of Web applications, NoSQL systems have emerged to fill the gap left by traditional databases. Early NoSQL systems used simple techniques, such as consistent hashing and parallel RPCs, to distribute their data, and thus were not able to make nuanced tradeoffs between desirable properties. In this article we presented HyperDex, a new high-performance key-value store that provides strong consistency guarantees, fault-tolerance against failures whose maximum size can be bounded, and high performance coupled with a rich API. These techniques are made possible through the use of hyperspace hashing and value-dependent chaining, two novel techniques for laying out and managing data. We hope that HyperDex, with its strong consistency and fault-tolerance guarantees, high performance, and rich API, will enable a new class of applications that were not served well by existing NoSQL systems.

Acknowledgments

We would like to thank the HyperDex open source community for their contributions, feedback, and support. In addition, we would like to highlight the extensive contributions of Pawel Loj and By Zhang, who have submitted substantial functionality to improve HyperDex.

References

[1] 10gen, Inc.: http://www.mongodb.org, accessed November 29, 2011.

[2] Apache Software Foundation: http://hbase.apache.org, accessed November 29, 2011.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber, "Bigtable: A Distributed Storage System for Structured Data," *Proceedings of OSDI*, November 2006, pp. 205–218.

[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshmanx, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels, "Dynamo: Amazon's Highly Available Key-value Store," *Proceedings of SOSP*, October 2007, pp. 205–220.

[5] Robert Escriva, Bernard Wong, and Emin Gün Sirer, "HyperDex: A Distributed, Searchable Key-Value Store for Cloud Computing," Computer Science Department, Cornell University Technical Report, December 2011.

[6] Avinash Lakshman and Prashant Malik, "Cassandra—A Decentralized Structured Storage System," *Proceedings of LADIS*, October 2009.