HyperDex Reference Manual

Release 0.4.0

Robert Escriva, Bernard Wong, and Emin Gün Sirer

June 05, 2012

CONTENTS

CHAPTER

ONE

CONCEPTS

HyperDex is a distributed, searchable key-value store. The key features of HyperDex are:

- **Fast**: HyperDex has lower latency, higher throughput, and lower variance than other key-value stores.
- **Scalable**: HyperDex scales as more machines are added to the system.
- **Consistent**: HyperDex guarantees linearizability for key-based operations. Thus, it a GET always returns the latest value inserted into the system. Not just "eventually," but immediately and always.
- **Fault tolerant**: HyperDex automatically replicates data on multiple machines so that concurrent failures, up to an application-determined limit, will not cause data loss.
- **Searchable**: HyperDex enables lookups of non-primary data attributes. Such searches are implemented efficiently and contact a small number of servers.

1.1 A New Approach to NoSQL

HyperDex embodies a unique set of features not found in other NoSQL systems or RDBMSs. Like many NoSQL systems and unlike traditional RDBMSs, it provides very high performance and very high scalability. Yet it does not achieve its high performance by compromising on consistency. To the contrary, it provides uniquely strong consistency guarantees while outperforming other NoSQL systems. Further, HyperDex provides well-defined fault-tolerance guarantees against both node failures and network partitions. And it achieves these performance, consistency, availability and fault tolerance properties while providing a richer API than most other NoSQL stores; specifically, it enables clients to recall objects using attributes other than the object key.

HyperDex derives its strong consistency, fault-tolerance and performance properties from two aspects of its design: hyperspace hashing, which forms the key organizing principle behind the system, and value-dependent chaining, a technique for replicating hyperspace hashed objects to achieve fault-tolerance. Since the latter topic is entirely invisible to users, we defer its detailed description to the accompanying white papers, and instead focus on hyperspace hashing in this document.

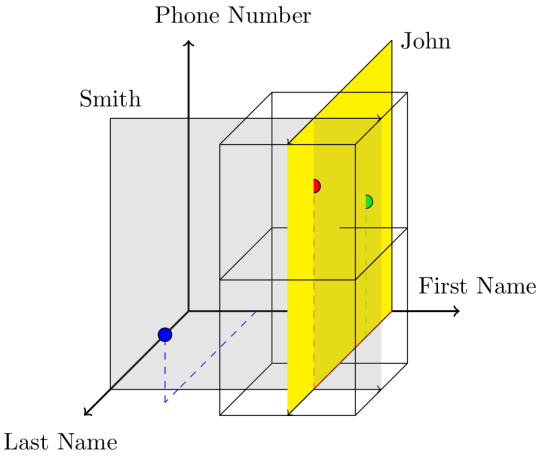
Hyperspace Hashing is a new technique for determining how to distribute objects onto nodes in a cluster. Specifically, it maps objects with multiple attributes into points in a multidimensional hyperspace, which has in turn been divided into zones assigned to servers. Search queries on secondary object attributes can therefore be mapped to small, hyperspace regions representing the set of feasible locations for the matching objects. This geometric mapping enables efficient searches that do not require enumerating across every object in the system.

The following sections detail how HyperDex places servers and objects in the Hyperspace, and addresses classic problems with using high dimensional data-structures.

1.2 Node and Object Placement

HyperDex strategically places objects on servers so that both search and key-based operations contact a small subset of all servers in the system. Whereas typical key-value stores map objects to nodes using just the key, HyperDex takes into account all attributes of an object when mapping it to servers.

HyperDex uses Hyperspace Hashing to map objects to points in a multidimensional space. Hyperspace hashing creates a multidimensional euclidean space for each table, where a table holds objects with the same attribute types, and each attribute type corresponds to a dimension in the euclidean space. HyperDex determines the position of each object in this space by hashing all of the object's attribute values to determine a spatial coordinate.

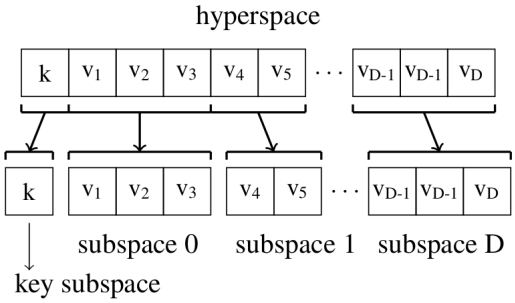


For example a table of objects with "first name", "last name" and "phone number" attributes corresponds to a three dimensional hyperspace where each dimension corresponds to one object attribute. Such a space is depicted in the diagram above. In this space, there exist three objects. The red point is "John Smith" whose phone number is 555-8000. The green point is "John Doe" whose phone number is 555-7000. The blue point is "Jim Bob" whose phone number is 555-2000. Anyone named "John" must map to somewhere in the yellow plane. Similarly, anyone with the last name "Smith" must map to somewhere within the translucent plane. Naturally, all people named "John Smith" must map to somewhere along the line where these two planes intersect.

In each multi-dimnesional space corresponding to a type of object, HyperDex assigns nodes to disjoint regions of the space, which we call zones. The example figure shows two of these assignments. Notice that the line for "John Smith" only intersects two out of the eight zones. 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.

1.3 Subspaces

HyperDex's Euclidean space construction enables geometric reasoning to significantly restrict the set of nodes that need to be contacted to find matching objects. However, a naive Euclidean space construction can suffer from the "curse of dimensionality", as the space exhibits an exponential increase in volume with each additional secondary attribute. For objects with many attributes, the resulting Euclidean space would be large, and consequently, sparse. Nodes would then be responsible for large regions in the hyperspace, which would increase the number of nodes whose regions intersect search hyperplanes and thus limit the effectiveness of the basic approach.



HyperDex addresses the exponential growth of the search space by introducing an efficient and lightweight mechanism that partitions the data into smaller, limited-size subspaces, where each subspace covers a subset of object attributes in a lower dimensional hyperspace. Thus, by folding the hyperspace back into a lower number of dimensions, HyperDex can ensure higher node selectivity during searches. The figure above shows how HyperDex can represent a table with `D` searchable attributes as a set of subspaces `s`.

Data partitioning increases the efficiency of a search by reducing the number of nodes which must be contacted to perform a search. For example, consider a table with 9 secondary attributes. A simple hyperspace over this whole table would require 512 zones to provide two regions along each dimension of the hyperspace. A search over 3 attributes would need to contact exactly 64 zones. If, instead, the same table were created with 3 subspaces of 3 dimensions each, each subspace can be filled with exactly 8 nodes. A search with no specificity in this table will need to contact 8 nodes. A search which specifies all the attributes in one subspace will contact exactly one node. If a search includes attributes from multiple subspaces, it selects the subspace with the most restrictive search hyperplane and performs the search in that subspace. Such a partitioned table provides a worst case bound on the number of server nodes contacted during a search.

1.4 Key Subspace

HyperDex's basic hyperspace construction scheme, as described so far, does not distinguish the key of an object from its secondary attributes. This leads to two significant problems when implementing a key-value store. First, key lookups would be equivalent to single attribute searches. Although HyperDex provides efficient search, a single attribute search in a multi-dimensional space would likely involve at least two zones. In this hypothetical scenario, key operations would be strictly more costly than than key operations in competing key-value stores. Second, because keys may reside on multiple nodes, they would not necessarily be unique, which may violate the uniqueness invariant applications have come to expect from key-value stores.

The preceding data partitioning technique enables a natural way to fix these issues by creating a dedicated key subspace. The one-dimensional key subspace maps each key to exactly one zone in the subspace. This is because the key fully specifies the position of the object in the subspace. To ensure uniqueness, `put` operations are applied to the key subspace before the remaining subspaces. If an object with the same key already exists, it is deleted from all subspaces at the same time the new object is being inserted. By introducing a one-dimensional key subspace, HyperDex provides efficient key operations and ensures system-wide key uniqueness.

CHAPTER

TWO

INSTALLING HYPERDEX

For the purposes of this document, we will need a working copy of HyperDex. This document makes use of features available in HyperDex 0.4 and later. There are two ways of installing the most recent version of the system: from precompiled binaries, or from source.

2.1 Installing Precompiled Binaries

The easiest way to install the system is through the precompiled packages. The HyperDex distribution currently supports Debian, Ubuntu and Fedora. The HyperDex download page has instructions on how to add the HyperDex repository so a simple install operation will get the most recent binaries on your system.

The precompiled binaries have the necessary dependencies built into them so installing them should pull in all other required components.

2.2 Installing From Source

Building from source is recommended when prebuilt packages are unsuitable or unavailable for your environment. In addition to the source tarballs provided below, you'll need to install Google's CityHash (we test against 1.0.x), Google's glog (we test against 0.3.x) and libpopt (we use 1.16). Additionally, you'll need libpo6 (version 0.2.3 or newer), libe (version 0.2.7 or newer), busybee (version 0.1.0 or newer) and the HyperDex source distribution. The latter four can be found on the HyperDex download page.

HyperDex is extremely easy to build from source. Once you've installed all of the prerequisites, you can install HyperDex with:

```
$ ./configure --enable-python-bindings --prefix=/path/to/install
$ make
# make install
```

2.3 Summary

Once you have HyperDex installed, you should be able to start the coordinator and daemon processes as follows.

```
$ hyperdex-coordinator -?
$ hyperdex-daemon -?
```

Once you get to this stage, HyperDex is ready to go. We can set up a data store and examine how to place values in it, which is the topic of the next chapter.

CHAPTER

THREE

BASIC OPERATIONS

A HyperDex cluster consists of three types of components: clients, servers, and a coordinator.

Applications built on top of HyperDex are the clients. They use the HyperDex API through various bindings for popular languages (e.g. C, C++, and Python) to issue operations, such as GET, PUT, SEARCH, DELETE, etc, to the storage system.

The HyperDex servers are responsible for storing the data in the system. You can have a cluster with as few as just a single server, though typical installations will have dozens to hundreds of servers. These servers store the data in memory and on disk, and they can crash at any time. HyperDex can be configured to tolerate as many failed nodes as desired.

The HyperDex coordinator maintains the "hyperspace." This involves making sure that servers are up, detecting failed or slow nodes, taking them out of the system, and replacing them where necessary. The coordinator maintains a critical data structure, the hyperspace map, that establishes the mapping between the hyperspace and servers. Clients use this map to locate the servers they need to contact, while servers use it to perform object propagation and replication to achieve the application's desired goals.

In this tutorial, we'll discuss how to get a HyperDex cluster up and running. In particular, we'll create a simple space, insert objects into it, retrieve those objects, and then perform queries over these objects.

3.1 Starting the Coordinator

First, we need to start up a coordinator that will oversee the organization of the hyperspace.

The following command starts the coordinator:

```
$ hyperdex-coordinator --control-port 6970 --host-port 1234 --logging debug
```

The coordinator has a control-port over which we can instruct it to rearrange the hyperspace and a host-port over which it communicates with server nodes. While a regular user should never have to interact with either of these ports directly, those of you who like to hack can always fire up a telnet session to the control port and issue commands directly to the coordinator.

At this point, the coordinator is up and running, and we're ready to start up additional nodes in our cluster.

3.2 Starting HyperDex Daemons

The HyperDex servers are the workhorse processes that actually house the data in the data store and respond to client requests. Let's start a server on the the same machine as the coordinator:

```
$ hyperdex-daemon --host 127.0.0.1 --port 1234 --bind-to 127.0.0.2 --data /path/to/data
```

The first two arguments are the IP and host port of the controller we started earlier. This enables the server to announce its presence, which in turn enables the coordinator to assign a zone to the newly arrived server. At that point, the server can take over some of the data load from an existing server, participate in the data propagation protocol, and start handling client requests. But since we have no data yet and this is our first node, this particular server does not have much work to do.

Note that the third argument (127.0.0.2) is the IP address to which we want to bind the server node. We are using a loopback address here for the purposes of the tutorial, while in real life you will undoubtedly want to use one of the IP addresses bound to your internal network.

The last argument is a pointer to a directory where the server will store all data.

It doesn't hurt to start a few more server instances at this point (although it is not required to continue with the tutorial).

You now have a functional HyperDex cluster. It's time to do something with it.

3.3 Creating a new Space

Let's imagine that we are building a phone book application. Such a phonebook would end up keeping track of people's first name, last name, and phone number. And to distinguish unique users, it might assign a user id to each user. Let's see how we can instruct HyperDex to create a suitable space for holding such objects:

```
$ 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 2 3
EOF
```

This command creates a new space called `phonebook`. Since we have four attributes (username, firstname, lastname, phone number) per object, we have chosen to create a four-dimensional space. Typically, we will assign a separate dimension to each searchable attribute, though we can assign fewer dimensions if desired. The resulting four-dimensional space is hard to visualize, but you can see why the name HyperDex is so apt.

Note that, under the covers, HyperDex will not necessarily create one giant hyperdimensional space. Doing so would cause lots of problems when trying to map objects with large numbers of attributes. Instead, we will typically want to create *subspace* consisting of smaller numbers of dimensions. The lower number of dimensions enable the mapping from points in space to nodes in the cluster to be more efficient; in particular, fewer nodes need to be contacted during search operations. In this simple example, we will want to create a 1-dimensional subspace for the object key so object lookups using just a single key can be resolved to a single host immediately, and a 3-dimensional subspace for the rest of the attributes. Let's see how this is done.

The `key` line designates the `username` attribute to be the key under which objects are stored and retrieved. The key plays a special role in HyperDex, though it's different from the role keys play in other NoSQL systems. In other NoSQL systems, objects can _only_ be retrieved by the key under which they were inserted. So an object `<rescrv, Robert, Escriva, 555-1212>` can only be retrieved by its key `rescrv`. In HyperDex, we will be able to perform retrievals for all Roberts or Escrivas or, even, reverse lookups by the phone number. The key simply serves as an object identifier such that updates to the object (e.g. changes to the phone number or name) are sequenced and handled consistently.

Since large scale cloud-computing deployments are sure to encounter failures, we will want to safeguard the data in our key-value store by creating replicas. The `1 3` at the end of the key line instructs the system to automatically divide the key subspace into `pow(2, 1)` zones and to replicate each zone on three nodes. Likewise, the subspace of the `first`, `last` and `phone` attributes will be divided into `pow(2, 2)` zones. Unless you started multiple servers earlier, each zone will only be replicated once.

As a general rule, we will want to automatically partition the hyperspace into a number of zones which is a power of two that is not significantly greater than the number of nodes in the cluster. A replication value of 0 does not make sense (what does it mean to have 0 replicas? we should just delete the item if we do not want it stored), 1 is fine for soft-state, and any value greater than 1 will enable us to tolerate failures in our server ensemble.

3.4 Interacting with the `phonebook` Space

Now that we have our hyperspace defined and ready to go, it's time to insert some information into our `phonebook`.

First, let's connect to HyperDex:

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

This line instructs the client bindings to talk to the controller and get the current hyperspace configuration. There is no need for static configuration files. Clients always receive the most up-to-date configuration (and 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 a fresh config).

Now that we have a workable client, we can put an object onto the servers:

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

This operation will determine the right spot in the hyperspace for this object, contact the servers responsible, and issue the `put` operation. The operation will only return once the object has been committed at all requisite nodes.

Now that we have an object in the phonebook, we can easily retrieve the `jsmith1` object by using a standard `get`:

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

Yay, we inserted an object under the key `jsmith1` and retrieved it using the same key. This looks exactly like every other NoSQL store out there, but there are a few differences.

First, it's blazingly fast. You can look in our latest performance graphs for the precise comparisons, but typically, HyperDex is just way faster than other key-value stores.

Second, it's fault-tolerant. When we performed the `put`, our operation was sent through a *value-dependent chain* of servers assigned to a particular point. The client received an acknowledgment only when the object was replicated on every single server in the chain. Unlike NoSQL stores that optimistically assume that an update was committed when it's in the send buffer of a single client (we're looking at you MongoDB), or when it's in the filesystem cache of a single server (we're looking at you Cassandra), HyperDex responds only when all the servers have been updated. And we can pick our replication levels to achieve any level of fault-tolerance we desire.

Finally, it's consistent. If we had multiple concurrent `put` operations being issued by multiple clients at the same time, we would never see an inconsistent state. What is an inconsistent state? It's what you get when you settle for *eventual consistency*. For instance, we would not want a prescription tracking system to say that we dispensed a drug, then to say we did not, only to settle on (say) having dispensed it. Yet this is precisely what might happen with an eventually consistent NoSQL key-value store. Eventual consistency is no consistency at all. In contrast, HyperDex provides linearizability. Time will never roll backwards from the point of any client.

And it gets better. For we can not only retrieve objects by their key, but we can also retrieve them when we don't know their key. Here are some examples:

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

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

Let's do that reverse phone number search:

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

Here's a fully-qualified search. Hyperspace hashing makes this nearly as fast as a key-based lookup:

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

Let's add another user named "John Doe":

```
>>> c.put('phonebook', 'jd', {'first': 'John', 'last': 'Doe', 'phone': 6075557878})
True
>>> [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 John Doe decide he no longer wants to be listed in the phonebook, it's trivial to remove his listing:

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

Suppose John Smith needs to change his phone number. This is easily accomplished by specifying just the key for the object and the changed attribute. All other attributes will be preserved (or be blank in the case where the object doesn't exist).

```
>>> c.put('phonebook', 'jsmith1', {'phone': 6075552048})
True
>>> c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075552048}
```

Smith is a popular name. Let's say there was "John Smith" from Rochester (area code 585):

```
>>> c.put('phonebook', 'jsmith2',
...          {'first': 'John', 'last': 'Smith', 'phone': 5855552048})
True
>>> c.get('phonebook', 'jsmith2')
{'first': 'John', 'last': 'Smith', 'phone': 5855552048}
```

Suppose we want to locate everyone named "John Smith" from Ithaca (area code 607). We can do this with a range query in HyperDex.

```
>>> [x for x in c.search('phonebook',
...  {'last': 'Smith', 'phone': (6070000000, 6080000000)})]
[{'first': 'John', 'last': 'Smith', 'phone': 6075552048, 'username': 'jsmith1'}]
```

CHAPTER

FOUR

ADVANCED TUTORIAL

HyperDex offers advanced features beyond the simple `GET`/`PUT`/`SEARCH` API presented in the basic tutorial. If you haven't checked out that tutorial yet, it's a great starting point as this tutorial builds upon it. The HyperDex client provides more advanced features, such as asynchronous operations, atomic read-modify-write operations, and transparent fault tolerance. These advanced features extract more performance and enable richer applications without sacrificing consistency.

4.1 Setup

The setup for this tutorial is very similar to that in the basic tutorial. First, we launch the coordinator:

```
$ hyperdex-coordinator --control-port 6970 --host-port 1234 --logging debug
```

Next, let's launch a few daemon processes. Execute the following commands (note that each instance has a different `/path/to/data`, as every node will be storing a different portion of the hyperspace (aka shard)):

```
$ hyperdex-daemon --host 127.0.0.1 --port 1234 --bind-to 127.0.0.2 --data /path/to/data1
$ hyperdex-daemon --host 127.0.0.1 --port 1234 --bind-to 127.0.0.2 --data /path/to/data2
$ hyperdex-daemon --host 127.0.0.1 --port 1234 --bind-to 127.0.0.2 --data /path/to/data3
```

This brings up three different daemons ready to serve in the HyperDex cluster. Finally, we create a space which makes use of all three systems in the cluster:

```
$ hyperdex-coordinator-control --host 127.0.0.1 --port 6970 add-space << EOF
space phonebook
dimensions username, first, last, phone (int64)
key username auto 0 3
subspace first, last, phone auto 0 3
EOF
```

This replicates data three times, allowing for up to two failures at any given time.

4.2 Asynchronous Operations

In the previous tutorial, 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 API method we've covered so far in the tutorials (e.g. `get`) has a corresponding `async_*` version (e.g. `async_get`) for performing asynchronous operations. The basic pattern of usage for asynchronous operations is:

- Initiate the asynchronous operation
- Do some work and perhaps issue more operations, async or otherwise,
- Wait for selected asynchronous operations to complete

This enables the application to continue doing other work while HyperDex performs the requested operations. Here's how we could insert the "jsmith1" 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.

By itself, an asynchronous operation is not very useful if it is waited on right away. The true power comes from requesting multiple concurrent operations:

```
>>> d1 = c.async_put('phonebook', 'jd',
...                  {'first': 'John', 'last': 'Doe', 'phone': 6075557878})
>>> d2 = c.async_get('phonebook', 'jsmith1')
>>> d1.wait()
True
>>> d2.wait()
{'first': 'John', 'last': 'Smith', 'phone': 6075551024}
```

Note that the order in which operations are waited on does not matter. We could just as easily execute them in a different order, and still get the desired effect:

```
>>> d1 = c.async_put('phonebook', 'jd',
...                  {'first': 'John', 'last': 'Doe', 'phone': 6075557878})
>>> d2 = c.async_get('phonebook', 'jsmith1')
>>> d2.wait()
{'first': 'John', 'last': 'Smith', 'phone': 6075551024}
>>> d1.wait()
True
```

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.

4.3 Atomic Read-Modify-Write Operations

Atomic read-modify-write operations enable concurrent applications that would otherwise be impossible to implement correctly. For instance, an application which performs a `GET` request followed by a `PUT` to the same key is not guaranteed to have these two requests operate immediately back to back, as other clients may issue requests to the key in the mean time.

The canonical example here 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%, not the cushy 15% that people on Wall Street seem to pay. 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). 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. We certainly cannot have that!

Atomic read-modify-write operations provide a solution to this problem. Such operations are guaranteed to execute without interference from other operations. The operation ensures that the read-modify-write sequence that comprises the operation is executed in a manner that cannot be interrupted by or interleaved with any other operation. The entire block is one atomic unit.

HyperDex supports a few different types of atomic instructions. Perhaps the most general one is the `condput`. A `condput` performs an `put` if and only if the value being updated matches a condition specified along with the new values to be inserted.

Since our sample database was set up with phone numbers, let's do some examples involving phone record updates. Let's say that the application wants to update John Smith's phone number, but wants the application to fail if the application has changed the phone number since it was last read:

```
>>> c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075551024}
>>> c.condput('phonebook', 'jsmith1',
...           {'phone': 6075551024}, {'phone': 6075552048})
True
>>> c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075552048}
```

Here, we told HyperDex to update John's phone number to end in 607-555-2048 if and only if it is currently equal to 607-555-1024. The third argument specified a set of object attributes that must match the object for the update to succeed. The fourth argument specified the set of new values to insert into the object in case of a match.

Not surprisingly, this request succeeded, as John's phone number matched the specified values. Let's try issuing the same operation again.

```
>>> c.condput('phonebook', 'jsmith1',
...           {'phone': 6075551024}, {'phone': 6075552048})
False
```

Notice that `condput` failed because the value of the phone number field is no longer 6075551024.

Note that the last argument has the same generality as the arguments to a regular `put` operation. So there is no requirement that a `condput` check and update the same field. The following is a perfectly legitimate operation that updates the first name field of object with key "jsmith" to "James" if the phone number has not changed:

```
>>> c.condput('phonebook', 'jsmith1',
...           {'phone': 6075552048}, {'first': 'James'})
True
```

The great thing about HyperDex is that `condput` operations are fast. In fact, their performance is indistinguishable from a normal `put`, all else being equal. Thus, you can rely heavily upon `condput` operations to avoid race conditions without sacrificing performance.

That's not all HyperDex offers in the way of atomic operations. In many applications, the clients will want to increment or decrement a numerical field. For instance, Google +1 and Reddit-style up/down-vote services will want to perform such arithmetic atomically. The way we set up our space and data for this example is not a good match for a good example, but let's pretend that John Smith has switched offices, and the application knows that this simply increments his phone number by 1. We could accomplish this with the following:

```
>>> c.atomic_add('phonebook', 'jsmith1', {'phone': 1})
True
>>> c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075552049}
>>> c.atomic_sub('phonebook', 'jsmith1', {'phone': 1})
True
>>> c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075552048}
```

Notice that each of these changes requires just one request to the server.

We can increment or decrement by any signed 64-bit value:

```
>>> c.atomic_add('phonebook', 'jsmith1', {'phone': 10})
True
>>> c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075552058}
>>> c.atomic_sub('phonebook', 'jsmith1', {'phone': 10})
True
>>> c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075552048}
```

Keep in mind that `condput` operations can and will fail, as intended, if there are interceding operations that update the object fields that must match. In these cases, the client will typically want to re-fetch the object, re-perform its updates, and re-submit the conditional operation.

Of course, it is perfectly reasonable to issue atomic operations asynchronously, as discussed in the preceding section. One would need to just use the `async_` prefix to the operations.

4.4 Fault Tolerance

HyperDex handles failures automatically. When a node fails, HyperDex will detect the failure, repair the subspace by eliminating the failed nodes from the value-dependent chains it is using to propagate values, and will automatically reintegrate any spare nodes into the space to restore the desired level of fault tolerance.

Let's see this in action by killing some nodes and checking what happens to our data:

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

So now we have a data item we deeply care about. We certainly would not want our NoSQL store to lose this data item because of a failure. Let's create a failure by killing one of the three hyperdaemon processes we started in the setup phase of the tutorial. Feel free to use "kill -9", there is no requirement that the nodes shut down in an orderly fashion. HyperDex is designed to handle crash failures.

```
>>> # kill a node at random
>>> c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075551024}
>>> c.put('phonebook', 'jsmith1', {'phone': 6075551025})
True
>>> c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075551025}
>>> c.put('phonebook', 'jsmith1', {'phone': 6075551026})
True
```

So, our data is alive and well. Not only that, but the subspace is continuing to operate as normal and handling updates at its usual rate.

Let's kill one more server.

```
>>> # kill a node at random
>>> c.get('phonebook', 'jsmith1')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "hyperclient.pyx", line 473, in hyperclient.Client.put ...
File "hyperclient.pyx", line 499, in hyperclient.Client.async_put ...
File "hyperclient.pyx", line 255, in hyperclient.DeferredInsert.__cinit__ ...
hyperclient.HyperClientException: Connection Failure
>>> c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075551026}
```

Note that the HyperDex API exposes some failures to the clients at the moment, so a client may have to catch HyperClientException and retry the operation. We plan to hide this behavior in the bindings in the near future, but for now, this is the behavior of the system.

In this example, behind the scenes, there were two node failures in the triply-replicated space. Each failure was detected, the space was repaired by cleaving out the failed node, and normal operations resumed without data loss.

CHAPTER

FIVE

DATASTRUCTURE TUTORIAL

As we have seen from the basic and advanced tutorials, HyperDex offers support for both the basic `GET`/`PUT`/`SEARCH` primitives, as well as advanced features, such as asynchronous and atomic read-modify-write operations, that are critical to enabling more sophisticated and demanding applications. So far, we have only illustrated HyperDex's features using strings and integers. In this tutorial, we explore HyperDex's richer datastructures: lists, sets, and maps. We will show that, by providing efficient atomic operations on these rich, native datastructures, HyperDex can greatly simplify space design for applications with complicated data layout requirements.

5.1 Setup

The setup for this tutorial is very similar to that in the basic tutorial. First, we launch the coordinator:

```
$ hyperdex-coordinator --control-port 6970 --host-port 1234 --logging debug
```

Next, let's launch a few daemon processes. Execute the following commands (note that each instance has a different `/path/to/data`, as every node will be storing a different portion of the hyperspace (aka shard)):

```
$ hyperdex-daemon --host 127.0.0.1 --port 1234 --bind-to 127.0.0.2 --data /path/to/data1
$ hyperdex-daemon --host 127.0.0.1 --port 1234 --bind-to 127.0.0.2 --data /path/to/data2
$ hyperdex-daemon --host 127.0.0.1 --port 1234 --bind-to 127.0.0.2 --data /path/to/data3
```

This brings up three different daemons ready to serve in the HyperDex cluster. Finally, we create a space which makes use of all three systems in the cluster. In this example, let's create a space that may be suitable for serving data in a social network:

```
$ hyperdex-coordinator-control --host 127.0.0.1 --port 6970 add-space << EOF
space socialnetwork
dimensions username, first, last,
           pending_requests (list(string)),
           hobbies (set(string)),
           unread_messages (map(string,string))
           upvotes (map(string,int64))
key username auto 0 3
subspace first, last auto 0 3
EOF
```

This space captures a user's profile including pending friend requests, hobbies and unread private messages. The data is replicated to tolerate up to two failures at any given time.

5.2 Lists

We've created a space for a user's profile in our social networking app. Let's add support for friend requests. For this we'll use the `pending_requests` attribute in the `socialnetwork` space.

To start, let's create a profile for John Smith:

```
>>> c.put('socialnetwork', 'jsmith1', {'first': 'John', 'last': 'Smith'})
True
>>> c.get('socialnetwork', 'jsmith1')
{'first': 'John', 'last': 'Smith',
 'pending_requests': [],
 'hobbies': set([]),
 'unread_messages': {},
 'upvotes': {}}
```

Imagine that shortly after joining, John Smith receives a friend request from his friend Brian Jones. Behind the scenes, this could be implemented with a simple list operation, pushing the friend request onto John's `pending_requests`:

```
>>> c.list_rpush('socialnetwork', 'jsmith1', {'pending_requests': 'bjones1'})
True
>>> c.get('socialnetwork', 'jsmith1')['pending_requests']
['bjones1']
```

The operation `list_rpush` is guaranteed to be performed atomically, and will be applied consistently with respect to all other operations on the same list.

5.3 Sets

Our social networking app captures the notion of hobbies. A set of strings is a natural representation for a user's hobbies, as each hobby is represented just once and may be added or removed.

Let's add some hobbies to John's profile:

```
>>> hobbies = set(['hockey', 'basket weaving', 'hacking',
...                'air guitar rocking']
>>> c.set_union('socialnetwork', 'jsmith1', {'hobbies': hobbies})
True
>>> c.set_add('socialnetwork', 'jsmith1', {'hobbies': 'gaming'})
True
>>> c.get('socialnetwork', 'jsmith1')['hobbies']
set(['hacking', 'air guitar rocking', 'hockey', 'gaming', 'basket weaving'])
```

If John Smith decides that his life's dream is to just write code, he may decide to join a group on the social network filled with like-minded individuals. We can use HyperDex's intersect primitive to narrow down his interests:

```
>>> c.set_intersect('socialnetwork', 'jsmith1',
...                 {'hobbies': set(['hacking', 'programming'])})
True
>>> c.get('socialnetwork', 'jsmith1')['hobbies']
set(['hacking'])
```

Notice how John's hobbies become the intersection of his previous hobbies and the ones named in the operation.

Overall, HyperDex supports simple set assignment (using the `put` interface), adding and removing elements with `Client.set_add()` and `hyperclient.Client.set_remove()`, taking the union of a set with `hyperclient.Client.set_union()` and storing the intersection of a set with `hyperclient.Client.set_intersect()`.

5.4 Maps

Lastly, our social networking system needs a means for allowing users to exchange messages. Let's demonstrate how we can accomplish this with the `unread_messages` attribute. In this contrived example, we're going to use an object attribute as a map (aka dictionary) to map from a user name to a string that contains the message from that user, as follows:

```
>>> c.map_add('socialnetwork', 'jsmith1',
...           {'unread_messages' : {'bjones1' : 'Hi John'}})
True
>>> c.map_add('socialnetwork', 'jsmith1',
...           {'unread_messages' : {'timmy' : 'Lunch?'}})
True
>>> c.get('socialnetwork', 'jsmith1')['unread_messages']
{'timmy': 'Lunch?', 'bjones1': 'Hi John'}
```

HyperDex enables map contents to be modified in-place within the map. For example, if Brian sent another message to John, we could separate the messages with "|" and just append the new message:

```
>>> c.map_string_append('socialnetwork', 'jsmith1',
...                     {'unread_messages' : {'bjones' : '| Want to hang out?'}})
True
>>> c.get('socialnetwork', 'jsmith1')['unread_messages']
{'timmy': 'Lunch?', 'bjones1': 'Hi John| Want to hang out?'}
```

Note that maps may have strings or integers as values, and every atomic operation available for strings and integers is also available in map form, operating on the values of the map.

For the sake of illustrating maps involving integers, let's imagine that we will use a map to keep track of the plus-one's and like/dislike's on John's status updates.

First, let's create some counters that will keep the net count of up and down votes corresponding to John's link posts, with ids "http://url1.com" and "http://url2.com".

```
>>> url1 = "http://url1.com"
>>> url2 = "http://url2.com"
>>> c.map_add('socialnetwork', 'jsmith1',
...           {'upvotes' : {url1 : 1, url2: 1}})
True
```

So John's posts start out with a counter set to 1, as shown above.

Imagine that two other users, Jane and Elaine, upvote John's first link post, we would implement it like this:

```
>>> c.map_atomic_add('socialnetwork', 'jsmith1', {'upvotes' : {url1: 1}})
True
>>> c.map_atomic_add('socialnetwork', 'jsmith1', {'upvotes' : {url1: 1}})
True
```

Charlie, sworn enemy of John, can downvote both of John's urls like this:

```
>>> c.map_atomic_add('socialnetwork', 'jsmith1', {'upvotes' : {url1: -1, url2: -1}})
True
```

This shows that any map operation can operate atomically on a group of map attributes at the same time. This is fully transactional; all such operations will be ordered in exactly the same way on all replicas, and there is no opportunity for divergence, even through failures.

Checking where we stand:

```
>>> c.get('socialnetwork', 'jsmith1')['upvotes']
{'http://url1.com': 2, , 'http://url2.com': 0}
```

All of the preceding operations could have been issued concurrently – the results will be the same because they commute with each other and are executed atomically.

5.5 Asynchronous Datastructure Operations

As with all other API methods in HyperDex, there are corresponding asynchronous methods for manipulating HyperDex datastructures. For example, the social networking application can make an asynchronous call to make friend requests:

```
>>> d = c.async_list_rpush('socialnetwork', 'jsmith1', {'pending_requests': 'timmy'})
>>> d.wait()
True
>>> c.get('socialnetwork', 'jsmith1')['pending_requests']
['bjones1', 'timmy']
```

Here, we issued an asynchronous operation on a list, waited for it to complete, and saw that the end result indeed reflected the effect of the asynchronous operation.

So, overall, HyperDex provides a very rich API with complex, aggregate datastructures. And it supports atomic operations on these datastructures such that concurrent clients can use the without the need to coordinate with an external lock server (in fact, if needed, they can use HyperDex to *implement* a high-performance lock server!).

CHAPTER

SIX

PYTHON API

class hyperclient.**Client**(*address*, *port*)

A client of the HyperDex cluster. Instances of this class encapsulate all resources necessary to communicate with nodes in a HyperDex cluster.

get(*space*, *key*)

A `get` request retrieves an object from a specific space using the object's key. If the object exists in the space at the time of the request, it will be returned; otherwise, the API will indicate that the object was not found.

Consistency: A `get` request will always see the result of all complete `put` requests.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of a `get` is one network round trip for cached objects, and one disk read for uncached objects.

On success, the returned object will be a `dict` mapping attribute names to their respective values. If the object does not exist, `None` will be returned. On error, a `HyperClientException` is thrown.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

put(*space*, *key*, *attrs*)

A `put` request creates or updates the attributes of an object stored in a specific space under a specific key. Only those attributes specified will be affected by the `put` operation. If the object does not exist, it will be created, and all specified attributes will be set to the values provided. All other attributes will be initialized to their default state.

Consistency: `put` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `put` request completes will see the result of that `put`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of a `put` is one network round trip per replica for cached objects, and one disk read per replica for uncached objects.

The return value is a boolean indicating success. On error, a `HyperClientException` is thrown.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary mapping attribute names to their respective values. Each value must match the type defined when `space` was created.

`condput`(*space*, *key*, *condition*, *attrs*)

A `condput` request creates or updates the attributes of an object stored in a specific space under a specific key if and only if certain attributes match the specified condition.. Only those attributes specified will be affected by the `condput` operation. If the object does not exist, the operation will fail. All unspecified attributes will be left to their current values.

Consistency: `condput` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `condput` request completes will see the result of that `condput`, or of a later operation which superseded it.

Efficiency: The dominating cost of a `condput` is the same as a `put` when the comparison succeeds. If the comparison fails, the cost is the same as a `get`.

The return value is a boolean indicating success. If the value is `True`, the comparison was successful, and the object was updated. If the value is `False`, the comparison failed, and the object remains unchanged.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

condition: A dictionary mapping attribute names to a value which is compared against the object's current state.

attrs: A dictionary mapping attribute names to their respective values. Each value must match the type defined when `space` was created.

`delete`(*space*, *key*)

A `del` request removes an object identified by a key from the specified space. If the object does not exist, the operation will do nothing.

Consistency: `del` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `del` request completes will see the result of that `del`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of a `del` is one network round trip per replica for cached objects, and one disk read per replica for uncached objects.

The return value is a boolean indicating success. If the value is `True`, the delete was successful, and the object was removed. If the value is `False`, there was no object to remove.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

`search`(*space*, *predicate*)

A `search` request retrieves all objects which match the specified predicate. These objects may reside on multiple nodes, and the client transparently handles retrieving them from each node. Multiple errors may be returned from a single search; however, the client library will continue to make progress until all servers finish the search or encounter an error.

Consistency: `search` results will see the result of every operation which completes prior to the search, unless there is a concurrent operation which supersedes the prior operation.

Efficiency: The dominating cost of a search is one network round trip per object. Multiple objects will be retrieved in parallel, so searches should be more efficient than simple object retrieval.

This object returns a generator of type `Search`. Both objects retrieved by the search, and errors encountered during the search will be yielded by the generator. Each object is a `dict` mapping attribute names to their respective values.

space: A string naming the space in which the object will be inserted.

predicate: A dictionary specifying comparisons used for selecting objects. Each key-value pair in `predicate` maps the name of an attribute to a value or range of values which constitute the search. An equality search is specified by supplying the value to match. A range search is a 2-tuple specifying the lower and upper bounds on the range.

atomic_add(*space*, *key*, *value*)

An `atomic_add` request adds the amount specified to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_add` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_add` request completes will see the results of that `atomic_add`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_add` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

atomic_sub(*space*, *key*, *value*)

An `atomic_sub` request subtracts the amount specified from the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_sub` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_sub` request completes will see the results of that `atomic_sub`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_sub` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

atomic_mul(*space*, *key*, *value*)

An `atomic_mul` request multiplies the amount specified by the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mul` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mul` request completes will see the results of that `atomic_mul`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mul` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

atomic_div(*space*, *key*, *value*)

An `atomic_div` request divides the value of each attribute specified by the request by the amount specified. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_div` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_div` request completes will see the results of that `atomic_div`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_div` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

atomic_mod(*space*, *key*, *value*)

An `atomic_mod` request computes the value of each attribute specified by the request modulo the amount specified. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mod` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mod` request completes will see the results of that `atomic_mod`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mod` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

atomic_and(*space*, *key*, *value*)

An `atomic_and` request stores the bitwise-and of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_and` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_and` request completes will see the results of that `atomic_and`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_and` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

atomic_or(*space*, *key*, *value*)

An `atomic_or` request stores the bitwise-or of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_or` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_or` request completes will see the results of that `atomic_or`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_or` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

atomic_xor(*space*, *key*, *value*)

An `atomic_xor` request stores the bitwise-xor of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_xor` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_xor` request completes will see the results of that `atomic_xor`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_xor` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

`string_prepend`(*space*, *key*, *value*)

A `string_prepend` request prepends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_prepend` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_prepend` request completes will see the results of that `string_prepend`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_prepend` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

`string_append`(*space*, *key*, *value*)

A `string_append` request appends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_append` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_append` request completes will see the results of that `string_append`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_append` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

list_lpush (*space*, *key*, *value*)

A `list_lpush` request pushes the specified object to the head of the list of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `list_lpush` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `list_lpush` request completes will see the results of that `list_lpush`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `list_lpush` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

list_rpush (*space*, *key*, *value*)

A `list_rpush` request pushes the specified object to the tail of the list of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `list_rpush` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `list_rpush` request completes will see the results of that `list_rpush`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `list_rpush` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

set_add (*space*, *key*, *value*)

A `set_add` request adds a single element to a set or does nothing if the element is already in the set. The insertion will be atomic without interference from other operations.

Consistency: `set_add` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_add` request completes will see the results of that `set_add`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_add` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

set_remove(*space*, *key*, *value*)

A `set_remove` request removes a single element to a set, or does nothing if the element is not in the set. The removal will be atomic without interference from other operations.

Consistency: `set_remove` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_remove` request completes will see the results of that `set_remove`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_remove` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

set_intersect(*space*, *key*, *value*)

A `set_intersect` performs set intersection, and stores the resulting set in the object. The intersection will be atomic without interference from other operations.

Consistency: `set_intersect` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_intersect` request completes will see the results of that `set_intersect`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_intersect` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

set_union(*space*, *key*, *value*)

A `set_intersect` performs set intersection, and stores the resulting set in the object. The intersection will be atomic without interference from other operations.

Consistency: `set_intersect` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_intersect` request completes will see the results of that `set_intersect`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_intersect` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

`map_atomic_add`(*space*, *key*, *value*)

An `atomic_add` request adds the amount specified to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_add` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_add` request completes will see the results of that `atomic_add`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_add` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

`map_atomic_sub`(*space*, *key*, *value*)

An `atomic_sub` request subtracts the amount specified from the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_sub` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_sub` request completes will see the results of that `atomic_sub`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_sub` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the

inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

map_atomic_mul(*space*, *key*, *value*)

An `atomic_mul` request multiplies the amount specified by the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mul` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mul` request completes will see the results of that `atomic_mul`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mul` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

map_atomic_div(*space*, *key*, *value*)

An `atomic_div` request divides the value of each attribute specified by the request by the amount specified. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_div` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_div` request completes will see the results of that `atomic_div`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_div` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

map_atomic_mod(*space*, *key*, *value*)

An `atomic_mod` request computes the value of each attribute specified by the request modulo the amount specified. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mod` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mod` request completes will see the results of that `atomic_mod`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mod` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

map_atomic_and(*space*, *key*, *value*)

An `atomic_and` request stores the bitwise-and of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_and` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_and` request completes will see the results of that `atomic_and`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_and` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

map_atomic_or(*space*, *key*, *value*)

An `atomic_or` request stores the bitwise-or of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_or` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_or` request completes will see the results of that `atomic_or`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_or` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

map_atomic_xor(*space*, *key*, *value*)

An `atomic_xor` request stores the bitwise-xor of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_xor` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_xor` request completes will see the results of that `atomic_xor`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_xor` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

map_string_prepend(*space*, *key*, *value*)

A `string_prepend` request prepends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_prepend` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_prepend` request completes will see the results of that `string_prepend`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_prepend` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

map_string_append(*space*, *key*, *value*)

A `string_append` request appends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_append` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_append` request completes will see the results of that `string_append`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_append` is the same as a `put`.

The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically added. If the value is `False`, the object was not found.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

async_get (*space*, *key*)

A `get` request retrieves an object from a specific space using the object's key. If the object exists in the space at the time of the request, it will be returned; otherwise, the API will indicate that the object was not found.

Consistency: A `get` request will always see the result of all complete `put` requests.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of a `get` is one network round trip for cached objects, and one disk read for uncached objects.

The returned object will be a `DeferredGet` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

async_put (*space*, *key*, *value*)

A `put` request creates or updates the attributes of an object stored in a specific space under a specific key. Only those attributes specified will be affected by the `put` operation. If the object does not exist, it will be created, and all specified attributes will be set to the values provided. All other attributes will be initialized to their default state.

Consistency: `put` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `put` request completes will see the result of that `put`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of a `put` is one network round trip per replica for cached objects, and one disk read per replica for uncached objects.

The returned object will be a `DeferredPut` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary mapping attribute names to their respective values. Each value must match the type defined when `space` was created.

async_condput (*space*, *key*, *condition*, *value*)

A `condput` request creates or updates the attributes of an object stored in a specific space under a specific key if and only if certain attributes match the specified condition.. Only those attributes specified will be affected by the `condput` operation. If the object does not exist, the operation will fail. All unspecified attributes will be left to their current values.

Consistency: `condput` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `condput` request completes will see the result of that `condput`, or of a later operation which superseded it.

Efficiency: The dominating cost of a `condput` is the same as a `put` when the comparison succeeds. If the comparison fails, the cost is the same as a `get`.

The returned object will be a `DeferredCondput` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

condition: A dictionary mapping attribute names to a value which is compared against the object's current state.

attrs: A dictionary mapping attribute names to their respective values. Each value must match the type defined when `space` was created.

async_delete(*space*, *key*)

A `del` request removes an object identified by a key from the specified space. If the object does not exist, the operation will do nothing.

Consistency: `del` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `del` request completes will see the result of that `del`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of a `del` is one network round trip per replica for cached objects, and one disk read per replica for uncached objects.

The returned object will be a `DeferredDelete` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

async_atomic_add(*space*, *key*, *value*)

An `atomic_add` request adds the amount specified to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_add` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_add` request completes will see the results of that `atomic_add`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_add` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_atomic_sub(*space*, *key*, *value*)

An `atomic_sub` request subtracts the amount specified from the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_sub` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_sub` request completes will see the results of that `atomic_sub`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_sub` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_atomic_mul(*space*, *key*, *value*)

An `atomic_mul` request multiplies the amount specified by the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mul` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mul` request completes will see the results of that `atomic_mul`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mul` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_atomic_div(*space*, *key*, *value*)

An `atomic_div` request divides the value of each attribute specified by the request by the amount specified. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_div` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_div` request completes will see the results of that `atomic_div`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_div` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_atomic_mod(*space*, *key*, *value*)

An `atomic_mod` request computes the value of each attribute specified by the request modulo the amount specified. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mod` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mod` request completes will see the results of that `atomic_mod`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mod` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_atomic_and(*space*, *key*, *value*)

An `atomic_and` request stores the bitwise-and of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_and` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_and` request completes will see the results of that `atomic_and`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_and` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_atomic_or(*space*, *key*, *value*)

An `atomic_or` request stores the bitwise-or of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_or` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_or` request completes will see the results of that `atomic_or`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_or` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_atomic_xor(*space*, *key*, *value*)

An `atomic_xor` request stores the bitwise-xor of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_xor` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_xor` request completes will see the results of that `atomic_xor`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_xor` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_string_prepend(*space*, *key*, *value*)

A `string_prepend` request prepends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_prepend` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_prepend` request completes will see the results of that `string_prepend`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_prepend` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_string_append(*space*, *key*, *value*)

A `string_append` request appends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_append` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_append` request completes will see the results of that `string_append`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_append` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_list_lpush(*space*, *key*, *value*)

A `list_lpush` request pushes the specified object to the head of the list of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `list_lpush` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `list_lpush` request completes will see the results of that `list_lpush`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `list_lpush` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_list_rpush(*space*, *key*, *value*)

A `list_rpush` request pushes the specified object to the tail of the list of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `list_rpush` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `list_rpush` request completes will see the results of that `list_rpush`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `list_rpush` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_set_add(*space*, *key*, *value*)

A `set_add` request adds a single element to a set or does nothing if the element is already in the set. The insertion will be atomic without interference from other operations.

Consistency: `set_add` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_add` request completes will see the results of that `set_add`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_add` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_set_remove(*space*, *key*, *value*)

A `set_remove` request removes a single element to a set, or does nothing if the element is not in the set. The removal will be atomic without interference from other operations.

Consistency: `set_remove` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_remove` request completes will see the results of that `set_remove`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_remove` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_set_intersect(*space*, *key*, *value*)

A `set_intersect` performs set intersection, and stores the resulting set in the object. The intersection will be atomic without interference from other operations.

Consistency: `set_intersect` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_intersect` request completes will see the results of that `set_intersect`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_intersect` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

async_map_atomic_add(*space*, *key*, *value*)

An `atomic_add` request adds the amount specified to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_add` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_add` request completes will see the results of that `atomic_add`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_add` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

async_map_atomic_sub(*space*, *key*, *value*)

An `atomic_sub` request subtracts the amount specified from the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_sub` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_sub` request completes will see the results of that `atomic_sub`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_sub` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

async_map_atomic_mul(*space*, *key*, *value*)

An `atomic_mul` request multiplies the amount specified by the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mul` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mul` request completes will see the results of that `atomic_mul`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mul` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the

inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

async_map_atomic_div(*space*, *key*, *value*)

An `atomic_div` request divides the value of each attribute specified by the request by the amount specified. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_div` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_div` request completes will see the results of that `atomic_div`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_div` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

async_map_atomic_mod(*space*, *key*, *value*)

An `atomic_mod` request computes the value of each attribute specified by the request modulo the amount specified. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mod` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mod` request completes will see the results of that `atomic_mod`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mod` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

async_map_atomic_and(*space*, *key*, *value*)

An `atomic_and` request stores the bitwise-and of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_and` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_and` request completes will see the results of that `atomic_and`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_and` is the same as a `put`.

The returned object will be a DeferredFromAttrs instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

async_map_atomic_or(*space*, *key*, *value*)

An atomic_or request stores the bitwise-or of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: atomic_or requests to the same key will be observed in the same order by all clients. All get requests which execute after a given atomic_or request completes will see the results of that atomic_or, or of a later operation which superseded it.

This operation is totally ordered with respect to all get, put, condput, delete and other atomic operations.

Efficiency: The dominating cost of an atomic_or is the same as a put.

The returned object will be a DeferredFromAttrs instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

async_map_atomic_xor(*space*, *key*, *value*)

An atomic_xor request stores the bitwise-xor of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: atomic_xor requests to the same key will be observed in the same order by all clients. All get requests which execute after a given atomic_xor request completes will see the results of that atomic_xor, or of a later operation which superseded it.

This operation is totally ordered with respect to all get, put, condput, delete and other atomic operations.

Efficiency: The dominating cost of an atomic_xor is the same as a put.

The returned object will be a DeferredFromAttrs instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

async_map_string_prepend(*space*, *key*, *value*)

A string_prepend request prepends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_prepend` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_prepend` request completes will see the results of that `string_prepend`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_prepend` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

async_map_string_append(*space*, *key*, *value*)

A `string_append` request appends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_append` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_append` request completes will see the results of that `string_append`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_append` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A nested dictionary of map keys to be changed. Each key in the outter dictionary sections a map attribute. Keys in the inner dictionaries correspond to map keys on which to operate. Values in the inner dictionary will be used to modify the specified map key, thus allowing several attributes to be changed in different ways. If the key does not exist in the map, it wil be created.

async_set_union(*space*, *key*, *value*)

A `set_intersect` performs set intersection, and stores the resulting set in the object. The intersection will be atomic without interference from other operations.

Consistency: `set_intersect` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_intersect` request completes will see the results of that `set_intersect`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_intersect` is the same as a `put`.

The returned object will be a `DeferredFromAttrs` instance which tracks the request.

space: A string naming the space in which the object will be inserted.

key: The key of the object. Keys may be either byte strings or integers.

attrs: A dictionary of attributes to be changed. Each value in the dictionary will be used to modify the specified attribute, thus allowing several attributes to be changed in different ways.

loop()
The `loop` call is the core event loop that receives and processes server responses. In the normal case, `loop` will return exactly one completed operation, or in the case of search, one item matching the search result.

It is imperative that the application periodically call `loop` in order to prevent unbounded resource consumption. Only in the `loop` function does the client library perform I/O which will clear the `recv` socket buffers.

The returned object will be a `Deferred` instance which tracks the request. The object will allow the user to immediately call `wait()` without blocking.

loop()
The `loop` call is the core event loop that receives and processes server responses. In the normal case, `loop` will return exactly one completed operation, or in the case of search, one item matching the search result.

It is imperative that the application periodically call `loop` in order to prevent unbounded resource consumption. Only in the `loop` function does the client library perform I/O which will clear the `recv` socket buffers.

The returned object will be a `Deferred` instance which tracks the request. The object will allow the user to immediately call `wait()` without blocking.

class hyperclient.**HyperClientException**(*status*, *attr*)
An exception that describes an error within the HyperClient library.

status()
A numeric error code indicating the issue. This will be one of the `HYPERCLIENT_*` error codes.

class hyperclient.**DeferredGet**

wait()
Wait for the operation to complete. On success, the returned object will be a `dict` mapping attribute names to their respective values. If the object does not exist, `None` will be returned. On error, a `HyperClientException` is thrown.

class hyperclient.**DeferredPut**

wait()
Wait for the operation to complete. The return value is a boolean indicating success. On error, a `HyperClientException` is thrown.

class hyperclient.**DeferredCondPut**

wait()
Wait for the operation to complete. The return value is a boolean indicating success. If the value is `True`, the comparison was successful, and the object was updated. If the value is `False`, the comparison failed, and the object remains unchanged.

class hyperclient.**DeferredCondPut**

wait()
Wait for the operation to complete. The return value is a boolean indicating success. If the value is `True`,

the delete was successful, and the object was removed. If the value is `False`, there was no object to remove.

class `hyperclient.`**`DeferredFromAttrs`**

`wait``()`
The return value is a boolean indicating success. If the value is `True`, the object exists, and all values were atomically changed. If the value is `False`, the object was not found.

class `hyperclient.`**`Search`**`(`*client*, *space*, *predicate*`)`

`__next__``()`
Return the next object or exception resulting from the search. Objects are Python dictionaries mapping attributes to their values.

CHAPTER

SEVEN

C/C++ API

The C API provides programmers with the most efficient way to access a HyperDex cluster. For this reason, the C API is used to implement scripting language APIs, and high performance applications. Where applicable, C++ names for functions are indicated. We ask that all custom benchmarks utilize the C API when other systems use native APIs as well.

7.1 Types

hyperclient
An opaque type containing all information about a single HyperDex client. An instance of `hyperclient` will maintain connections to the HyperDex coordinator, and HyperDex daemons.

This class exists in C++ as well.

See Also:

`hyperclient_create()`, `hyperclient_destroy()`

hyperdatatype
An enum indicating the type of a HyperDex attribute. Valid values are:

HYPERDATATYPE_STRING: The data is a byte string consisting of uninterpreted data.

HYPERDATATYPE_INT64: The data is interpreted as a signed, 64-bit little-endian integer. HyperDex will ignore trailing NULL bytes (they don't impact the integer's number), and will ignore any bytes beyond the first 8.

HYPERDATATYPE_GARBAGE: There was an internal error which leaves the API unable to interpret the type of an object. This indicates an internal error within the HyperDex client library which should be fixed.

The C++ API uses this type as-is.

hyperclient_attribute
A struct representing one attribute of an object. Each attribute corresponds to one dimension in the hyperspace. The members of this struct are:

const char* **hyperclient_attribute.attr**
A NULL-terminated C-string containing the human-readable name assigned to the attribute at subspace creation.

const char* **hyperclient_attribute.value**
The bytes to be used for the attribute. These bytes will be interpreted by HyperDex according to the `datatype` field. This field is not NULL-terminated because its length is defined in `value_sz`.

size_t **hyperclient_attribute.value_sz**
The number of bytes pointed to by `value`.

hyperclient_datatype **hyperclient_attribute.datatype**
The manner in which HyperDex should interpret the contents of `value`. This field must match the type specified for `attr` at the time the space was defined.

The C++ API uses this type as-is.

hyperclient_range_query
A struct representing one element of a range query. The members of this struct are:

const char* **hyperclient_range_query.attr**
A NULL-terminated C-string containing the human-readable name assigned to the attribute at subspace creation.

uint64_t **hyperclient_range_query.lower**
The lower bound on objects matching the range query. All objects which match the query will have `attr` be greater than or equal to `lower`.

uint64_t **hyperclient_range_query.upper**
The upper bound on objects matching the range query. All objects which match the query will have `attr` be strictly less than `upper`.

The C++ API uses this type as-is.

See Also:

`hyperclient_search()`

hyperclient_returncode
An enum indicating the result of an operation. Valid values are:

HYPERCLIENT_SUCCESS: The operation completed successfully. For operations which return objects, there is an object to be read.

HYPERCLIENT_NOTFOUND: The operation completed successfully, but there was no object to return or operate on.

HYPERCLIENT_SEARCHDONE: The indicated search operation has completed.

See Also:

`hyperclient_search()`

HYPERCLIENT_CMPFAIL: A conditional operation failed its comparison.

See Also:

`hyperclient_condput()`

HYPERCLIENT_UNKNOWNSPACE: The space specified does not exist.

HYPERCLIENT_COORDFAIL: The client library has lost contact with the coordinator.

HYPERCLIENT_SERVERERROR: A server has malfunctioned. This indicates the existence of a bug.

HYPERCLIENT_CONNECTFAIL: A connection to a server has failed.

HYPERCLIENT_DISCONNECT: A server has disconnected from the client library.

HYPERCLIENT_RECONFIGURE: The operation failed because the host contacted for the operation changed identities part way through the operation.

HYPERCLIENT_LOGICERROR: This indicates a bug in the client library.

HYPERCLIENT_TIMEOUT: The requested operation has exceeded its timeout, and is returning without completing any work.

See Also:

`hyperclient_loop()`

HYPERCLIENT_UNKNOWNATTR: The attribute is not one of the attributes which define the space.

See Also:

`hyperclient_attribute.attr`

HYPERCLIENT_DUPEATTR: An attribute is specified twice.

See Also:

`hyperclient_attribute.attr`

HYPERCLIENT_SEEERRNO: The `errno` variable will have more information about the failure.

HYPERCLIENT_NONEPENDING: Loop was called, but no operations are pending.

See Also:

`hyperclient_loop()`

HYPERCLIENT_DONTUSEKEY: The key was used as part of an equality search. The result of the search will be equivalent to a `GET`, so a get should be performed instead.

See Also:

`hyperclient_get()`, `hyperclient_search()`

HYPERCLIENT_WRONGTYPE: There was a type mismatch.

See Also:

`hyperclient_attribute.datatype`

HYPERCLIENT_EXCEPTION: The underlying C code threw an exception that was masked to prevent it from propagating up a C call stack. This indicates a bug in the client library.

HYPERCLIENT_ZERO: A value usable for testing for uninitialized variables.

HYPERCLIENT_A: A value usable for testing for uninitialized variables.

HYPERCLIENT_B: A value usable for testing for uninitialized variables.

The C++ API uses this type as-is.

7.2 Functions

hyperclient* **hyperclient_create** (const char* *coordinator*, in_port_t *port*)

Return a `hyperclient` instance. The caller is responsible for releasing all resources held by this instance by calling `hyperclient_destroy()`.

coordinator: A string containing the IP of the coordinator for the HyperDex cluster.

port: A number containing the port of the coordinator for the HyperDex cluster.

The C++ API provides a constructor for `hyperclient` which takes the same arguments and performs the same functionality.

See Also:

hyperclient, hyperclient_destroy()

void **hyperclient_destroy** (struct hyperclient* *client*)

Free all resources associated with the hyperclient pointed to by client.

client: The hyperclient instance to be freed.

The C++ API provides a destructor for hyperclient in place of this call.

See Also:

hyperclient, hyperclient_create()

int64_t **hyperclient_get** (struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, enum hyperclient_returncode* *status*, struct hyperclient_attribute** *attrs*, size_t* *attrs_sz*)

A get request retrieves an object from a specific space using the object's key. If the object exists in the space at the time of the request, it will be returned; otherwise, the API will indicate that the object was not found.

Consistency: A get request will always see the result of all complete put requests.

This operation is totally ordered with respect to all get, put, condput, delete and other atomic operations.

Efficiency: The dominating cost of a get is one network round trip for cached objects, and one disk read for uncached objects.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when hyperclient_loop() returns the same ID. If the integer returned is negative, it indicates an error generating the request, and *status contains the reason why.

client: An initialized hyperclient instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by key.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until hyperclient_loop() returns the same ID returned by this function.

attrs: A return value in which the object (if any) will be stored. This value will be changed if and only if *status is HYPERCLIENT_SUCCESS. If this function returns successfully, this pointer must remain valid until hyperclient_loop() returns the same ID returned by this function. This returned value must be released with a call to hyperclient_destroy_attrs().

See Also:

hyperclient_attribute, hyperclient_destroy_attrs()

attrs_sz: The number of hyperclient_attribute instances returned in attrs. This value will be changed if and only if *status is HYPERCLIENT_SUCCESS. If this function returns successfully, this pointer must remain valid until hyperclient_loop() returns the same ID returned by this function.

The C++ API provides hyperclient::get in addition to this call.

int64_t **hyperclient_put** (struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A put request creates or updates the attributes of an object stored in a specific space under a specific key. Only those attributes specified will be affected by the put operation. If the object does not exist, it will be created, and all specified attributes will be set to the values provided. All other attributes will be initialized to their default state.

Consistency: `put` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `put` request completes will see the result of that `put`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of a `put` is one network round trip per replica for cached objects, and one disk read per replica for uncached objects.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The attributes to be changed on the object. This pointer must remain valid for the duration of the call.

attrs_sz: THe number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::put` in place of this call.

int64_t **hyperclient_condput**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *condattrs*, size_t *condattrs_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A `condput` request creates or updates the attributes of an object stored in a specific space under a specific key if and only if certain attributes match the specified condition.. Only those attributes specified will be affected by the `condput` operation. If the object does not exist, the operation will fail. All unspecified attributes will be left to their current values.

Consistency: `condput` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `condput` request completes will see the result of that `condput`, or of a later operation which superseded it.

Efficiency: The dominating cost of a `condput` is the same as a `put` when the comparison succeeds. If the comparison fails, the cost is the same as a `get`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

condattrs: The attributes to be compared against. All attributes must match for the CONDITIONAL PUT to be successful. This pointer must remain valid for the duration of the call.

condattrs_sz: THe number of attributes pointed to by `condattrs`.

attrs: The attributes to be changed on the object. This pointer must remain valid for the duration of the call.

attrs_sz: THe number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::condput` in place of this call.

int64_t **`hyperclient_del`**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, enum hyperclient_returncode* *status*)

A `del` request removes an object identified by a key from the specified space. If the object does not exist, the operation will do nothing.

Consistency: `del` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `del` request completes will see the result of that `del`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of a `del` is one network round trip per replica for cached objects, and one disk read per replica for uncached objects.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::del` in place of this call.

int64_t **`hyperclient_atomic_add`**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_add` request adds the amount specified to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_add` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_add` request completes will see the results of that `atomic_add`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_add` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates

an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::atomic_add` in place of this call.

int64_t **hyperclient_atomic_sub**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_sub` request subtracts the amount specified from the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_sub` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_sub` request completes will see the results of that `atomic_sub`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_sub` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::atomic_sub` in place of this call.

int64_t **hyperclient_atomic_mul**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_mul` request multiplies the amount specified by the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mul` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mul` request completes will see the results of that `atomic_mul`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mul` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::atomic_mul` in place of this call.

int64_t **hyperclient_atomic_div**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_div` request divides the value of each attribute specified by the request by the amount specified. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_div` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_div` request completes will see the results of that `atomic_div`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_div` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::atomic_div` in place of this call.

int64_t **hyperclient_atomic_mod**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_mod` request computes the value of each attribute specified by the request modulo the amount specified. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mod` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mod` request completes will see the results of that `atomic_mod`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mod` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::atomic_mod` in place of this call.

int64_t **hyperclient_atomic_and**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_and` request stores the bitwise-and of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_and` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_and` request completes will see the results of that `atomic_and`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_and` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::atomic_and` in place of this call.

int64_t **hyperclient_atomic_or**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_or` request stores the bitwise-or of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_or` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_or` request completes will see the results of that `atomic_or`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_or` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::atomic_or` in place of this call.

int64_t **hyperclient_atomic_xor**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_xor` request stores the bitwise-xor of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_xor` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_xor` request completes will see the results of that `atomic_xor`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_xor` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::atomic_xor` in place of this call.

int64_t **hyperclient_string_prepend**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A `string_prepend` request prepends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_prepend` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_prepend` request completes will see the results of that `string_prepend`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_prepend` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::string_prepend` in place of this call.

int64_t **hyperclient_string_append**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A `string_append` request appends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_append` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_append` request completes will see the results of that `string_append`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_append` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::string_append` in place of this call.

int64_t **hyperclient_list_lpush**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A `list_lpush` request pushes the specified object to the head of the list of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `list_lpush` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `list_lpush` request completes will see the results of that `list_lpush`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `list_lpush` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::list_lpush` in place of this call.

int64_t **hyperclient_list_rpush**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A `list_rpush` request pushes the specified object to the tail of the list of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `list_rpush` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `list_rpush` request completes will see the results of that `list_rpush`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `list_rpush` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates

an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::list_rpush` in place of this call.

int64_t **hyperclient_set_add**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A `set_add` request adds a single element to a set or does nothing if the element is already in the set. The insertion will be atomic without interference from other operations.

Consistency: `set_add` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_add` request completes will see the results of that `set_add`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_add` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::set_add` in place of this call.

int64_t **hyperclient_set_remove**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A `set_remove` request removes a single element to a set, or does nothing if the element is not in the set. The removal will be atomic without interference from other operations.

Consistency: `set_remove` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_remove` request completes will see the results of that `set_remove`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_remove` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::set_remove` in place of this call.

int64_t **hyperclient_set_intersect**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A `set_intersect` performs set intersection, and stores the resulting set in the object. The intersection will be atomic without interference from other operations.

Consistency: `set_intersect` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_intersect` request completes will see the results of that `set_intersect`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_intersect` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::set_intersect` in place of this call.

int64_t **hyperclient_set_union**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A `set_union` performs set union, and stores the resulting set in the object. The union will be atomic without interference from other operations.

Consistency: `set_union` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `set_union` request completes will see the results of that `set_union`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `set_union` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each `hyperclient_attribute.value` field in attrs will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::set_union` in place of this call.

int64_t **hyperclient_map_atomic_add**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_map_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_add` request adds the amount specified to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_add` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_add` request completes will see the results of that `atomic_add`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_add` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each struct will indicate a map on which to operate with `hyperclient_attribute.attr` and the key within that map with `hyperclient_attribute.map_key`. The `hyperclient_attribute.value` field will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::map_atomic_add` in place of this call.

int64_t **hyperclient_map_atomic_sub**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_map_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_sub` request subtracts the amount specified from the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_sub` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_sub` request completes will see the results of that `atomic_sub`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_sub` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each struct will indicate a map on which to operate with `hyperclient_attribute.attr` and the key within that map with `hyperclient_attribute.map_key`. The `hyperclient_attribute.value` field will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::map_atomic_sub` in place of this call.

int64_t **hyperclient_map_atomic_mul** (struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_map_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_mul` request multiplies the amount specified by the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mul` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mul` request completes will see the results of that `atomic_mul`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mul` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each struct will indicate a map on which to operate with `hyperclient_attribute.attr` and the key within that map with `hyperclient_attribute.map_key`. The `hyperclient_attribute.value` field will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::map_atomic_mul` in place of this call.

int64_t **hyperclient_map_atomic_div** (struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_map_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_div` request divides the value of each attribute specified by the request by the amount specified.

The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_div` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_div` request completes will see the results of that `atomic_div`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_div` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each struct will indicate a map on which to operate with `hyperclient_attribute.attr` and the key within that map with `hyperclient_attribute.map_key`. The `hyperclient_attribute.value` field will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::map_atomic_div` in place of this call.

int64_t **hyperclient_map_atomic_mod**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_map_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_mod` request computes the value of each attribute specified by the request modulo the amount specified. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_mod` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_mod` request completes will see the results of that `atomic_mod`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_mod` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each struct will indicate a map on which to operate with `hyperclient_attribute.attr` and the key within that map with `hyperclient_attribute.map_key`. The `hyperclient_attribute.value` field will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::map_atomic_mod` in place of this call.

int64_t **hyperclient_map_atomic_and**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_map_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_and` request stores the bitwise-and of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_and` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_and` request completes will see the results of that `atomic_and`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_and` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each struct will indicate a map on which to operate with `hyperclient_attribute.attr` and the key within that map with `hyperclient_attribute.map_key`. The `hyperclient_attribute.value` field will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::map_atomic_and` in place of this call.

int64_t **hyperclient_map_atomic_or**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_map_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_or` request stores the bitwise-or of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_or` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_or` request completes will see the results of that `atomic_or`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_or` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each struct will indicate a map on which to operate with `hyperclient_attribute.attr` and the key within that map with `hyperclient_attribute.map_key`. The `hyperclient_attribute.value` field will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::map_atomic_or` in place of this call.

int64_t **hyperclient_map_atomic_xor**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_map_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

An `atomic_xor` request stores the bitwise-xor of the amount specified with the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `atomic_xor` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `atomic_xor` request completes will see the results of that `atomic_xor`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `atomic_xor` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each struct will indicate a map on which to operate with `hyperclient_attribute.attr` and the key within that map with `hyperclient_attribute.map_key`. The `hyperclient_attribute.value` field will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::map_atomic_xor` in place of this call.

int64_t **hyperclient_map_string_prepend**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_map_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A `string_prepend` request prepends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_prepend` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_prepend` request completes will see the results of that `string_prepend`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_prepend` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each struct will indicate a map on which to operate with `hyperclient_attribute.attr` and the key within that map with `hyperclient_attribute.map_key`. The `hyperclient_attribute.value` field will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::map_string_prepend` in place of this call.

int64_t **hyperclient_map_string_append**(struct hyperclient* *client*, const char* *space*, const char* *key*, size_t *key_sz*, const struct hyperclient_map_attribute* *attrs*, size_t *attrs_sz*, enum hyperclient_returncode* *status*)

A `string_append` request appends the specified string to the value of each attribute specified by the request. The attributes will be modified atomically without interference from other operations.

Consistency: `string_append` requests to the same key will be observed in the same order by all clients. All `get` requests which execute after a given `string_append` request completes will see the results of that `string_append`, or of a later operation which superseded it.

This operation is totally ordered with respect to all `get`, `put`, `condput`, `delete` and other atomic operations.

Efficiency: The dominating cost of an `string_append` is the same as a `put`.

On success, the integer returned will be a positive integer unique to this request. The request will be considered complete when `hyperclient_loop()` returns the same ID. If the integer returned is negative, it indicates an error generating the request, and `*status` contains the reason why. `HYPERCLIENT_UNKNOWNATTR`, `HYPERCLIENT_WRONGTYPE` and `HYPERCLIENT_DUPEATTR` indicate which attribute caused the error by returning `-1 - idx_of_bad_attr`.

client: An initialized `hyperclient` instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

key: A sequence of bytes to be used as the key. This pointer must remain valid for the duration of the call.

key_sz: The number of bytes pointed to by `key`.

attrs: The set of attributes to be changed. Each struct will indicate a map on which to operate with `hyperclient_attribute.attr` and the key within that map with `hyperclient_attribute.map_key`. The `hyperclient_attribute.value` field will be used to atomically modify the specified attribute, thus allowing several attributes to be changed in different ways. This pointer must remain valid for the duration of the call.

attrs_sz: The number of attributes pointed to by `attrs`.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until `hyperclient_loop()` returns the same ID returned by this function.

The C++ API provides `hyperclient::map_string_append` in place of this call.

int64_t **hyperclient_search**(struct hyperclient* *client*, const char* *space*, const struct hyperclient_attribute* *eq*, size_t *eq_sz*, const struct hyperclient_range_query* *rn*, size_t *rn_sz*, enum hyperclient_returncode* *status*, struct hyperclient_attribute** *attrs*, size_t* *attrs_sz*)

A `search` request retrieves all objects which match the specified predicate. These objects may reside on multiple nodes, and the client transparently handles retrieving them from each node. Multiple errors may be returned from a single search; however, the client library will continue to make progress until all servers finish the search or encounter an error.

Consistency: `search` results will see the result of every operation which completes prior to the search, unless there is a concurrent operation which supersedes the prior operation.

Efficiency: The dominating cost of a search is one network round trip per object. Multiple objects will be retrieved in parallel, so searches should be more efficient than simple object retrieval.

The search must match attributes specified by `eq` and `rn`. On success, the integer returned will be a positive integer unique to this request. One object or error is returned on each subsequent invocation of

hyperclient_loop() which returns the same ID. When hyperclient_loop() set *status to HYPERCLIENT_SEARCHDONE, the search is complete. If the integer returned is negative, it indicates an error generating the request, and *status contains the reason why. HYPERCLIENT_UNKNOWNATTR, HYPERCLIENT_WRONGTYPE and HYPERCLIENT_DUPEATTR indicate which attribute caused the error by returning -1 - idx_of_bad_attr, where idx_of_bad_attr is an index to into the combined attributes of eq and rn.

client: An initialized hyperclient instance.

space: A NULL-terminated C-string containing the name of the space to retrieve the object from.

eq: The attributes to which must match under a strict equality check. This pointer must remain valid for the duration of the call.

eq_sz: The number of attributes pointed to by eq.

rn: The attributes to which must fall within the range specified by hyperclient_range_query.lower and hyperclient_range_query.upper

See Also:

hyperclient_range_query

rn_sz: The number of attributes pointed to by rn.

status: A return value in which the result of the operation will be stored. If this function returns successfully, this pointer must remain valid until hyperclient_loop() returns the same ID returned by this function.

attrs: A return value in which the object (if any) will be stored. This value will be changed if and only if *status is HYPERCLIENT_SUCCESS. If this function returns successfully, this pointer must remain valid until hyperclient_loop() returns the same ID returned by this function with *status set to HYPERCLIENT_SEARCHDONE. This returned value must be released with a call to hyperclient_destroy_attrs().

This value will be overwritten each time hyperclient_loop() returns, and the value will not be saved. It is up to the caller of this function to either destroy the attributes, or preserve a pointer to them before calling hyperclient_loop() again.

See Also:

hyperclient_attribute, hyperclient_destroy_attrs()

attrs_sz: The number of hyperclient_attribute instances returned in attrs. This value will be changed if and only if *status is HYPERCLIENT_SUCCESS. If this function returns successfully, this pointer must remain valid until hyperclient_loop() returns the same ID returned by this function with *status set to HYPERCLIENT_SEARCHDONE.

The C++ API provides hyperclient::search in place of this call.

int64_t **hyperclient_loop** (struct hyperclient* *client*, int *timeout*, enum hyperclient_returncode* *status*)

The loop call is the core event loop that receives and processes server responses. In the normal case, loop will return exactly one completed operation, or in the case of search, one item matching the search result.

It is imperative that the application periodically call loop in order to prevent unbounded resource consumption. Only in the loop function does the client library perform I/O which will clear the recv socket buffers.

The return value is a 64-bit integer which identifies the outstanding operation that was processed. If an error is encountered or the event loop times out when processing the outstanding operations, the return value will be -1, and *status will be set to indicate the reason why.

client: An initialized hyperclient instance.

timeout: The number of milliseconds to wait before giving up on event processing.

status: A return value in which the result of the operation will be stored. This pointer must remain valid for the duration of the call.

The C++ API provides `hyperclient::loop` in place of this call.

void **hyperclient_destroy_attrs** (struct hyperclient_attribute* *attrs*, size_t *attrs_sz*)
Free an array of `hyperclient_attribute` objects returned via the `hyperclient` API. The returned values are allocated in a custom manner, and it is incorrect to free the memory using any other means.

7.3 Thread Safety

The HyperClient API uses no global or per-thread state. All state is enclosed in `hyperclient` instances. Concurrent access to the same `hyperclient` instance must be protected by external synchronization. The C++ API requires that all calls to members of `hyperclient` be protected with external synchronization.

CHAPTER

EIGHT

HYPERDEX-DAEMON MANUAL PAGE

8.1 Synopsis

hyperdex-daemon [*options*]

8.2 Description

hyperdex-daemon is a high performance data storage daemon. Each **hyperdex-daemon** is designed to be part of a larger cluster, and holds only a portion of the data stored in the cluster.

The daemon is designed to be as close to zero-conf as possible. The options shown below are the only parameters that a user must specify.

8.3 Options

-help
Show a help message

-d, -daemon
Run HyperDex in the background

-f, -foreground
Run HyperDex in the foreground

-D, -data=D
Directory to store all data and log files. It must exist. If no directory is specified, the current directory is used.

-h, -host=IP
IP address to use when connecting to the coordinator

-p, -port=P
Port to use when connecting to the coordinator

-t, -threads=N
Number of threads to create which will handle client requests. This should be equal to the number of cores for workloads which may be cached by main memory.

-b, -bind-to=IP
Local IP address on which to handle network requests. This must be addressable and unique among all nodes in the HyperDex cluster.

-i, -incoming-port=P
Port to listen on for incoming connections. If none is specified, a port is selected at random. It is best to let HyperDex select a random port.

-o, -outgoing-port=P
Port to use for all outgoing connections. If none is specified, a port is selected at random. It is best to let HyperDex select a random port.

8.4 See also

- *hyperdex-binary-test(1)*
- *hyperdex-coordinator(1)*
- *hyperdex-coordinator-control(1)*
- *hyperdex-daemon(1)*
- *hyperdex-replication-stress-test(1)*
- *hyperdex-simple-consistency-stress-test(1)*

CHAPTER

NINE

HYPERDEX-REPLICATION-STRESS-TEST MANUAL PAGE

9.1 Synopsis

hyperdex-replication-stress-test [*options*]

9.2 Description

hyperdex-replication-stress-test puts the replication code through its paces. There are certain potential race conditions that this code is designed to expose. To execute the test, create a space with the following description:

```
space replication
dimensions A, B, C
key A auto 4 2
subspace B auto 4 2
subspace C auto 4 2
```

The code will execute a number of operations in quick succession in a manner that bounces objects around the subspaces for `B` and `C`. This test relies upon every host in the cluster being deployed with a single network thread (`--threads=1`).

9.3 Options

-help
Show a help message.

-P, -prefix=number
The power of two used to indicate the size of the space.

-s, -space=space
The name of the space to operate on. By default "replication" is used.

-h, -host=IP
IP address to use when connecting to the coordinator

-p, -port=P
Port to use when connecting to the coordinator

9.4 See also

- *hyperdex-binary-test(1)*
- *hyperdex-coordinator(1)*
- *hyperdex-coordinator-control(1)*
- *hyperdex-daemon(1)*
- *hyperdex-replication-stress-test(1)*
- *hyperdex-simple-consistency-stress-test(1)*

CHAPTER

TEN

HYPERDEX-SIMPLE-CONSISTENCY-STRESS-TEST MANUAL PAGE

10.1 Synopsis

hyperdex-simple-consistency-stress-test [*options*]

10.2 Description

hyperdex-simple-consistency-stress-test puts the fault tolerance code its paces. This code is designed to operate on the following space:

```
space consistency
dimensions n, repetition
key n auto 0 2
```

The stress tester spawns a single writer thread which writes to keys in the range [0, window-size) in order with strictly increasing values for `repetition`. Multiple reader threads read in reverse order, looking out for a case in which the `repetition` value of successive keys goes backwards.

It is expected that failures will be generated externally (e.g. by forced process kills or dropped network connections) while running this test.

10.3 Options

-help
: Show a help message.

-w, -window-size=keys
: The number of sequential keys which will be used for the test. The smaller this number, the more likely it is to trigger an error condition. Use the largest value possible when hunting a bug.

-r, -repetitions=number
: The number of repetitions to run before exiting. Each key in [0, window-size) will be written this many times. A higher value runs for longer.

-t, --threads=number
The number of reader threads which will check for inconsistencies.

-s, --space=space
The name of the space to operate on. By default "consistency" is used.

-h, --host=IP
IP address to use when connecting to the coordinator

-p, --port=P
Port to use when connecting to the coordinator

10.4 See also

- *hyperdex-binary-test(1)*
- *hyperdex-coordinator(1)*
- *hyperdex-coordinator-control(1)*
- *hyperdex-daemon(1)*
- *hyperdex-replication-stress-test(1)*
- *hyperdex-simple-consistency-stress-test(1)*

PYTHON MODULE INDEX

INDEX

Symbols

A

C

D

G

H

L

M

P

S

W