HyperDex: A Distributed, Searchable Key-Value Store for Cloud Computing

Robert Escriva†, Bernard Wong‡, Emin Gün Sirer†
† Computer Science Department, Cornell University
‡ Cheriton School of Computer Science, University of Waterloo

Abstract
Distributed key-value stores are now a standard component of high-performance web services and cloud computing applications. While key-value stores offer significant performance and scalability advantages compared to traditional databases, they achieve these properties through a restricted API that limits object retrieval—i.e., an object can only be retrieved by the (primary and only) key under which it was inserted. This paper presents HyperDex, a novel distributed key-value store that provides a unique search primitive that enables queries on secondary attributes. The key insight behind HyperDex is the concept of hyperspace hashing in which objects with multiple attributes are mapped into a multi-dimensional hyperspace. This mapping leads to efficient implementations not only for retrieval by primary key, but also for partially-specified secondary attribute searches and range queries. A novel chaining protocol enables the system to provide strong consistency guarantees while supporting replication. An evaluation of the full system shows that HyperDex is orders of magnitude faster than Cassandra and MongoDB for finding partially specified objects. Additionally, HyperDex achieves high performance for simple get/put operations compared to current state-of-the-art key-value stores, with stronger fault-tolerance and comparable scalability properties.

1 Introduction
Modern cloud-deployed web services are increasingly based on lightweight key-value stores such as BigTable [13], Cassandra [26], and Dynamo [18]. While key-value stores can provide high performance, scalability and availability, they gain these properties through a restricted retrieval API. These systems only permit an object to be retrieved using the key under which it was stored. Queries based on secondary attributes are either not supported or implemented by enumerating all objects of a given type.

This paper introduces HyperDex, a high-performance, scalable, and distributed key-value store that provides a new search primitive for retrieving objects by secondary attributes. HyperDex achieves this extended functionality by organizing its data in a fundamentally different way. Similar to other key-value stores, HyperDex deterministically maps objects to nodes and enables efficient put and get operations for a key. However, in place of consistent hashing on just the key, HyperDex introduces hyperspace hashing in which objects are deterministically mapped to coordinates in a multi-dimensional Euclidean space (a hyperspace), where each dimension corresponds to an object attribute. HyperDex then creates an object to node mapping by tessellating the hyperspace into disjoint regions and assigning each region to a unique set of nodes. A region within the hyperspace along with its associated node assignment is called a zone. Every HyperDex node is responsible for storing all objects whose coordinates reside within the regions associated with each of the node’s zones. Clients map a search onto a subset of zones in the system, and map key-based operations to exactly one zone.

Hyperspace hashing facilitates efficient search by significantly reducing the number of servers to contact for each partially-specified search. In HyperDex, each specified exact match or range for an attribute value defines a region of hyperspace (a hyperplane) that encompasses the hyperspace locations of all matching objects. By construction of the hyperspace, all matching objects necessarily reside within the intersection region of these hyperplanes, which is itself a hyperplane smaller than or equal to the original hyperplane. Further, these objects can be located by contacting solely the nodes whose zones intersect the search hyperplane. Therefore, the set of matching objects can be retrieved by contacting a subset of the server nodes, instead of enumerating objects across all nodes. This yields a search complexity of $O(N^{(D-Q)/D})$, assuming $N \geq 2^D$, where $N$ is the number of server nodes in the key-value store, $D$ is the number of object attributes, and $Q$ is the number of attribute values specified in the query. Note that in the worst case, the exponent is 1, and all servers will be contacted. Fully-specified searches will need to contact exactly one node.

A naive Euclidean space construction, however, can suffer from the “curse of dimensionality,” as the space exhibits an exponential increase in volume with each additional secondary attribute [8]. 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 this problem 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.

Since failures are bound to occur in any large-scale cloud deployment, a key-value store needs to provide facilities for data replication as well as protocols for returning consistent responses in the presence of concurrency and failures. HyperDex ensures that all data in the system is replicated to provide a desired level of fault-tolerance. The system employs a value-dependent replication technique to maintain the desired level of fault tolerance while providing strong consistency guarantees. Value-dependent replication ensures a desired number of copies, provides flexibility in assigning replica nodes to different regions in hyperspace, and enables replica sets to be selected from non-fate-sharing nodes. Unlike fixed replication techniques, the replica sets are dynamically constructed on the fly depending on the contents of the object fields as well as the previous contents of the object, enabling efficient search operations and strong consistency semantics even in the presence of concurrent updates and crash failures.

Based on the structure provided by hyperspace hashing, HyperDex provides range query support for an object attribute by utilizing order-preserving hash functions for coordinate values. While users can provide custom hash functions, HyperDex provides a novel, fixed-size, lossy encoding function, called cfloat, that preserves the ordering between numeric values and lends itself naturally to an efficient encoding of position within the hyperspace. The returned search results are precise, even though cfloats and other hash functions used during the queries may be lossy.

Overall, this paper makes three contributions. First, it describes a new hashing technique for mapping structured data to server nodes. This hashing technique enables a novel search primitive that supports efficient object retrieval even when the search query requests multiple objects, specifies them through secondary attributes, and includes ranges. Second, it describes a replication scheme that enables hyperspace hashing to provide both fault-tolerance and consistency guarantees. The novel chaining used in the replication protocol guarantees a constant number of replicas to be maintained and ensures that operations using primary keys are strongly consistent. Finally, it reports from a full implementation of the system and deployment in a data center setting consisting of 64 servers, and demonstrates that HyperDex provides performance that is comparable to or better than Cassandra and MongoDB, two current state-of-the-art cloud storage systems, as measured using the industry-standard YCSB [16] benchmark. More importantly, HyperDex is qualitatively faster (specifically, 2x-13x on the YCSB benchmarks) than traditional key-value stores when searching for partially specified objects.

The rest of this paper is structured as follows: Section 2 describes hyperspace hashing. Section 3 specifies how to deal with spaces of high dimensionality through data partitioning. Section 4 specifies the value-dependent replication protocol used in HyperDex. Section 5 outlines our full implementation of HyperDex. Section 6 evaluates HyperDex under a variety of workloads. Section 7 discusses related work for hyperspace hashing and HyperDex. Section 8 summarizes our contributions.

2 Approach

In this section, we describe the data model used in HyperDex, outline hyperspace hashing, and sketch the high-level organization and operation of the system.

2.1 Data Model and API

HyperDex stores objects that consist of a key and zero or more secondary attributes. Objects are organized into distributed, persistent tables, where a table is a collection of objects with the same attributes with distinct keys. This data organization closely resembles tables in standard relational databases and offers a strict super-set of the typical key-value store operations, which permits straightforward migration from existing key-value stores and database systems.

Object-level operations in HyperDex operate on semi-

| get(t, k) | Retrieve an object from table t using the key k |
| put(t, ⟨k, v₁, ..., v₉⟩) | Insert an object into the table t |
| del(t, k) | Remove an object from table t using the key k |
| cas(t, ⟨k, v₁, ..., v₉⟩, r) | Perform a put using the first two arguments if and only if the previous version of the object is r. |
| search(t, P) | Find all objects from table t which match predicate P. |

Table 1: The HyperDex object API. The get, put, del, and cas operations are typical of key-value stores. HyperDex is unique in that it efficiently supports the search operation. Each predicate $P = ⟨c₁ = v₁, c₂ = v₂, ..., c₉ = v₉⟩$ where attributes $c₁, c₂, ..., c₉$ match the values $v₁, v₂, ..., v₉$. Attribute values $v_i$ in the predicate may consist of ranges for concision.
structured data. Objects consist of \( k \)-tuples stored under a key. Keys and \( k \)-tuple elements may be interpreted as opaque byte strings or numeric values. A summary of all object-level operations is provided in Table 1.

System-level operations in HyperDex provide the user with complete control over table creation, deletion, and renaming. These operations require global coordination and are relatively expensive compared to manipulating objects. System-level operations impact only operations on the same table. Other operations are unaffected. For maximum flexibility, the system-level operations afford the user complete control over the key, secondary attributes, partitioning scheme and replication factor.

2.2 Hyperspace Hashing

HyperDex represents each table as an independent multi-dimensional space, where the dimensional axes are the attributes of the table. Consider, for the following discussion, a table containing user information that has the attributes first-name, last-name, and telephone-number. HyperDex creates a three dimensional space where the first-name attribute constitutes the x-axis, the last-name attribute constitutes the y-axis, and the telephone-number attribute constitutes the z-axis. HyperDex assigns every object a corresponding coordinate based on the object's attribute values. An object is mapped to a deterministic coordinate in this space by hashing each of its attribute values to a location along the corresponding axis. Figure 1 illustrates this mapping.

2.3 Node Layout

Efficient lookup of fully-specified objects is critical to object insertion and deletion performance, and requires a deterministic object to node mapping. Much like in ring-based key-value stores[3, 18, 40], HyperDex maps both object coordinates and nodes to the same hyperspace. Specifically, HyperDex tessellates the hyperspace into a grid of \( N \) regions in space. Zones, which we previously defined to be mutually exclusive regions belonging to one or more nodes, are created by assigning nodes to each of the tessellated regions to be responsible for all objects which hash to a coordinate within the region. The zone mapping is disseminated to all clients which may operate directly on the mapping without any routing between server nodes.

With this node and data layout, insertion and deletion of an object in the most basic HyperDex implementation consists of determining the coordinates for the object, and storing or deleting the object in the zone which contains that coordinate. We will explore the additional complexities related to insertion and deletion operations that support data partitioning in Section 3 and object replication in Section 4.

2.4 Search Queries

The data and node organization described in the preceding sections facilitates a geometric approach to resolving search queries. HyperDex maps a search query consisting of a (potentially incomplete) set of attribute values to a corresponding search hyperplane within the hyperspace. For example, a search query specifying \( Q \) secondary attributes defines a \((D - Q)\)-dimensional search hyperplane. This search hyperplane is comprised of the intersection of \( Q \) individual \((D - 1)\)-dimensional hyperplanes, where each \((D - 1)\)-dimensional hyperplane corresponds to one attribute specified in the search. Each individual attribute's hyperplane intercepts the attribute's axis at the location to which the value hashes. The search hyperplane encompasses the location of all objects that match the search query.

Range queries correspond to extruded hyperplanes. The search hyperplane for a single range query attribute corresponds to the union of all hyperplanes that intersect the axis between the lower and upper bounds of the range. Note that for such a scheme to work, the attribute must be numeric in nature, and the axis must preserve order of the attribute. Conceptually, HyperDex executes range queries by taking the union of all hyperplanes that result from intersecting a single hyperplane.
from the range query with the remainder of the search. In reality, such a scheme is computationally intensive and intended only to give the geometric intuition behind how range queries are handled in HyperDex. Section 5.2 discusses how to implement range queries efficiently.

The hyperspace and query constructions are easy to visualize. Figure 1 shows how a query for `first_name='John'` and `last_name='Smith'` corresponds to a line in the three-dimensional space. The query for `first_name='John'` forms a two-dimensional plane which intercepts the first_name axis at hashof('John'). Similarly, the query for `last_name='Smith'` creates another plane which intersects the last_name axis. The intersection of the two planes is the line along which all phone numbers for John Smith reside. Each phone number is a single point in space along this line. A range query, perhaps restricting a search to a particular area code, could be used to further restrict the phone number to just a segment of this line.

The set of hosts which may store objects matching the search query consists of those hosts assigned to a zone which intersects the search hyperplane. This mapping is established by a coordinator and made known to all clients, who can independently compute this intersection. Query resolution concludes by collecting matching objects from one node in each zone. The hyperspace construction rewards specificity in the search query. The more attributes a query specifies, the fewer the number of dimensions that the search hyperplane spans, which, in turn, reduces the number of zones which intersect the search hyperplane.

3 Data Partitioning

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, the drawback of coupling the dimensionality of hyperspace with the number of searchable attributes is that, for tables with many searchable attributes, the hyperspace can become very large, as volume grows exponentially with the number of dimensions. Covering such large spaces with a regular grid of nodes may not be feasible even for large data-center deployments. For example, consider a D-dimensional hyperspace. If the hyperspace is split into two zones in each dimension, the resulting tessellation will have $2^D$ distinct zones. If $N$ machines are deployed to cover this hyperspace, the machines will cover the regions for $\lg N$ of these dimensions. Searches which specify more than $\lg N$ dimensions would have to contact all nodes with high probability even if a search is highly specific. Note that having at least $2^D$ distinct zones is desirable, as it enables each attribute to contribute to the search prun-

![Figure 2: HyperDex distributes the original many-dimensional hyperspace across multiple fewer-dimensional subspaces. Each subspace contains only the item attributes that correspond with the dimensions it covers.](image)

HyperDex avoids the problems associated with high-dimensionality by partitioning tables with many attributes into multiple lower-dimensional hyperspaces. Each of these subspaces uses a subset of object attributes as the dimensional axes for an independent hyperspace. Figure 2 shows how HyperDex can represent a table with $D$ searchable attributes as a set of subspaces $s$, where each subspace $s_i \in s$ contains $D_i$ dimensions.

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.

Data partitioning forces a trade-off between search generality and efficiency. On the one hand, a single hyperspace can accommodate arbitrary searches over its associated attributes. On the other hand, a hyperspace which is too large will always require that partially-specified queries contact more than one zone. Applications often have locality in the attributes that are searched over. By creating subspaces which exploit this locality, HyperDex applications can tune search efficiency. As the number of subspaces grows, so, too, do the costs associated with maintaining data consistency across subspaces. Updates to objects can change the objects’ locations in only a subset of the subspaces. Section 4 details how Hy-
perDex does this efficiently with a predictably low overhead.

3.1 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 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.

3.2 Object Distribution Over Subspaces

Subspace partitioning exposes a design choice in how objects are distributed and stored on server nodes. One possible design choice is to keep data in normalized form, where every subspace retains, for each object, only those object attributes that serve as the subspace’s dimensional axes. While this approach minimizes storage requirements per node, as attributes are not duplicated across subspaces, it leads to more expensive search and object retrieval operations as reconstituting the object requires a multi-node join operation. In contrast, an alternative design choice is to store a full copy of each object in each subspace, which leads to faster search and retrieval operations at the expense of additional storage requirements per node.

Hyperspace hashing supports both of these object distribution techniques. HyperDex, however, relies upon the later approach to implement the replication scheme described in Section 4.

3.3 Heterogeneous Objects

In a real deployment, the key-value store will likely be used to hold disparate objects with different schema. HyperDex supports this through the table abstraction. Each table has a separate set of attributes which make up the objects within, and these attributes are partitioned into subspaces independent of all other tables. As a result, HyperDex manages multiple independent hyperspaces.

4 Consistency and Replication

Because hyperspace hashing maps each object to multiple nodes, maintaining a consistent view of the object poses a challenge. HyperDex employs a novel technique called value-dependent chaining to provide strong consistency in the presence of concurrent updates.

For clarity, we first describe value-dependent chaining without concern for fault tolerance. Under this scheme, a single failure leaves portions of the hyperspace unavailable for updates and searches. We then describe how value-dependent chaining can be extended to provide fault tolerance such that the system can tolerate up to $f$ failures in any one zone.

4.1 Value Dependent Chaining

Because hyperspace hashing determines the location of an object by its contents, and subspace partitioning creates many object replicas, objects will be spread across multiple nodes and these nodes will change as the objects are updated. Change in an object’s location would cause problems if implemented naively. For example, if object updates were to be implemented by simply sending the object to all affected zones, there would be no guarantees associated with subsequent operations on that object. Such a scheme would at best provide eventual consistency because nodes may receive updates out-of-order, with no sensible means of resolving concurrent updates.

HyperDex orders updates by arranging an object’s replicas into a value-dependent chain whose members are deterministically chosen based upon an object’s hyperspace hash values. The head of the chain is called the point leader, and is determined by hashing the object in the key subspace. Subsequent nodes in the chain are determined by hashing attribute values for each of the remaining subspaces.

This construction of value-dependent chains enables efficient, deterministic propagation of updates. The point leader for an object is in a position to dictate the total order on all updates to that object. Each update flows from the point leader through the chain, and is considered pending until an acknowledgement of that update is received from the next node in the chain. When an update reaches the tail of the chain, the tail sends an acknowledgement through the chain in reverse so that all other nodes may clean up replication-related state. When the acknowledgement reaches the point leader, the client is notified that the operation is complete. Sending the client notification from the point leader enables us to provide clients with a single point of contact for
Figure 3: HyperDex’s replication protocol propagates updates from the key subspace through additional subspaces along value-dependent chains. This example depicts a table that is partitioned into two subspaces in addition to the mandatory key subspace. Each chain passes through exactly one or two nodes in each subspace. Updates flow forward along chains as indicated by the arrows, while acknowledgements flow in reverse.

Updates are more complicated because a change in an attribute value might require relocating an object to a different zone. HyperDex accomplishes this by constructing chains where each zone corresponding to the old version of the object immediately precedes the zone corresponding to the new version of the object within the same subspace. This property ensures that the update will take effect in every zone to which the old and new objects map. Further, the old version will only be deleted after the new version is committed, because the operations commit in reverse order as acknowledgements flow in reverse. But because of the dependency information, $h_2$ will be able to route it through the chain in the correct order by hashing the previous versions.

The interplay between destructive operations and updates poses an even greater challenge for state management. Each HyperDex zone retransmits messages until they are acknowledged. It would be possible to trigger a race condition with a naive implementation if the retransmission arrives at an inopportune time after the destructive operation is processed. Consider what happens if update $u_3$ in Figure 3 were a client initiated `delete` operation. As acknowledgements for $u_4$ propagate backwards along the value-dependent chain for $u_3$, they would remove any and all trace associated with $u_3$. If the acknowledgement of $u_2$ from $h_5$ to $h_2$ were lost, the next retransmission of $u_2$ from $h_2$ to $h_5$ would be handled the exact same way it was handled the first time. That is, $u_2$ would contain no dependency information for $h_5$, and would therefore be applied immediately. This would then lead to inconsistencies. Solutions using sequence numbers are possible; however, managing these sequence numbers adds increasing cost, as they must be maintained through the entire lifetime of the system – even as system membership changes. HyperDex avoids interplay between destructive operations and updates by having members which reside in both chains defer operations until they will no longer interact with the destructive operations. In the example given above, $u_3$ would have a version number of 3 and depend upon update $u_2$ with version number 2 and type `put`. When a host receives an update which has an unmet dependency, it defers the update until such a time that the dependency is met and the update may be acted upon. By construction of value-dependent chains, hosts are guaranteed to receive the dependencies in time bounded by the length of chains and number of failures.

Key-based operations, and obviates any need to maintain state about clients across nodes. In Figure 3, the update $u_1$ illustrates an object insertion which passes through $h_1, h_2, h_3$, where $h_1$ is the point leader.

Successive updates to an object will construct chains which overlap in each subspace. Concurrent updates may arrive out of order at each of these points of overlap. For example, consider the update $u_3$ in Figure 3. The value-dependent chain for this update is $h_1, h_2, h_5, h_3$. Notice that it is possible for $u_3$ to arrive at $h_5$ before $u_2$. HyperDex efficiently handles this case by dictating that the point leader embed in each message dependency information which specifies the order in which updates are applied. Specifically, the point leader embeds a version number for the update, and the version number and type of the previous update. For instance, $u_3$ will have a version number of 3 and depend upon update $u_2$ with version number 2 and type `put`. When a host receives an update which has an unmet dependency, it defers the update until such a time that the dependency is met and the update may be acted upon. By construction of value-dependent chains, hosts are guaranteed to receive the dependencies in time bounded by the length of chains and number of failures.

Embedding dependency information – specifically a key, version number, and hash of the previous object – in operations enables HyperDex to avoid complex corner cases stemming from the interaction of value dependent chains with destructive operations. For example, consider the case of an update $u_4$ which follows update $u_3$ in Figure 3 and maps the object to $h_1, h_2, h_6$. The value-dependent chain for $u_4$ is $h_1, h_5, h_2, h_3, h_6$. When $h_2$ receives $u_4$, it will have no knowledge of any of the preceding updates as $u_3$ will have relocated the object to $h_5$. But because of the dependency information, $h_2$ will be able to route it through the chain in the correct order by hashing the previous versions.
4.2 Fault Tolerance

To guard against node failures, HyperDex provides additional replication within each zone. The replicas acquire and maintain their state by being incorporated into value-dependent chains. In particular, each zone has $R$ replicas which appear as a block in the value-dependent chain. For example, we can extend the layout of Figure 3 to tolerate one failure by introducing additional hosts $h'_j$ through $h''_j$. As with regular chain replication [41], new replicas are introduced at the tail, and nodes are bumped forward in the chain as other nodes fail. For example, the first update in Figure 3 has the value-dependent chain $h_1, h'_1, h_2, h'_2, h_3, h_5$. If $h_2$ were to fail, the resulting chain would be $h_1, h'_1, h'_2, h'_3, h_3, h_5$. HyperDex ensures that such a transition will be done without compromising strong consistency.

HyperDex ensures that point leader failures do not allow clients to observe an inconsistency. For instance, if $h_1$, the point leader, were to fail in our previous example, $h'_1$ will take over the role of point leader. The client will detect the point leader failure, and notify the application of the reconfiguration. Operations which are interrupted by reconfiguration exhibit at-most-once semantics.

4.3 Node and Configuration Management

HyperDex utilizes a logically centralized coordinator to manage system-wide global state. The coordinator declare the global state in the form of a configuration. Each configuration encapsulates all aspects of the HyperDex state, including the way in which the coordinator tessellates the hyperspace, the mapping between zones and hosts, and which zones are currently undergoing failure recovery. The coordinator is responsible for generating and distributing configurations.

Each host in the system is uniquely identified by its IP address, port, and epoch id. When a new HyperDex process is spawned, the process requests an epoch id from the coordinator. The combination of the address and the epoch id uniquely identify the process with which one is communicating (an instance of a HyperDex server) for the entire lifetime of the HyperDex cluster. In the event of a failure, a host may relaunch HyperDex on the same IP/port; however, the epoch id received from the coordinator will be different, and all other processes can distinguish between the two instances.

The coordinator maintains a mapping between each zone and an ordered set of replicas. Messages are the primitive of communication within HyperDex and are logically addressed using the zone and replica identifier. The receiver of a message acts upon the message only when the source and destination logical addresses match the source and destination instances. This mechanism ensures that each host will act only upon messages which are consistent with the host’s current configuration.

Each host changes its configuration in an all-or-nothing fashion which appears instantaneous to threads handling network communication. This is accomplished on each host by creating state relevant to the new configuration, pausing network traffic, swapping pointers to make the new state visible, and unpausing network traffic. This pause period is quick in practice and completes in sub-millisecond time on each host.

4.4 Consistency Guarantees

Overall, the preceding protocol ensures that HyperDex provides useful guarantees for applications. The specific guarantees made by HyperDex are:

**Key Consistency** All actions which operate on a specific key (e.g., get and put) are linearizable [21] with all operations on all keys. This guarantees that all clients of HyperDex will observe updates in the same order.

**Search Consistency** HyperDex guarantees that a search which is not concurrent with updates that affect the search will return the latest version of all items. The searches are not strongly consistent with concurrently modified objects because there is a small window of time during which a client may observe inconsistency. In Figure 3, this window of vulnerability occurs if a search is concurrent with $u_2$ and is processed at $h_5$ before $u_2$ is committed, but is processed at $h_2$ after $u_2$ is committed.

Overall, HyperDex trades a slight reduction in consistency guarantees for increased search performance, without compromising strong guarantees for key-based operations.

5 Implementation

HyperDex is a fully implemented system which supports all features described by this paper. The implementation is approximately 39,750 lines of C++. The HyperDex software distribution contains an embeddable storage layer, a reusable hyperspace hashing library, the HyperDex server, the client library, and a Python-based coordinator.

In this section, we focus on three critical pieces of HyperDex’s design that make the system perform well in practice. First, we describe how to reapply the principles of hyperspace hashing at the storage layer in our HyperDisk subsystem. Second, we show how to implement hyperspace hashing efficiently using integer operations. Third, we discuss how we employ a distributed coordinator to eliminate single points of failure.

5.1 HyperDisk: On-Disk Data Storage

HyperDisk provides persistent storage using hyperspace hashing by mapping objects to files on disk called sectors. Each sector is a fixed-size, log-structured file which is mapped into memory into which objects are inserted. HyperDisk tessellates a node’s assigned zone into sec-
HyperDisk applies hyperspace hashing to map objects to sectors. Every object stored in a sector matches both the key and all attributes that define the sector. When a sector’s log becomes full, HyperDisk re-tessellates the hyperspace by splitting a full sector into four non-overlapping sectors of equal size. Figure 4 shows an example HyperDisk. A key operation accesses only those sectors residing on a line perpendicular to the y-axis. Similarly, a fully-specified search involves only those sectors residing on a line perpendicular to the x-axis.

This division of a zone into sectors enables a HyperDex node to efficiently insert and remove objects as well as take snapshots for search operations. For example, an insertion into the HyperDisk shown in Figure 4 under the key \( k_0 \) would only affect the single sector \( \beta \). A lookup for the same key would require three sector reads from \( \alpha \), \( \beta \) and \( \gamma \). A hash table stored in the sector header, and the append-only structure of the sector facilitate easy lookup of an object within the sector without having to read the whole sector. A search, shown with the vertical dashed line in the figure, would access sectors \( \beta, \delta, \zeta, \eta \).

Concurrency between key operations and searches requires careful handling of object enumeration over a HyperDisk. In a naive implementation, a concurrent update could cause an object relocated between sectors to appear twice or not at all in search results. HyperDex avoids this by taking snapshots of the HyperDisk at the time the search is issued. Such snapshots can be performed efficiently by noting the end of the each sector’s log and ensuring that no sector in the snapshot is deleted until the search operation is complete.

HyperDisk provides the same consistency guarantees as the HyperDex system. The header information of each sector may be mapped into memory to ensure fast lookup. Our implementation uses fine-grained locking to ensure that key-based operations and snapshots are linearizable. Note that HyperDisk is not a full database library and does not handle failure recovery. This does not pose a problem for fault tolerance because HyperDex relies upon recovery mechanisms at the granularity of servers and not at the granularity of individual disks.

Garbage collection of old sectors is simple and inexpensive. When a sector splits, and is no longer necessary, HyperDisk unlinks the old sector. The reference counting mechanisms within HyperDisk and the filesystem ensure that the sector will remain available until it is no longer referenced by any snapshot. At that time, HyperDisk unmaps the sector from memory and the filesystem reclaims the underlying storage.

Overall, HyperDisk applies the hyperspace hashing approach to the data stored within a HyperDex node to achieve, in the average case, \( O(\sqrt{S}) \) disk accesses per operation, where \( S \) is the number of sectors.

5.2 Hyperspace Hashing Functions

HyperDex provides a hash library for strings and numbers that is efficient, compact, and accurate. The goal of this library is to efficiently support hyperspace operations through bitwise manipulation of fixed-size numbers.

HyperDex creates a hash of an object by mixing the bits from the object’s attributes into a single conglomerate hash. This interleaved representation ensures that if two objects’ hashes are different, the hashes by at least one bit for each differing attribute. An object matches a search if and only if its hash, when masked to extract the bits corresponding to the search attributes, matches the hash of the search terms. Figure 5 shows how an object with three attributes is mapped to a single hash.

![Figure 5: The conglomerate hash of an object with three attributes.](image)

The hash of an object likely will not contain every bit from each input’s respective hashes. This poses a
problem for numeric attributes, since they must preserve order for range queries. Applications may wish to customize the hash for numeric attributes to ensure they are uniformly distributed. One hashing technique that we have found works well is a novel representation called \( cfloat \) in which we encode the number in floating point format using only the number of bits which will appear in the conglomerate hash. Such an encoding preserves order, while allowing a large range of numbers to be encoded.

Hyperspace hashing is used to map objects to zones or sectors by matching partial patterns against the objects’ hashes. This is quick in practice, and consists of a mask and compare operation. By construction of the hyperspace, HyperDex ensures that there are no overlapping zones. It therefore follows that the first zone which matches an object’s hash is the zone to which the object must map. Sectors, in contrast, must check every sector until the object is found.

HyperDex application developers may replace these hash functions with hash functions more-suited to the application’s need. In particular, numeric hash functions which map non-uniform inputs to uniform outputs are desirable, and entirely application dependent.

5.3 Distributed Coordination

Zone assignment and management in HyperDex is performed by a logically centralized coordinator. Since a physically centralized coordinator would limit scalability and be a single point of failure [29], the HyperDex coordinator is distributed and makes use of a coordination service [11, 22] to scale the coordinator beyond a single physical server. The coordinator implementation ensures that servers may migrate between coordinators so that no coordinator failure leads to a partition in the system. The coordinator directs all failure recovery actions. Servers may report observed failures to the coordinator, or the coordinator may directly observe failures through periodic failure detection pings to servers. Epoch ids provide long-term protection against communicating with failed nodes. As an optimization, the coordinator may forcibly isolate failed components – by killing the VM or shutting off the appropriate switch port – and directly roll out new configurations. If this is not possible, the coordinator must first issue a configuration which invalidates the failed epoch id, and then issue a second configuration which replaces the failed node.

6 Evaluation

We deployed HyperDex on both a small and medium-size computational cluster and evaluated the performance of each deployment using the Yahoo! Cloud Serving Benchmark (YCSB) [16], an industry-standard tool for evaluating cloud storage performance. Our evaluation also closely examines the performance of HyperDex’s basic operations, specifically, \texttt{get}, \texttt{put}, and \texttt{search}, using targeted micro-benchmarks. These micro-benchmarks enable us to isolate and exercise specific components in our system and help expose the performance impact of each of our design decisions. For both YCSB and the micro-benchmarks, we compare HyperDex with Cassandra [26], a popular key-value store for Web 2.0 applications, and MongoDB [1], a distributed document database.

The performance benchmarks are executed on our small, dedicated lab-size cluster in order to avoid any confounding issues arising from sharing a virtualized platform, while the scalability benchmarks are executed on the VICCI [33] testbed. Our dedicated cluster consists of fourteen nodes, each of which is equipped with two Intel Xeon 2.5 GHz E5420 processors, 16 GB of RAM, and a 500 GB SATA 3.0 Gbit/s hard disk operating at 7200 RPM. All nodes are running 64-bit Debian 6 with the Linux 2.6.32 kernel. A single gigabit Ethernet switch connects all fourteen machines. On each of the machines, we deployed Cassandra version 0.7.3, MongoDB version 2.0.0, and HyperDex.

For all lab-scale tests, the storage systems are configured to provide sufficient replication to tolerate one node failure. Each system was configured to use its default consistency settings. Specifically, both Cassandra and MongoDB perform replication asynchronously; update confirmations are returned to the clients before the updates are consistently applied across the replicas. In contrast, HyperDex utilizes value-depending chaining and, as a result, always provides clients with strong consistency even in the presence of node failures. MongoDB allocates replicates in pairs, necessitating an even cluster size. To that end, we allocate twelve machines for the storage nodes; one machine for the clients; and, where applicable, one node for the coordinator. HyperDex is configured to have two subspaces in addition to the key subspace to accommodate the ten attributes in the YCSB dataset.

6.1 Get/Put Performance

High get/put performance is paramount to any cloud-based storage systems, and YCSB provides an industry-standard way to evaluate get/put performance. It specifies six different workloads that exercise the storage system with a mixture of request types and object distributions resembling real-world applications. In our experimental setup, each request specifies one or more objects, out of a total of 10,000,000, that are loaded into the system prior to the experiment. Figure 6 shows the throughput achieved by each system across the YCSB workloads. HyperDex provides a throughput that is between a factor of two to thirteen higher than the other
systems. This significant improvement in get/put performance is attributable mostly to the efficient handling of get operations under hyperspace hashing. Our implementation demonstrates that the hyperspace construction and maintenance can be realized efficiently.

In order to gain insights into the performance of the system, we examined the request latency distributions of the different systems under workload B, representing read-intensive workloads with 5% writes and 95% reads, and workload A, representing read-write balanced workloads with 50% reads and writes. Both workloads select keys from a Zipf distribution. Figure 7 shows the request latency distributions of the three systems under workload B. We see that HyperDex completes 100% of its get operations in 1 ms, which is several times faster than Cassandra and MongoDB. The latency distribution shown for all three systems is qualitatively similar for workloads A, B, C, and D.

For completeness, we present the performance of all three systems on a write-heavy workload. Figure 8 shows the latency distribution for loading 10,000,000 objects into the database in preparation for the YCSB tests. This test illustrates the effect that consistency guarantees have on put latency. MongoDB’s default consistency guarantees consider a put complete when it has been successfully sent to the destination server with no errors. Cassandra’s default consistency guarantees consider a put complete when it is queued in the filesystem cache on at least one replica. In contrast, HyperDex guarantees that the update is committed on all replicas before returning a response to the client. As a result, MongoDB is able to complete a majority of operations in less than 1 ms. Nevertheless, 99% of HyperDex operations complete in less than 2 ms.

6.2 Search vs. Scan

Unlike existing key-value stores, HyperDex is architected from the ground-up to perform search operations efficiently. Current applications that rely on existing key-value stores emulate search functionality by embedding additional information about other attributes. For example, applications typically group logically related objects by using a shared prefix in the key of each object, and then rely upon the key-value store to locate keys with the common prefix. In Google BigTable and Cassandra, this operation is efficient because keys are stored in sorted order, and returning all logically grouped keys is an efficient linear scan. Fittingly, the YCSB benchmark calls this a scan operation. HyperDex’s search functionality is a strict superset of the scan operation. Rather than using a shared prefix to support
scans, HyperDex stores, for each object, the prefix and suffix of the key as two secondary attributes. Scans are then implemented as a multi-attribute search that exactly matches a provided prefix value and a provided range of suffix values. Thus, all YCSB benchmarks involving a scan operation operate on secondary attributes in HyperDex, but operate on the key for other systems.

Despite operating on secondary attributes instead of the key, HyperDex outperforms the other systems by an order of magnitude for scan operations (Figure 9). Seventy five percent of search operations complete in less than 2 ms, and nearly all complete in less than 6 ms. Cassandra clusters data in a sorted order based on the primary key, and is therefore able to retrieve matching items relatively quickly. MongoDB’s sharding requires construction of an index; consequently, scan operations in MongoDB are relatively fast. The search performance of HyperDex is not attributable to our efficient implementation as the amount of improvement eclipses the differences observed for get/put latency and throughput. Hyperspace hashing in HyperDex clusters the matching results on a small number of servers; this enables effective pruning of the search space and explains enables each search to complete by contacting exactly one host. Furthermore, the HyperDisk structure enables each host to efficiently compute the search results.

An additional benefit of HyperDex’s aggressive search space pruning is the relatively low latency overhead associated with search operations. Figure 10 shows the average latency of a single scan operation as the total number of scan operations performed increases. In this test, searches were constructed by choosing the lower bound of the range uniformly at random from the set of possible values, as opposed to workload E which uses a Zipf distribution to. The uniform distribution of searches evenly distributes load in all three systems. HyperDex consistently offers low latency even in the presence of a search heavy workload.

A critical parameter that affects HyperDex’s search performance is the number of subspaces to create for a HyperDex table. Increasing the number of subspaces will lead to additional opportunities for pruning the search space for search operations, but will simultaneously require longer value-dependent chains that can result in higher put latencies. In Figure 11, we explore the tradeoff using between zero and ten additional subspaces beyond the mandatory key subspace. As expected, HyperDex’s put latency increases and throughput decreases with additional subspaces; latency exhibits a linear trend. Because the number of clients is fixed, throughput falls significantly with the first few additional subspaces, but eventually reaches a point where additional subspaces impact throughput less.

6.3 Scalability

We have deployed HyperDex on the VICCI [33] testbed to evaluate its performance in an environment representative of the cloud. Each VICCI cluster has 70 Dell R410 PowerEdge servers, each of which has 2 Intel Xeon X5650 CPUs, 48 GB of RAM, three 1 TB hard drives, and two 1 Gbit ethernet ports. Users are provided with an isolated virtual machine in which they may conduct experiments. Each virtual machine comes preinstalled with Fedora 12 and runs the 2.6.32 Linux kernel.

We examined the performance of a HyperDex cluster as the cluster increases in size. Increasing the number of servers in the cluster provides HyperDex with additional resources, and leads to a proportional increase in
throughput. In Figure 12, we explore the system throughput for a variety of configurations. HyperDex scales linearly as resources are added to the cluster. Each point in the graph represents the average throughput observed over a 30 second window and the error bars show the 5th and 95th percentiles. At its peak, HyperDex is able to average 3.2 million operations per second.

The measurements reported in Figure 12 are taken in steady state. The reported measurements exclude the warm-up time for the system. In all experiments, 15 seconds was sufficient to achieve steady state. Clients operate in parallel, and are run on separate machines from the servers in all but the largest configurations. Clients issue requests in parallel, and each client maintains an average of 1,000 outstanding requests per server. Increasing the number of clients does not significantly impact the achievable throughput. The workload for each client consists of 95% get operations and 5% put operations with randomly generated 8 B keys and 64 B values.

This experiment shows that a medium-sized HyperDex cluster is able to achieve high throughput. For smaller cluster sizes, adding additional clients does not negatively impact throughput, and in many cases raises the 5th percentile mark.

7 Related Work

Database system Storage systems that organize their data in high-dimensional spaces were pioneered by the database community more than thirty years ago [7, 9, 20, 25, 30, 32, 36]. These systems, collectively known as Multi-Dimensional Databases (MDB), leverage multi-dimensional data structures to, through more optimized memory and disk access, improve the performance of data warehousing and online analytical processing applications. However, unlike hyperspaces in HyperDex, the data structures in MDBs are designed for organizing data on a single machine and are not directly applicable to large-scale distributed storage systems. Alternatively, more recent database systems [2, 17] have begun exploring efficient mechanisms for building and maintaining large-scale, tree-based distributed indices. HyperDex is a functional superset of distributed indices; it provides efficient search for many attributes without the overhead of building multiple distributed indices.

Peer-to-peer systems Past work in peer-to-peer systems have explored multi-dimensional network overlays to facilitate rapid and decentralized data storage and retrieval. Much like HyperDex, CAN [35], a distributed hash-table, organizes peers in a multi-dimensional space. However, CAN only provides key inserts and lookups; the purpose of CAN’s multi-dimensional peer configuration is to limit a CAN node’s peer-set size and provide a regular structure for efficient overlay routing.

MURK [19], SkipIndex [42], and SW AM-V [24] dynamically partition the multi-dimensional space into kd-trees, skip graphs, and Voronoi diagrams respectively to provide multi-dimensional range lookups. Although conceptually similar to HyperDex, providing coordination and management of nodes for these dynamic space partitioning schemes is significantly more complex and error-prone than for HyperDex’s static space partitioning and require additional operational overhead. These systems also do not address several critical and inherent problems associated with mapping structured data into a multi-dimensional space and providing reliable data storage. Specifically, they are not efficient for high dimensional data due to the curse-of-dimensionality, and
they either lack data replication or provide only eventually consistent operations on replicated data. Addressing these problems by augmenting dynamic space partitioning schemes with subspaces and value-dependent chaining would further increase the complexity and overhead of node coordination and management. Mercury [10] builds on top of a Chord [40] ring, and uses consistent hashing [23] on each attribute as secondary indexes. In contrast, hyperspace hashing maps multiple attributes simultaneously. Arpeggio[14] provides search over multiple attributes by creating an index of all fixed-size subsets the attributes using a Chord ring. In contrast with enumerating all \( \binom{n}{k} \) subsets of attributes, HyperDex hashes objects into a multi-dimensional space, and employs data partitioning to reduce dimensionality.

**Space-filling curves** A common approach to providing multi-attribute search uses space-filling curves to partition multi-dimensional data across the storage nodes. This approach uses the space filling curve to map the multi-dimensional data into one dimension, which then enables the use of traditional peer-to-peer techniques for performing multi-attribute searches. SCRAP [19], Squid [37] and ZNet [38] are examples of this approach with each node responsible for data in a contiguous range of values. Similarly, MAAN[12] performs the same mapping, but uses uniform locality preserving hashing. Space-filling curves do not scale well when the dimensionality is high, because a single search query may be partitioned into many one-dimensional ranges of considerably varying size. Furthermore, unlike in HyperDex, fully-qualified searches, where values for all attributes are specified, may involve contacting more than one node in space-filling curve-based systems.

**Key-Value Stores** Recently, key-value stores have become the focus of research because their restricted interface, scalability and high performance fill a gap left by traditional database systems. BigTable [13], and its open source derivatives [4, 26] share a similar data model in which data is structured in a single multi-attribute map that is indexed by a row key, column key, and timestamp. Dynamo [18] and its open source clone Riak [6] support a more-restricted key-value interface, and expose causal dependencies to allow the application to resolve conflicts. All of these systems employ \( O(1) \) hop lookup, and are designed to scale on commodity machines in a datacenter environment. HyperDex expands on the restricted interface of key-value stores without impacting scalability, consistency, and performance.

Other work on key-value stores has focused on exploiting specific properties of the underlying hardware to achieve high performance. Fawn KV [3] and SILT [27] build fast and efficient key-value storage systems on underpowered hardware in order to maximize the amount of throughput achieved per unit of consumed power. RAMCloud [31] stores the primary copy of data in RAM, and utilizes fast network connections to rapidly restore failed replicas COPS [28] improves the performance of geo-replicated key-value stores while at the same time strengthening consistency guarantees. Other systems [39] utilize a multi-tier storage hierarchy to exploit cache-oblivious algorithms in the storage layer. These systems improve the performance of key-value stores while retaining the same basic structure: a hash table. In HyperDex, we take a complementary approach which expands the key-value interface to support search operations. Nothing about our approach restricts combining techniques from these other systems with HyperDex’s hyperspace hashing technique.

While HyperDex is a key-value store which offers search properties, other systems propose using key-value stores to transparently accelerate traditional systems. TxCache [34] implements a transactional caching layer on top of the memcached in-memory key-value store. This cache layer improves the performance of the underlying PostgreSQL database by caching multiple versions of each object in the key-value store and maintaining these versions in tandem with the database.

**NoSQL Storage** HyperDex falls under the category of “NoSQL” storage systems, of which key-value stores form a very specific subcategory. These systems relax traditional ACID guarantees in order to improve scalability or performance. For instance, Yahoo!’s PNUTS [15] supports selection and projection from a single table, but does not support join operations. CouchDB [5] and MongoDB [1] are document databases which do not impose structure on the underlying data. HyperDex provides strong single-key consistency and enables searches over structured data. On the spectrum of “NoSQL” stores, HyperDex resides between key-value stores and document databases as it provides more structure than document databases, and more flexibility than key-value stores.

**8 Conclusions**

In this paper, we described HyperDex, a high-performance, scalable and searchable key-value storage systems for cloud computing. HyperDex provides an efficient search primitive for retrieving objects by specifying their secondary attributes. It achieves this extended functionality through **hyperspace hashing**, in which multi-attribute objects are deterministically mapped to coordinates in a low dimension Euclidean space. This mapping leads to efficient implementations for key retrieval, partially-specified searches and range-queries. A novel replication protocol enables the system to provide strong consistency without sacrificing performance.

We have developed a full open-source implementation of HyperDex and show, through a large-scale cloud
deployment, that it is practical and efficient. In our experiments, HyperDex provides comparable performance to other key-value stores for key operations, and offers several orders of magnitude higher performance for finding partially specified objects.

The recent trend toward NoSQL data stores has been fueled by scalability and performance concerns at the cost of functionality. HyperDex bridges this gap by providing additional functionality and flexibility without sacrificing scalability and performance. HyperDex offers a rich API and strong consistency guarantees that can enable high-performance, high-assurance key-value store applications.

References


