HyperDex: A Distributed, Searchable Key-Value Store

Robert Escriva
Computer Science Department
Cornell University
escriva@cs.cornell.edu

Bernard Wong
Cheriton School of Computer Science
University of Waterloo
bernard@uwaterloo.ca

Emin Gün Sirer
Computer Science Department
Cornell University
egs@systems.cs.cornell.edu

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—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 multidimensional 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 achieve strong consistency, maintain availability and guarantee fault tolerance. An evaluation of the full system shows that HyperDex is 12-13× faster than Cassandra and MongoDB for finding partially specified objects. Additionally, HyperDex achieves 2-4× higher throughput for `get`/`put` operations.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design

Keywords

Key-Value Store, NoSQL, Fault-Tolerance, Strong Consistency, Performance

1. INTRODUCTION

Modern distributed applications are reshaping the landscape of storage systems. Recently emerging distributed key-value stores such as BigTable [11], Cassandra [32] and Dynamo [19] form the backbone of large commercial applications because they offer scalability and availability properties that traditional database systems simply cannot provide. Yet these properties come at a substantial cost: the data retrieval API is narrow and restrictive, permitting an object to be retrieved using only the key under which it was stored, and the consistency guarantees are often quite weak. Queries based on secondary attributes are either not supported, utilize costly secondary indexing schemes or enumerate all objects of a given type.

This paper introduces HyperDex, a high-performance, scalable, consistent 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 using a novel technique called *hyperspace hashing*. Similar to other hashing techniques [23, 29, 36, 47], hyperspace hashing deterministically maps objects to servers to enable efficient object insertion and retrieval. But it differs from these techniques because it takes into account the secondary attributes of an object when determining the mapping for an object. Specifically, it maps objects to coordinates in a multi-dimensional Euclidean space – a *hyperspace* – which has axes defined by the objects' attributes. Each server in the system is mapped onto a region of the same hyperspace, and owns the objects that fall within its region. Clients use this mapping to deterministically insert, remove, and search for objects.

Hyperspace hashing facilitates efficient search by significantly reducing the number of servers to contact for each partially-specified `search`. The construction of the hyperspace mapping guarantees that objects with matching attribute values will reside on the same server. Through geometric reasoning, clients can restrict the search space for a partially-specified query to a subset of servers in the system, thereby improving search efficiency. Specificity in searches works to the clients' advantage: a fully-specified search contacts exactly one server.

A naive hyperspace construction, however, may suffer from a well-known problem with multi-attribute data known as "curse of dimensionality [6]." With each additional secondary attribute, the hyperspace increases in volume exponentially. If constructed in this fashion, each server would be responsible for a large volume of the resulting hyperspace, which would in turn force search operations to contact a large number of servers, counteracting the benefits of hyperspace hashing. HyperDex addresses this problem by partitioning the data into smaller, limited size *subspaces* of fewer dimensions.

Failures are inevitable in all large-scale deployments. The standard approaches for providing fault tolerance store objects on a fixed set of replicas determined by a primary key. These techniques, whether they employ a consensus algorithm among the replicas and provide strong consis-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM'12, August 13–17, 2012, Helsinki, Finland.
Copyright 2012 ACM 978-1-4503-1419-0/12/08 ...$15.00.

tency [18,46], or spray the updates to the replicas and only achieve eventual consistency [19,32,45,49], assume that the replica sets remain fixed even as the objects are updated. Such techniques are not immediately suitable in our setting because, in hyperspace hashing, object attributes determine the set of servers on which an object resides, and consequently, each update may implicitly alter the replica set. Providing strong consistency guarantees with low overhead is difficult, and more so when replica sets change dynamically and frequently. HyperDex utilizes a novel replication protocol called *value-dependent chaining* to simultaneously achieve fault tolerance, high performance and strong consistency. Value-dependent chaining replicates an object to withstand f faults (which may span server crashes and network partitions) and ensures linearizability, even as replica sets are updated. Thus, HyperDex's replication protocol guarantees that all `get` operations will immediately see the result of the last completed `put` operation – a stronger consistency guarantee than those offered by the current generation of NoSQL data stores.

Overall, this paper describes the architecture of a new key-value store whose API is one step closer to that of traditional RDBMSs while offering strong consistency guarantees, fault-tolerance for failures affecting a threshold of servers and high performance, and makes three contributions. First, it describes a new hashing technique for mapping structured data to servers. This hashing technique enables efficient retrieval of multiple objects even when the search query specifies both equality and range predicates. Second, it describes a fault-tolerant, strongly-consistent replication scheme that accommodates object relocation. 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 [15] benchmark. More specifically, HyperDex achieves 12-13× higher throughput for search workloads than the other systems, and consistently achieves 2-4× higher throughput for traditional key-value workloads.

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. We discuss how our system relates to the CAP Theorem in Section 8 and conclude with a summary of 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. As in a relational database, HyperDex objects match an application-provided schema that defines the typed attributes of the object and are persisted in tables. This organization permits straightforward migration from existing key-value stores and database systems.

HyperDex provides a rich API that supports a variety of datastructures and a wide range of operations. The system natively supports primitive types, such as `string`s, `integer`s and `float`s, as well as composite types, such as `list`s, `set`s or `map`s constructed from primitive types. The dozens of operations that HyperDex provides on these datatypes fall into three categories. First, *basic operations*, consisting of `get`, `put`, and `delete`, enable a user to retrieve, update, and destroy an object identified by its key. Second, the `search` operation enables a user to specify zero or more ranges for secondary attributes and retrieve the objects whose attributes fall within the specified, potentially singleton, ranges. Finally, a large set of atomic operations, such as `cas` and `atomic-inc`, enable applications to safely perform concurrent updates on objects identified by their keys. Since composite types and atomic operations are beyond the scope of this paper, we focus our discussion on the `get`, `put`, `delete`, and `search` operations that form the core of HyperDex.

2.2 Hyperspace Hashing

HyperDex represents each table as an independent multi-dimensional space, where the dimensional axes correspond directly to the attributes of the table. HyperDex assigns every object a corresponding coordinate based on the object's attribute values. An object is mapped to a deterministic coordinate in space by hashing each of its attribute values to a location along the corresponding axis.

Consider, for the following discussion, a table containing user information that has the attributes **first-name**, **last-name**, and **telephone-number**. For this schema, HyperDex would create a three dimensional space where the first-name attribute comprises the x-axis, the last-name attribute comprises the y-axis, and the telephone-number attribute comprises the z-axis. Figure 1 illustrates this mapping.

Hyperspace hashing determines the object to server mapping by tessellating the hyperspace into a grid of non-overlapping regions. Each server is assigned, and is responsible for, a specific region. Objects whose coordinates fall within a region are stored on the corresponding server. Thus, the hyperspace tessalation serves like a multi-dimensional hash bucket, mapping each object to a unique server.

The tessalation of the hyperspace into regions (called the *hyperspace mapping*), as well as the assignment of the regions to servers, is performed by a fault-tolerant coordinator. The primary function of the coordinator is to maintain the hyperspace mapping and to disseminate it to both servers and clients. The hyperspace mapping is initially created by dividing the hyperspace into hyperrectangular regions and assigning each region to a virtual server. The coordinator is then responsible for maintaining this mapping as servers fail and new servers are introduced into the system.

The geometric properties of the hyperspace make object insertion and deletion simple. To insert an object, a client computes the coordinate for the object by hashing each of the object's attributes, uses the hyperspace mapping to determine the region in which the object lies, and contacts that server to store the object. The hyperspace mapping obviates the need for server-to-server routing.

2.3 Search Queries

The hyperspace mapping described in the preceding sections facilitates a geometric approach to resolving `search`

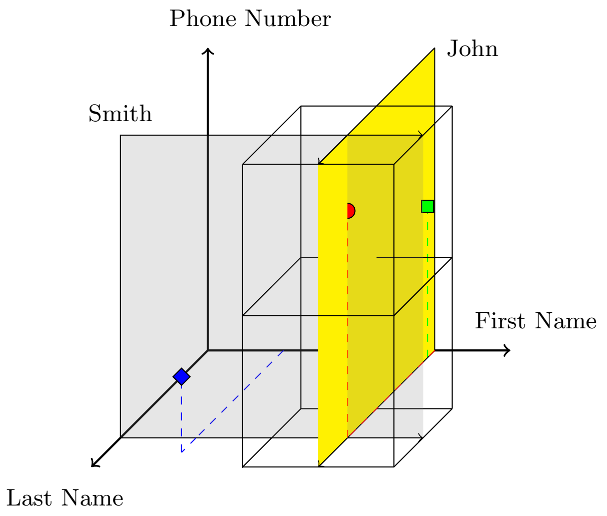

Figure 1: Simple hyperspace hashing in three dimensions. Each plane represents a query on a single attribute. The plane orthogonal to the axis for "Last Name" passes through all points for `last_name = 'Smith'`, while the other plane passes through all points for `first_name = 'John'`. Together they represent a line formed by the intersection of the two search conditions; that is, all phone numbers for people named "John Smith". The two cubes show regions of the space assigned to two different servers. The query for "John Smith" needs to contact only these servers.

operations. In HyperDex, a `search` specifies a set of attributes and the values that they must match (or, in the case of numeric values, a range they must fall between). HyperDex returns objects which match the `search`. Each `search` operation uniquely maps to a *hyperplane* in the hyperspace mapping. A `search` with one attribute specified maps to a hyperplane that intersects that attribute's axis in exactly one location and intersects all other axes at every point. Alternatively, a `search` that specifies all attributes maps to exactly one point in hyperspace. The hyperspace mapping ensures that each additional search term potentially reduces the number of servers to contact and guarantees that additional search terms will not increase `search` complexity.

Clients maintain a copy of the hyperspace mapping, and use it to deterministically execute `search` operations. A client first maps the `search` into the hyperspace using the mapping. It then determines which servers' regions intersect the resulting hyperplane, and issues the `search` request to only those servers. The client may then collect matching results from the servers. Because the hyperspace mapping maps objects and servers into the same hyperspace, it is never necessary to contact any server whose region does not intersect the `search` hyperplane.

Range queries correspond to extruded hyperplanes. When an attribute of a `search` specifies a range of values, the corresponding hyperplane will intersect the attribute's axis at every point that falls between the lower and upper bounds of the range. Note that for such a scheme to work, objects' relative orders for the attribute must be preserved when mapped onto the hyperspace axis.

Figure 1 illustrates a query for `first_name = 'John'` and `last_name = 'Smith'`. The query for `first_name = 'John'`

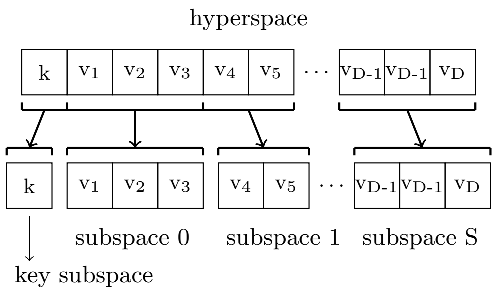

Figure 2: HyperDex partitions a high-dimensional hyperspace into multiple low-dimension subspaces.

corresponds to a two-dimensional plane which intercepts the `first_name` axis at the hash of `'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. Since a search for John Smith in a particular area code defines a line segment, a HyperDex search needs to contact only those nodes whose regions intersect that segment.

3. DATA PARTITIONING

HyperDex's Euclidean space construction significantly reduces the set of servers that must 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 be very large since its volume grows exponentially with the number of dimensions. Covering large spaces with a grid of servers may not be feasible even for large data-center deployments. For example, a table with 9 secondary attributes may require 2^9 regions or more to support efficient searches. In general, a D dimensional hyperspace will need $O(2^D)$ servers.

HyperDex avoids the problems associated with high-dimensionality by partitioning tables with many attributes into multiple lower-dimensional hyperspaces called *subspaces*. Each of these subspaces uses a subset of object attributes as the dimensional axes for an independent hyperspace. Figure 2 shows how HyperDex can partition a table with D attributes into multiple independent subspaces. When performing a `search` on a table, clients select the subspace that contacts the fewest servers, and will issue the `search` to servers in exactly one subspace.

Data partitioning increases the efficiency of a `search` by reducing the dimensionality of the underlying hyperspace. In a 9-dimensional hyperspace, a `search` over 3 attributes would need to contact 64 regions of the hyperspace (and thus, 64 servers). If, instead, the same table were partitioned into 3 subspaces of 3 attributes each, the `search` will never contact more than 8 servers in the worst case, and exactly one server in the best case. By partitioning the table, HyperDex reduces the worst case behavior, decreases the number of servers necessary to maintain a table, and increases the likelihood that a `search` is run on exactly one server.

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 many servers. Since applications often exhibit search locality, HyperDex applications can tune search efficiency by creating corresponding subspaces. As the number of subspaces grows, so, too, do the costs associated with maintaining data consistency across subspaces. Section 4 details how HyperDex efficiently maintains consistency across subspaces while maintaining a predictably low overhead.

3.1 Key Subspace

The basic hyperspace mapping, 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 practical 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 contacting more than one server. In this hypothetical scenario, key operations would be strictly more costly than key operations in traditional key-value stores.

HyperDex provides efficient key-based operations by creating a one-dimensional subspace dedicated to the key. This subspace, called the *key subspace*, ensures that each object will map to exactly one region in the resulting hyperspace. Further, this region will not change as the object changes because keys are immutable. To maintain the uniqueness invariant, `put` operations are applied to the key subspace before the remaining subspaces.

3.2 Object Distribution Over Subspaces

Subspace partitioning exposes a design choice in how objects are distributed and stored on servers. 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 would minimize storage requirements per server, as attributes are not duplicated across subspaces, it would lead to more expensive search and object retrieval operations since reconstituting the object requires cross-server cooperation. 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 server.

Hyperspace hashing supports both of these object distribution techniques. The HyperDex implementation, however, relies upon the latter 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 servers, maintaining a consistent view of objects poses a challenge. HyperDex employs a novel technique called *value-dependent chaining* to provide strong consistency and fault tolerance 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 such that the system can tolerate up to f failures in any one region.

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 mapped to multiple servers and these servers 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 servers, there would be no guarantees associated with subsequent operations on that object. Such a scheme would at best provide eventual consistency because servers 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 coordinate. The head of the chain is called the *point leader*, and is determined by hashing the object in the key subspace. Subsequent servers 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 remains pending until an acknowledgement of that update is received from the next server in the chain. When an update reaches the tail, the tail sends an acknowledgement back through the chain in reverse so that all other servers may commit the pending update and clean up transient state. When the acknowledgement reaches the point leader, the client is notified that the operation is complete. 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.

Updates to preexisting objects are more complicated because a change in an attribute value might require relocating an object to a different region of the hyperspace. Value-dependent chains address this by incorporating the servers assigned to regions for both the old and new versions of the object. Chains are constructed such that servers are ordered by subspace and the servers corresponding to the old version of the object immediately precede the servers corresponding to the new version. This guarantees that there is no instant during an update where the object may disappear from the data store. For example, in Figure 3, the update u_2 modifies the object in a way that changes its mapping in Subspace 0 such that the object no longer maps to h_2 and instead maps to h_5. The value-dependent chain for update u_2 is h_1, h_2, h_5, h_3. The update will result in the object being stored at h_5, and subsequently removed from h_2, as acknowledgments propagate in reverse.

Successive updates to an object will construct chains which overlap in each subspace. Consequently, 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_5, h_3. Notice that it

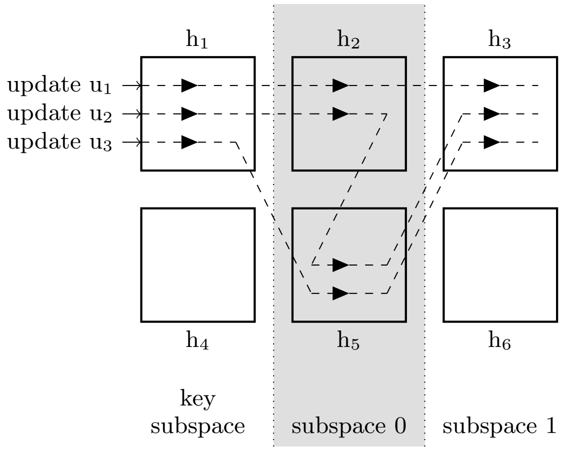

Figure 3: HyperDex's replication protocol propagates along value-dependent chains. Each update has a value-dependent chain that is determined solely by objects' current and previous values and the hyperspace mapping.

is possible for u_3 to arrive at h_5 before u_2. If handled improperly, such races could lead to inconsistent, out-of-order updates. Value-dependent chains efficiently handle this case by dictating that the point leader embed, in each update, 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, hash 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`. Servers which receive u_3 before u_2 will know to delay processing of u_3 until u_2 is also processed.

By design, HyperDex supports destructive operations that remove all state pertaining to deleted objects. Examples of destructive operations include `delete` and the cleanup associated with object relocation. Such operations must be carefully managed to ensure that subsequent operations get applied correctly. For instance, consider a `del` followed by a `put`. Since we would like a `del` to remove all state, the `put` must be applied on servers with no state. Yet, if another `del`/`put` pair were processed concurrently, servers which had processed either `del` would not be able to properly order the `put` operations. Value-dependent chains ensure that concurrently issued destructive operations are correctly ordered on all servers. Each server independently delays operations which depend upon a destructive operation until the destructive operation, and all that came before it, are acknowledged. This ensures that at most one destructive operation may be in-flight at any one time and guarantees that they will be ordered correctly. The delay for each message is bounded by the length of chains, and the number of concurrent operations.

4.2 Fault Tolerance

To guard against server failures, HyperDex provides additional replication within each region. The replicas acquire and maintain their state by being incorporated into value-dependent chains. In particular, each region has $f+1$ 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_1' through h_6'. As with regular chain replication [57], new replicas are introduced at the tail of the region's chain, and servers are bumped forward in the chain as other servers fail. For example, the first update in Figure 3 has the value-dependent chain $h_1, h_1', h_2, h_2', h_3, h_3'$. If h_2 were to fail, the resulting chain would be $h_1, h_1', h_2', h_2'', h_3, h_3'$. This transition will be performed without compromising strong consistency.

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. When a client detects a point leader failure, it will notify the application and preserve at-most-once semantics. Further, because all client acknowledgments are generated by the point leader, the client will only see a response after an object is fully fault-tolerant.

In our implementation, HyperDex uses TCP to transmit data which ensures that messages need only be retransmitted when servers fail and are removed from the value-dependent chain. In response to failures, HyperDex servers retransmit messages along the chains to make progress. Upon chain reconfiguration, servers will no longer accept messages from failed servers, ensuring that all future messages traverse the new, valid chain.

4.3 Server and Configuration Management

HyperDex utilizes a logically centralized coordinator to manage system-wide global state. The coordinator encapsulates the global state in the form of a *configuration* which consists of the hyperspace mapping between servers and regions and information about server failures. The coordinator assigns to each configuration an epoch identifier, a strictly increasing number that identifies the configuration. The coordinator distributes configurations to servers in order by epoch, and updates the configuration as necessary. HyperDex's coordinator holds no state pertaining to the objects themselves; only the mapping and servers.

HyperDex ensures that no server processes late-arriving messages from failed or out-of-date servers. Each HyperDex server process (an *instance*) is uniquely identified by its IP address, port and *instance id.* Instance ids are assigned by the coordinator and are globally unique, such that servers can distinguish between two instances (e.g. a failed process and a new one) that reuse the same IP and port number. HyperDex embeds into messages the instance ids, the regions of the hyperspace and indices into the chain for both the sender and recipient. A recipient acting upon a message can validate the sender and recipient against its most recent configuration. If the message contains a stale mapping, it will not be acted upon. If, instead, the mapping is valid, the host processes the message accordingly.

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 operation completes in sub-millisecond time on each host.

The logically centralized coordinator does not impose a bottleneck. Configurations are small in practice, and proportional to the size of the cluster and number of tables active on the cluster. Furthermore, in cases where bandwidth from the coordinator becomes a bottleneck, the coordinator need only distribute deltas to the configuration. Clients maintain a complete copy of the configuration in memory and perform local computation on the hyperspace mapping.

4.4 Consistency Guarantees

Overall, the preceding protocol ensures that HyperDex provides strong 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 [26] 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 will return all objects that were committed at the time of search. An application whose `put` succeeds is guaranteed to see the object in a future search. In the presence of concurrent updates, a search may return both the committed version, and the newly updated version of an object matching the search.

HyperDex provides the strongest form of consistency for key operations, and a conservative and predictable consistency guarantees for search operations.

5. IMPLEMENTATION

HyperDex is fully implemented to support all the features described in this paper. The implementation is nearly 44,000 lines of code. The HyperDex software distribution contains an embeddable storage layer called HyperDisk, a hyperspace hashing library, the HyperDex server, the client library and the HyperDex coordinator, as well as full client bindings for C, C++, and Python, and partial bindings for Java, Node.JS and Ruby.

This section describes key aspects of the HyperDex implementation.

5.1 HyperDex Server

High throughput and low latency access are essential for any key-value store. The HyperDex server achieves high performance through three techniques; namely, edge-triggered, event-driven I/O; pervasive use of lock-free datastructures and sharded locks to maximize concurrency; and careful attention to constant overheads. The HyperDex server is structured around a concurrent, edge-triggered event loop wherein multiple worker threads receive network messages and directly process client requests. Whereas common design patterns would distinguish between I/O threads and worker threads, HyperDex combines these two functions to avoid internal queuing and communication delays. Because the event-loop is edge-triggered, unnecessary interaction with the kernel is minimized. Socket buffer management ensures that threads never block in the kernel when sending a message and consumes a constant amount of memory per client.

HyperDex makes extensive use of lock-sharding and lock-free datastructures to reduce the probability of contention whenever possible. Per-key datastructures are protected by an array of locks. Although nothing prevents two threads from contending for the same lock to protect different keys, the ratio of locks to threads is high enough to reduce this occurrence to 1 in 10^6. Global datastructures, such as lookup tables for the per-key datastructures, are concurrent through the use of lock-free hash tables. Our implementation ensures that background threads may safely iterate over global state while worker threads insert and remove pointers to per-key state.

Finally, the use of cache-conscious, constant-time data structures reduces the overheads associated with common operations such as linked-list and hash-table management.

5.2 HyperDisk: On-Disk Data Storage

A key component of server performance for any key-value store is the storage back end used to organize data on disk. Since hyperspace hashing is agnostic to the choice of the back end, a number of design options are available. At one extreme, we could have used a traditional database to store all the objects of a server in a single, large, undifferentiated pool. While this approach would have been the simplest from an implementation perspective, it would make HyperDex dependent on the performance of a traditional database engine, require manual tuning of numerous parameters, and subject the system to the vagaries of a query optimizer.

Instead, HyperDex recursively leverages the hyperspace hashing technique to organize the data stored internally on a server. Called HyperDisk, this approach partitions the region associated with a server into smaller non-overlapping sub-regions, where a sub-region represents a file on disk, and objects are located in the file that contains their coordinate. Each file is arranged as a log, where insertions and deletions operate on the tail, and where a search operation linearly scans through the whole file.

HyperDisk's hashing scheme differs from the standard hyperspace hashing in two ways: first, HyperDisk partitions only the region assigned to a HyperDex server; and second, HyperDisk may alter the mapping dynamically to accommodate the number of objects stored within a region. Overall, recursive application of hyperspace hashing enables HyperDex to take advantage of the geometric structure of the data space at every system layer.

5.3 Distributed Coordination

In HyperDex, the hyperspace mapping is created and managed by a logically centralized coordinator. Since a physically centralized coordinator would limit scalability and pose a single point of failure, the HyperDex coordinator is implemented as a replicated state machine. It relies on a coordination service [2, 9, 27] to replicate the coordinator on multiple physical servers. The coordinator implementation ensures that servers may migrate between coordinators so that no coordinator failure leads to correlated failures 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.

Overall, the replicated state machine implementation ensures that the coordinator acts as a single, coherent component with well-defined state transitions, even though it is comprised of fault-tolerant, distributed components.

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) [15], an industry-standard benchmark for cloud storage performance. Our evaluation also examines the performance of HyperDex's basic operations, specifically, `get`, `put`, and `search`, using targeted micro-benchmarks. These micro-benchmarks isolate specific components and help expose the performance impact of design decisions. For both

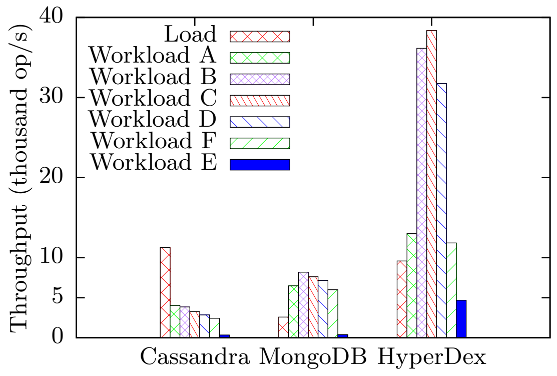

Figure 4: Average throughput for a variety of real-world workloads specified by the Yahoo! Cloud Serving Benchmark. HyperDex is 3-13 times faster than Cassandra and 2-12 times faster than MongoDB. Workload E is a search-heavy workload, where HyperDex outperforms other systems by more than an order of magnitude.

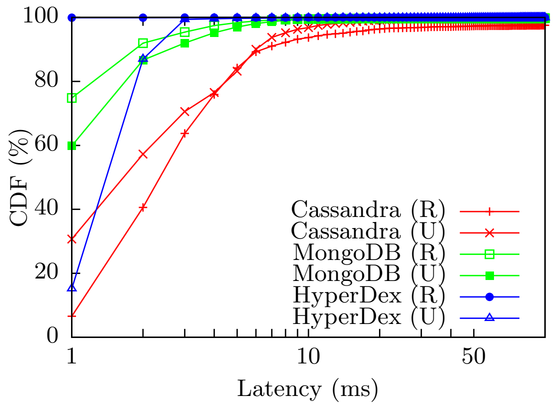

Figure 5: `GET/PUT` performance. Latency distribution for Workload A (50% reads, 50% updates, Zipf distribution).

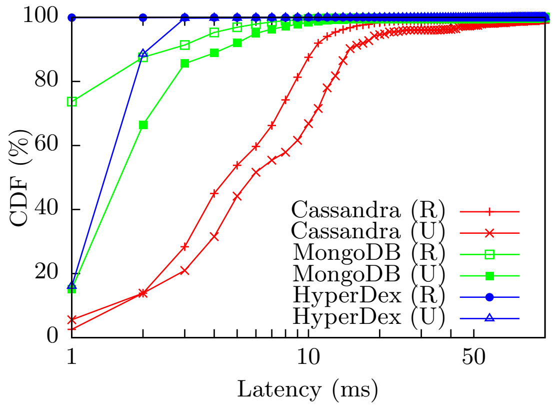

Figure 6: `GET/PUT` performance. Latency distribution for Workload B (95% reads, 5% updates, Zipf distribution). HyperDex maintains low latency for reads and writes.

YCSB and the micro-benchmarks, we compare HyperDex with Cassandra [32], a popular key-value store for Web 2.0 applications, and MongoDB [37], a distributed document database.

The performance benchmarks are executed on our small, dedicated lab-size cluster in order to avoid confounding issues arising from sharing a virtualized platform, while the scalability benchmarks are executed on the VICCI [42] 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 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 provide weak consistency and fault-tolerance guarantees; because acknowledgments are generated without full replication, small numbers of failures can lead to data loss. In contrast, HyperDex utilizes value-depending chaining and, as a result, always provides clients with strong consistency and fault-tolerance, even in the presence of failures. Since MongoDB allocates replicas in pairs, we allocate twelve machines for the storage nodes, one machine for the clients, and, where applicable, one node for the coordinator. HyperDex is configured with two subspaces in addition to the key subspace to accommodate all ten attributes in the YCSB dataset.

6.1 Get/Put Performance

High `get`/`put` performance is paramount to any cloud-based storage system. YCSB provides six different workloads that exercise the storage system with a mixture of request types and object distributions resembling real-world applications (Table 1). In all YCSB experiments, the database is preloaded with 10,000,000 objects and each operation selects the object and operation type in accordance with the workload specification. Figure 4 shows the throughput achieved by each system across the YCSB workloads. HyperDex provides throughput that is between a factor of two to thirteen higher than the other systems. The largest performance gains come from improvements in `search` performance. Significant improvements in `get`/`put` performance is attributable mostly to the efficient handling of `get` operations in HyperDex. Our implementation demonstrates that the hyperspace construction and maintenance can be realized efficiently.

In order to gain insight into the performance of the system, we examine the request latency distributions of the different systems under all read/write workloads. HyperDex's performance is predictable: all reads complete in under 1 ms, while a majority of writes complete in under 3 ms. Cassandra's latency distributions follow a similar trend for workloads B, C, D and F and show a slightly different trend for workload A. MongoDB, on the other hand, exhibits lower latency than Cassandra for all operations. For all workloads, HyperDex completes 99% of operations sooner than either Cassandra and MongoDB. Figures 5 and 6 show the latency distributions for workloads A and B respectively.

For completeness, we present the performance of all three systems on a write-heavy workload. Figure 7 shows the latency distribution for inserting 10,000,000 objects to set up the YCSB tests. Consistency guarantees have a significant effect on the `put` latency. MongoDB's default behavior considers a `put` complete when the request has been queued in

Name	Workload	Key Distribution	Sample Application
A	50% Read/50% Update	Zipf	Session Store
B	95% Read/5% Update	Zipf	Photo Tagging
C	100% Read	Zipf	User Profile Cache
D	95% Read/5% Insert	Temporal	User Status Updates
E	95% Scan/5% Insert	Zipf	Threaded Conversations
F	50% Read/50% Read-Modify-Write	Zipf	User Database

Table 1: The six workloads specified by the Yahoo! Cloud Serving Benchmark. These workloads model several applications found at Yahoo! Each workload was tested using the same YCSB driver program and system-specific Java bindings. Each object has ten attributes which total 1 kB in size.

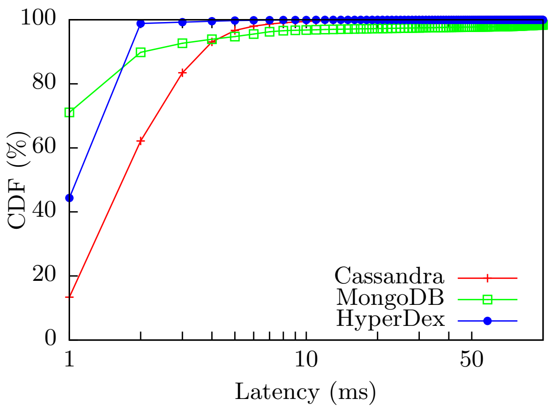

Figure 7: `PUT` performance. Latency distribution for 10,000,000 operations consisting of 100% insertions. Each operation created a new object under a unique, previously unused key. Although fast, HyperDex's minimum latency is bounded by the length of the value-dependent chains.

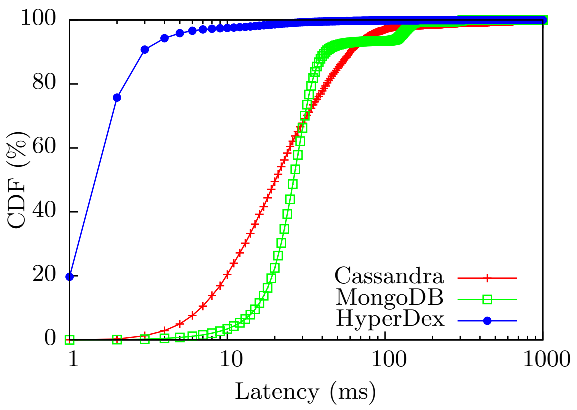

Figure 8: `SEARCH` performance. Latency distribution for 10,000 operations consisting of 95% range queries and 5% inserts with keys selected from a Zipf distribution. HyperDex is able to offer significantly lower latency for non-primary key range queries than the other systems are able to offer for primary-key range queries.

a client's send buffer, even before it has been seen by any server. Cassandra's default behavior considers a `put` complete when it is queued in the filesystem cache of just one replica. Unlike these systems, the latency of a HyperDex `put` operation includes the time taken to fully replicate the object on $f + 1$ servers. Because MongoDB does not wait for full fault tolerance, it is able to complete a majority of operations in less than 1 ms; however, it exhibits a long-tail (Figure 7) that adversely impacts average throughput. Similarly, Cassandra completes most operations in less than 2 ms. Despite its stronger fault-tolerance guarantees, HyperDex completes 99% of its operations 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 in the key itself. 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 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 8). Seventy five percent of search operations complete in less than 2 ms, and nearly all complete in less than 6 ms. Cassandra sorts data according to the primary key and is therefore able to retrieve matching items relatively quickly. Although one could alter YCSB to use Cassandra's secondary indexing schemes instead of key-based operations, the result would be strictly worse than what is reported for primary key operations. MongoDB's sharding maintains an index; consequently, `scan` operations in MongoDB are relatively fast. The `search` performance of HyperDex is not attributable to our efficient implementation as `search` is more than an order of magnitude faster in HyperDex, which eclipses the 2-4× performance advantage observed for `get`/`put` throughput. Hyperspace hashing in HyperDex ensures that search results are located on a small number of servers; this enables effective pruning of the search space and allows each search to complete by contacting exactly one host in our experiments.

An additional benefit of HyperDex's aggressive search pruning is the relatively low latency overhead associated with `search` operations. Figure 9 shows the average latency of a

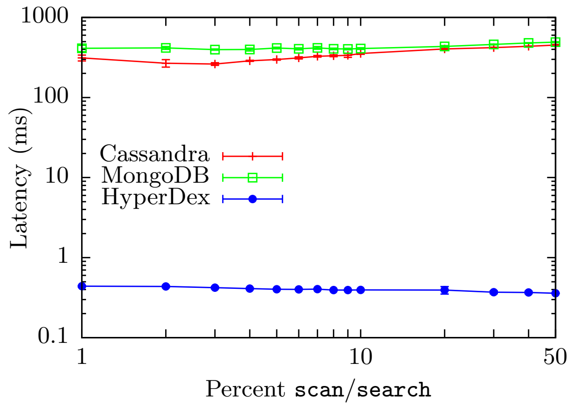

Figure 9: The effect of an increasing scan workload on latency. HyperDex performs significantly better than the other systems even as the scan workload begins to approach 50%.

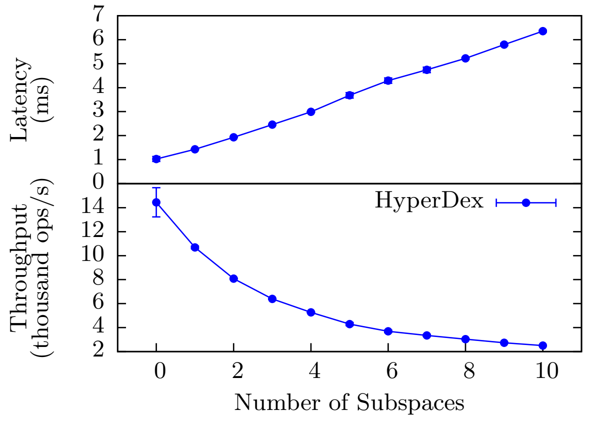

Figure 10: Latency and throughput for `put` operations as a function of non-key subspaces The error bars indicate standard deviation from 10 experiments. Latency increases linearly in the length of the chain, while throughput decreases proportionally. In applications we have built with HyperDex, all tables have three or fewer subspaces.

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 select objects. Using a uniform distribution ensures random access, and mitigates the effects of object caching. HyperDex consistently offers low latency for `search`-heavy workloads.

A critical parameter that affects HyperDex's search performance is the number of subspaces in a HyperDex table. Increasing the number of subspaces leads to additional opportunities for pruning the search space for *search* operations, but simultaneously requires longer value-dependent chains that result in higher `put` latencies. In Figure 10, we explore the tradeoff using between zero and ten additional subspaces beyond the mandatory key subspace. Note that adding ten additional subspaces increases the value-dependent chain to be at least 22 nodes long. As expected, HyperDex's `put` latency increases linearly with each additional subspace.

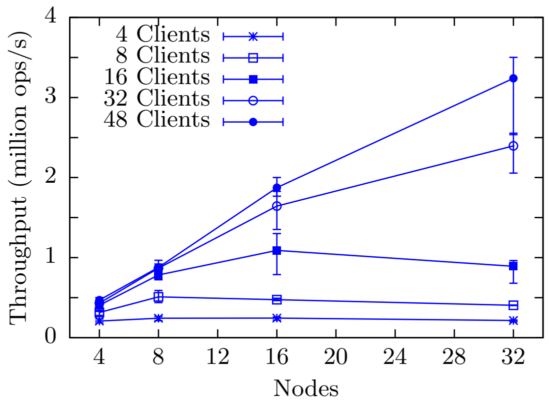

Figure 11: HyperDex scales horizontally. As more servers are added, aggregate throughput increases linearly. Each point represents the average throughput of the system in steady state over 30 second windows. The error bars show the 5th and 95th percentiles.

6.3 Scalability

We have deployed HyperDex on the VICCI [42] 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 for conducting 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 11, we explore the change in system throughput as resources are added to the cluster. As expected, 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 observed over any 1-second window. At its peak, HyperDex is able to average 3.2 million operations per second.

The workload for Figure 11 is a 95% read, 5% write workload operating on 8 B keys and 64 B values. The measurements reported are taken in steady state, with clients randomly generating requests according to the workload. This workload and measurement style reflects the workload likely to be encountered in a web application. 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 average throughput.

This experiment shows that a medium-sized HyperDex cluster is able to achieve high throughput for realistically sized deployments [3]. Additional resources allow the cluster to provide proportionally better throughput.

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 [5,7,24,31,39,41,51]. These systems, collectively known as Multi-Dimensional Databases (MDB), leverage multi-dimensional data structures to 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 [1, 17] have begun exploring efficient mechanisms for building and maintaining large-scale, tree-based distributed indices. In contrast, the mapping HyperDex constructs is not an index. Indices must be maintained and updated on object insertion. Hyperspace hashing, on other hand, is a mapping that does not change as objects are inserted and removed.

Peer-to-peer systems Past work in peer-to-peer systems has explored multi-dimensional network overlays to facilitate decentralized data storage and retrieval. CAN [47] is a distributed hash-table that, much like HyperDex, 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 to provide efficient overlay routing. HyperDex provides search, and does not do routing in the space.

MURK [21], SkipIndex [58], and SWAM-V [30] 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 [8] builds on top of a Chord [55] ring, and uses consistent hashing [29] on each attribute as secondary indexes. Although Mercury's implementation is unavailable, Cassandra uses a similar design and its performance illustrates how a multi-ring-based system would perform. Arpeggio [13] provides search over multiple attributes by enumerating and creating an index of all $\binom{k}{x}$ fixed-size subsets of attributes using a Chord ring. Both of these approaches insert redundant pointers into rings without concern for consistency.

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 a single dimension, which then enables the use of traditional peer-to-peer techniques for performing searches. SCRAP [21], Squid [52] and ZNet [53] are examples of this approach with each node responsible for data in a contiguous range of values. Similarly, MAAN [10] 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.

NoSQL Storage A new class of scalable storage systems, collectively dubbed "NoSQL", have recently emerged with the defining characteristic that they depart from the traditional architecture and SQL interface of relational databases. It is common practice for NoSQL systems to make explicit tradeoffs with respect to desirable properties. For instance, many NoSQL systems explicitly sacrifice consistency – even in the common case without failures – to achieve availability under extreme failure scenarios. NoSQL systems include document databases [16, 37] that offer a schema-free data model, in-memory cache solutions [36, 43, 48] that accelerate application performance, and graph databases [38] which model interconnected elements and key-value stores which offer predictable performance. Still, some systems do not fit neatly into these categories. For example, Yahoo!'s PNUTS [14] system supports the traditional SQL selection and projection, functions but does not support joins.

HyperDex explores a new point in the NoSQL design space. The rich HyperDex API provides qualitatively new functionality not offered by other NoSQL systems, HyperDex sets a new bar for future NoSQL systems by combining strong consistency properties with fault-tolerance guarantees, a rich API and high performance.

Key-Value Stores Modern key-value stores have roots in work on Distributed Data Structures [23,36] and distributed hash tables [29,47,50,55]. Most open source key-value stores draw heavily from the ring-based architecture of Dynamo [19] and the tablet-based architecture of BigTable [11]. For instance, Voldemort [45] and Riak [49] are heavily influenced by Dynamo's design. Other systems like HBase [25] and HyperTable [28] are open source implementations of BigTable. Cassandra [32] is unique in that it is influenced by BigTable's API and Dynamo's ring structure. Like HyperDex, all of these systems are designed to run in a datacenter environment on many machines.

Recent work on key-value stores largely focuses on improving performance by exploiting underlying hardware or manipulating consistency guarantees for a performance advantage. Fawn KV [4] builds a key-value store on underpowered hardware to improve the throughput-to-power-draw ratio. SILT [33] eliminates read amplification to maximize read bandwidth in a datastore backed by solid-state disk. RAMCloud [40] stores data in RAM and utilizes fast network connections to rapidly restore failed replicas. TSSL [54] utilizes a multi-tier storage hierarchy to exploit cache-oblivious algorithms in the storage layer. Masstree [35] uses concatenated B^+ trees to service millions of queries per second. COPS [34] provides a high-performance geo-replicated key-value store that provides causal+ consistency. Other systems [12, 19] trade consistency for other desirable properties, such as performance. Spanner [18] uses Paxos in the wide area to provide strong consistency. Spinnaker [46] uses Paxos to build a strongly consistent store that performs nearly as well as those that are only eventually consistent [19, 44, 56].

Each of these existing systems improves the performance, availability or consistency 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, among other operations, search over secondary attributes.

8. DISCUSSION

Surprisingly, the open-source release of the HyperDex system [20] uncovered various misunderstandings surrounding the CAP Theorem [22]. The popular CAP refrain ("C, A, P: pick any two") causes the subtleties in the definitions of C, A and P to be lost, even on otherwise savvy developers. There exists no conflict between claims in our paper and the CAP Theorem. The failure model used in the CAP Theorem is unconstrained; the system can be subject to partitions and node failures that affect any number of servers and network links. No system, including HyperDex, can simultaneously offer consistency and availability guarantees using such weak assumptions. HyperDex makes a stronger assumption that limits failures to affect at most a threshold of servers and is thus able to provide seemingly impossible guarantees.

9. CONCLUSIONS

This paper described HyperDex, a second-generation NoSQL storage system that combines strong consistency guarantees with high availability in the presence of failures and partitions affecting up to a threshold of servers. In addition, HyperDex provides an efficient search primitive for retrieving objects through 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-based retrieval, partially-specified searches and range-queries. HyperDex's novel replication protocol enables the system to provide strong consistency without sacrificing performance. Industry-standard benchmarks show that the system is practical and efficient.

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 without sacrificing scalability or performance.

10. ACKNOWLEDGMENTS

We would like to thank Pawel Loj for his contributions on cluster stop/restart, Deniz Altınbüken for her ConCoord Paxos implementation, and members of the HyperDex open source community for their code contributions, and the VICCI team for granting us time on their cluster. This material is based upon work supported by National Science Foundation under Grants No. CNS-1111698 and CCF-0424422 and by the National Science and Engineering Research Council of Canada.

11. REFERENCES

[1] M. K. Aguilera, W. M. Golab, and M. A. Shah. A Practical Scalable Distributed B-Tree. In *PVLDB,* 1(1), 2008.

[2] D. Altınbüken and E. G. Sirer. Commodifying Replicated State Machines with OpenReplica. Computing and Information Science, Cornell University, Technical Report 1813-29009, 2012.

[3] S. Anand. `http://www.infoq.com/presentations/NoSQL-Netflix/`.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In Proc. of *SOSP,* Big Sky, MT, Oct. 2009.

[5] R. Bayer. The Universal B-Tree for Multidimensional Indexing: General Concepts. In Proc. of *WWCA,* Tsukuba, Japan, Mar. 1997.

[6] R. E. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[7] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. In *CACM,* 18(9), 1975.

[8] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In Proc. of *SIGCOMM,* Portland, OR, Aug. 2004.

[9] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In Proc. of *OSDI,* Seattle, WA, Nov. 2006.

[10] M. Cai, M. R. Frank, J. Chen, and P. A. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. In Proc. of *GRID Workshop,* Phoenix, AZ, Nov. 2003.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. BigTable: A Distributed Storage System for Structured Data. In Proc. of *OSDI,* Seattle, WA, Nov. 2006.

[12] J. Cipar, G. R. Ganger, K. Keeton, C. B. M. III, C. A. N. Soules, and A. C. Veitch. LazyBase: Trading Freshness for Performance in a Scalable Database. In Proc. of *EuroSys,* Bern, Switzerland, Apr. 2012.

[13] A. T. Clements, D. R. K. Ports, and D. R. Karger. Arpeggio: Metadata Searching and Content Sharing with Chord. In Proc. of *IPTPS Workshop,* La Jolla, CA, Feb. 2005.

[14] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *PVLDB,* 1(2), 2008.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In Proc. of *SoCC,* Indianapolis, IN, June 2010.

[16] CouchDB. `http://couchdb.apache.org/`.

[17] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying Peer-to-Peer Networks Using P-Trees. In Proc. of *WebDB Workshop,* Paris, France, June 2004.

[18] J. Dean. Designs, Lessons, and Advice from Building Large Distributed Systems. Keynote. In Proc. of *LADIS,* Big Sky, MT, Oct. 2009.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In Proc. of *SOSP,* Stevenson, WA, Oct. 2007.

[20] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex. http://hyperdex.org/.

[21] P. Ganesan, B. Yang, and H. Garcia-Molina. One Torus to Rule Them All: Multidimensional Queries in P2P Systems. In Proc. of *WebDB Workshop,* Paris, France, June 2004.

[22] S. Gilbert and N. A. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. In *SIGACT News,* 33(2), 2002.

[23] S. D. Gribble. A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction. U.C. Berkeley, 2000.

[24] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In Proc. of *SIGMOD,* 1984.

[25] HBase. `http://hbase.apache.org/.`

[26] M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. In *ACM ToPLaS,* 12(3), 1990.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In Proc. of *USENIX,* Boston, MA, June 2010.

[28] Hypertable. `http://http://hypertable.org/.`

[29] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In Proc. of *STOC,* El Paso, TX, May 1997.

[30] F. B. Kashani and C. Shahabi. SWAM: A Family of Access Methods for Similarity-Search in Peer-to-Peer Data Networks. In Proc. of *CIKM,* Washington, D.C., Nov. 2004.

[31] A. Klinger. Patterns and Search Statistics. In *Optimizing Methods in Statistics,* Academic Press, 1971.

[32] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. In Proc. of *LADIS,* Big Sky, MT, Oct. 2009.

[33] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In Proc. of *SOSP,* Cascais, Portugal, Oct. 2011.

[34] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In Proc. of *SOSP,* Cascais, Portugal, Oct. 2011.

[35] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In Proc. of *EuroSys,* Bern, Switzerland, Apr. 2012.

[36] Memcached. `http://memcached.org/.`

[37] MongoDB. `http://www.mongodb.org/.`

[38] Neo4j. `http://neo4j.org/.`

[39] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. In *ACM ToDS,* 9(1), 1984.

[40] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In Proc. of *SOSP,* Cascais, Portugal, Oct. 2011.

[41] J. A. Orenstein and T. H. Merrett. A Class of Data Structures for Associative Searching. In Proc. of *PODS,* 1984.

[42] L. Peterson, A. Bavier, and S. Bhatia. VICCI: A Programmable Cloud-Computing Research Testbed. Princeton, Technical Report TR-912-11, 2011.

[43] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In Proc. of *OSDI,* Vancouver, Canada, Oct. 2010.

[44] D. Pritchett. BASE: An ACID Alternative. In *ACM Queue,* 6(3), 2008.

[45] Project Voldemort. `http://project-voldemort.com/.`

[46] J. Rao, E. J. Shekita, and S. Tata. Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. In *PVLDB,* 4(4), 2011.

[47] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A Scalable Content-Addressable Network. In Proc. of *SIGCOMM,* San Diego, CA, Aug. 2001.

[48] Redis. `http://redis.io/.`

[49] Riak. `http://basho.com/.`

[50] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In Proc. of *ICDSP*, volume 2218, 2001.

[51] H. Samet. Spatial Data Structures. In *Modern Database Systems: The Object Model, Interoperability, and Beyond,* Addison Wesley/ACM Press, 1995.

[52] C. Schmidt and M. Parashar. Enabling Flexible Queries with Guarantees in P2P Systems. In *Internet Computing Journal,* 2004.

[53] Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou. Supporting Multi-Dimensional Range Queries in Peer-to-Peer Systems. In Proc. of *IEEE International Conference on Peer-to-Peer Computing,* Konstanz, Germany, Aug. 2005.

[54] R. P. Spillane, P. J. Shetty, E. Zadok, S. Dixit, and S. Archak. An Efficient Multi-Tier Tablet Server Storage Architecture. In Proc. of *SoCC,* 2011.

[55] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In Proc. of *SIGCOMM,* San Diego, CA, Aug. 2001.

[56] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In Proc. of *SOSP,* Copper Mountain, CO, Dec. 1995.

[57] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In Proc. of *OSDI,* San Francisco, CA, Dec. 2004.

[58] C. Zhang, A. Krishnamurthy, and R. Y. Wang. SkipIndex: Towards a Scalable Peer-to-Peer Index Service for High Dimensional Data. Princeton, Technical Report TR-703-04, 2004.