Query-efficient Partitions for Dynamic Data

Nikos Vasilakis
University of Pennsylvania
3330 Walnut Street
Philadelphia, Pennsylvania 19104
nvas@cis.upenn.edu

Yash Palkhiwala
University of Pennsylvania
3330 Walnut Street
Philadelphia, Pennsylvania 19104
yashp@seas.upenn.edu

Jonathan M. Smith
University of Pennsylvania
3330 Walnut Street
Philadelphia, Pennsylvania 19104
jms@cis.upenn.edu

ABSTRACT

Large-scale data storage requirements have led to the development of distributed, non-relational databases (NoSQL). Single-dimension NoSQL achieves scalability by partitioning data over a single key space. Queries on primary ("key") properties are made efficient at the cost of queries on other properties. Multidimensional NoSQL systems attempt to remedy this inefficiency by creating multiple key spaces. Unfortunately, the structure of data needs to be known *a priori* and must remain fixed, eliminating many of the original benefits of NoSQL.

This paper presents three techniques that together enable query-efficient partitioning of *dynamic* data. First, unispace hashing (UH) extends multidimensional hashing to data of unknown structure with the goal of improving queries on secondary properties. Second, compression formulas leverage user insight to address UH's inefficiencies and further accelerate lookups by certain properties. Third, formula spaces use UH to simplify compression formulas and accelerate queries on the structure of objects. The resulting system supports dynamic data similar to single-dimension NoSQL systems, efficient data queries on secondary properties, and novel intersection, union, and negation queries on the structure of dynamic data.

KEYWORDS

Key-Value Store, Dynamic, NoSQL, Partitioning, Queries

1 INTRODUCTION

Scalability requirements during the last decade have led to the development of distributed, non-relational databases (NoSQL). Single-dimension NoSQL [5, 11, 18] divides data into partitions over the dimension of a "key" property whose values are unique for each object (Fig. 1 middle). Since the partitioning scheme depends only on a single property, the structure of the rest of the object (*i.e.*, its "secondary" properties) does not need to be known *a priori* nor does it need to remain fixed. Data can be dynamic and have their structure change during the program's runtime. This flexibility worked well with dynamic programming languages (*e.g.*, Ruby, Python, JavaScript, PHP) and interchange formats (*e.g.*, XML, JSON) popular in application development. However, an inability to exploit structure means that queries on properties other than the primary key become inefficient, as all partitions must be searched.

Multidimensional key-value stores, as pioneered by Hyperdex [6], attempt to remedy this problem by partitioning on multiple dimensions (Fig. 1 (c)). To create such a hyperspace, however, the system depends heavily on structure: it requires *a priori* knowledge of the structure of objects, it does not support changes to the object's properties, and needs to maintain a mapping from regions of a property's values to underlying nodes on the side.

The goal of our work is to enable efficient partitioning and querying of *dynamic data* using three techniques. *Unispace hashing* is a generalization of hyperspace hashing [6] that uses property names to identify which dimensions an object

```
var ac = {
 username: "aph",
 first: "Alyssa",
 last: "Hacker"
};
```

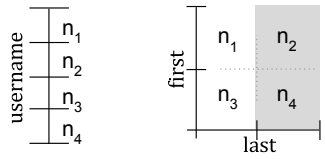


Figure 1: A dynamic object with three properties (left). Four servers partitioning similar objects by (middle) `username`; (right) `first-last`.

	1D	HH	UH
Dynamic Object Structure	✔	✘	✔
Efficient Search on "Secondary" Properties	✘	✔	✔
No Dimension-to-Node Mapping	✔	✘	✔
No Bounds on Number of Nodes	✔	✘	✔
Queries on Structure (*e.g.*, Union, Intersection)	✘	✘	✔

Table 1: Summary of features: (a) 1D (single-dimension NoSQL), (b) HH (Hyperspace Hashing) (c) UH (Unispace Hashing).

APSys '17, September 2, 2017, Mumbai, India

represents. This enables support for dynamic data and accelerated queries on secondary properties, but does not make ideal use of the available space of dimensions (§3). Therefore, *compression formulas* can be used to tune space use by configuring queryable dimensions at the granularity of individual objects. Formulas bring many benefits (§4), but their use needs to be consistent between all operations targeting a specific object. To alleviate this potential for inconsistencies, the system employs *formula spaces* (§5): it takes advantage of the fact that formulas are themselves dynamic objects to store and query them, adding a layer of indirection between their description and their use. This additional layer can be used to accelerate queries on the structure of dynamic objects (*e.g.*, unions, intersections). The resulting hybrid aims to support dynamic data similar to single-dimension NoSQL, efficient data queries on secondary properties similar to multi-dimension NoSQL, and novel queries on the structure of stored data (§7).

2 BACKGROUND AND NOTATION

Consider four nodes with ids n_1 to n_4; a function $H(s)$ that maps strings to nodes n_i; and an object `ac` that we want to store to one of our nodes. For now, we can think of $H(s) = h(s)\%4$, where h is a hash function. Objects are sets of properties: each property is a pair of a property *name* and a property *value*. In Fig. 1, the `ac` object has three properties: `username`, `first`, and `last`.

One of these properties takes values that are – or can be made – unique across all objects (*e.g.*, `username`). This property is often termed "key" in the distributed key-value store literature and is used to partition the data on a single dimension (Fig. 1 mid). Assuming the same nodes and "key", operations by "key" require contacting a single node, namely $H($`ac.username`$)$. The result is independent of the node receiving the request, independent of the property names and overall structure of the object, and is achieved without maintaining any indices or side-structures. Unfortunately, however, searching by other properties (*e.g.*, `first`, `last`, or both) requires contacting *every* node.

Hyperspace hashing [6] is a generalization of the previous idea to multiple dimensions. Assuming `first` and `last` are enough to uniquely identify an object, it partitions the two-dimensional plane into the four nodes n_1 to n_4 (Fig. 1 right). Insertion and retrieval require contacting the node at coordinates ($H($`ac.first`$)$, $H($`ac.last`$)$). Retrieval by partially-specified queries on secondary attributes is still more efficient than exhaustive search: to return all "Hacker"s, the system needs to contact only half of the nodes (shaded area). The system successfully solves queries on secondary attributes, but requires *a priori* knowledge of object structure, disallows changes to the number, names, and types of its properties, and maintains an explicit, centralized mapping from dimensions to nodes. Moreover, since partitioning is determined statically, changes in the number of available nodes may render the partitioning scheme void.[1]

Based on the previous discussion, our scheme has to solve three main challenges (Table 1): (i) handle objects whose structure is not known beforehand, (ii) provide efficient queries on "secondary" properties, and (iii) remove the need of a mapping from dimensions to nodes. A solution should not pose any requirements on the number of nodes (*e.g.*, work on a single node) to ensure use in any environment. Finally, since all objects are dynamic, it should offer efficient queries on their structure (*e.g.*, return all objects with a property name "model").

3 UNISPACE HASHING

The core technique is an extension to hyperspace hashing. To allow querying, each object is represented as a point in a multi-dimensional space. As with hyperspace hashing, the coordinate for each dimension is determined by hashing the object's property values. Unlike hyperspace hashing, dimensions are determined by hashing the object's property *names*.

All operations draw deterministically from a set of dimensions D with size $|D|$. For now, we assume a fixed number r of regions (nodes) per dimension. In single-dimensional systems r can be thought as the number of nodes in the cluster. We will later use r to assign multiple regions per physical server as a way to "even out" differences in the server's relative capabilities. Hashing the name of each property returns an integer from 0 to $|D| - 1$. Using this number to index in D returns a dimension D_i. Hashing the value of the property corresponding to this name returns a value from 0 to $r - 1$. This is the coordinate value for dimension D_i. Coordinate values for dimensions corresponding to property names that are not present get a default value of 0. Coordinate values for dimensions whose property values are unknown get the full range of values in r.

Insertions and updates require fully-specified objects. That is, the value of each property needs to be present in order to determine the location of the object. Queries and deletions fill unknown coordinates with wildcards: they will need to search all regions that fall under the values of a specific dimension.

To illustrate insertion and query, we will be using the `ac` object from Fig. 1, an r of three regions per dimension, and a 10-dimensional D. The `put(ac)` operation inserts `ac` into the database. It first calculates $h($`"username"`$)\%10$, $h($`"first"`$)\%10$,

[1] This is different from fault tolerance: there might not be enough nodes to even partition the data! In our example, if node n_3 did not exist, the scheme collapses because there are not enough servers to support the required dimensions.

and h("last")%10, all integers in the range from 0 to 9, inclusive. It then calculates h("aph")%3, h("Alyssa")%3, and h("Hacker")%3, all integers in the range from 0 to 2, inclusive. Suppose the first set of results is 3, 4, and 8 respectively; and the second is 1, 1, 2. The coordinate vector is the following:

```
[0, 0, 0, 1, 1, 0, 0, 0, 2, 0]
```

The `get({username: ANY, first: "Alyssa", last: ANY})` operation looks for all objects with an attribute of name `first` whose value is "Alyssa" and any `username` and `last` property. The resulting coordinate vector requires looking into all regions with coordinates:

```
[0, 0, 0, {0-2}, 1, 0, 0, 0, {0-2}, 0]
```

It will only search within nine out of 59,049 regions. If we want to look for similar objects that either do not have a username or have a username of "aph", the respective operations are `get({first: "Alyssa", last: ANY})` and `get({username: "aph", first: "Alyssa", last: ANY})` resulting in the following coordinate vectors:[2]

```
[0, 0, 0, 2, 1, 0, 0, 0, {0-2}, 0]
[0, 0, 0, 0, 1, 0, 0, 0, {0-2}, 0]
```

Each one of them will only hit three different regions.

It is important to note that this scheme works identically with any number of physical nodes. That is, it hides the distinction between distributed and non-distributed regions. For example, with a single node, regions can correspond to memory partitions. Tessellation, the process of assigning regions within a dimension to storage buckets (*i.e.*, IDs – they could refer to nodes or memory cells), can be done dynamically during runtime as long as all nodes agree on the same ordering of IDs. This is the only agreement required upon system startup or reconfiguration.

4 COMPRESSION FORMULAS

Unispace hashing as presented in §3 solves the challenges enumerated in Table 1. However, issues remain:

- It severely penalizes properties that can *uniquely* identify an object (*e.g.*, the "key" property). Using the previous examples, a query that only includes `username` should be enough to return a *single* node. Instead, it just reduces the search space by an order of magnitude (in base r). In fact, we need a fully-specified query to fill a single coordinate vector completely and get a single node – but then, we already have the object we are looking for!
- It wastes dimensions for properties not used for queries. Usually, there exist properties that are used only *after* the result is retrieved, but are never as exact search terms. Examples include multi-word text, template metadata, multimedia, lists of property values, and methods (code). Even if we wanted to search within some of these types, they require special pre-processing.
- It assigns equal query priority to all properties. Given a specific number of nodes, users should be able to accelerate selected queries at the expense of others. For example, it is more common to look up people based on their first and last name, and less common to look them up by eye color.

These issues require user insight, which is supplied by augmenting all operations with a second argument specifying a *compression formula*. For instance, users can insert objects using `put(`obj_1`,` ϕ_1`)` and query using `get(`q_1`,` ϕ_1`)`.

Formulas are configuration objects that specify structural preferences at the level of individual objects. They instruct the system on how to (re)construct the coordinate space on each operation. The q_1 argument above does not need to include wildcard properties (*e.g.*, `ANY`) of an object any more. Knowledge about which dimensions contain known values, which contain wildcards, and which are not even indexed can all be expressed using the second argument, formula ϕ_1.

Compression formulas are centered around three configuration parameters: queryable dimensions, weights, and space overlays.

Queryable Dimensions At the very least, users can specify a subset of dimensions that are important for queries. To locate where to place the object, the system will run the scheme described in the previous section *only* on the dimensions specified in this subset. If any of the properties specified does not exist, it will get a value of 0. The following formula is equivalent to setting a `username` as a primary key in a distributed key-value store.

```
{ space: ["username"] };
```

Weights Users can specify the relative ratio of regions per dimension between the dimensions they plan to index. A higher number of regions for a property means that queries with this property will be serviced more efficiently. The example formula below specifies that queries on `first` should be twice as efficient as queries on `last`.

```
{ space: {"first": 4, "last": 2} };
```

Space Overlays Users can create multiple overlays that are optimized for different types of queries. Each overlay can either contain a copy of the object or a pointer to a single location for this object (specified by, say, hashing all its contents). Since updates to any of its values changes the location of the object for all overlays that include updated values, the former is ideal for read-heavy systems and the latter for write-heavy systems. For queries that touch multiple overlays, the system can process queries with the goal

[2] Suppose h("aph")%3 results in 2.

```
var ac_f = [
 "username",
 "ssn",
 { "first": 4,
   "last": 2 }
];
```

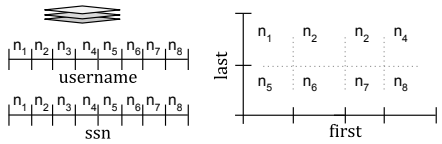


Figure 2: Three space overlays resulting from a compression formula.

of querying the smallest number of regions. The example below specifies three overlays; the previous two, and a third one for ssn:

```
{ spaces: [
  { space: ["username"] }, { space: ["ssn"] }
  { space: {"first": 4, "last": 2} }
]}
```

Fig. 2 illustrates the resulting spaces, and a more concise syntax actually used by the system today. If the ac object from Fig. 1 was updated to include, say, a notes property, none of the resulting spaces would use it to index ac.

Formulas have several features. They are dynamic: they can be generated during runtime for individual objects. They are also optional; if no formula is provided, the system will still operate as described in the previous section — at a possibly non-ideal configuration. Finally, they maintain the pure, deterministic nature of operations: given (a) a set of nodes (implicitly), (b) a data or query object (as before), and now (c) a compression formula (new), the system will return the same node *independently of the node receiving the request.*

5 FORMULA SPACES

So far our scheme solves the problems as posed (Table 1); and by taking advantage of user insight, it makes judicious use of available resources. However, the use of compression formulas introduces several inconveniences. These can be grouped into two main categories:

- *Formula Management*: Even though formulas are optional, a use upon insertion requires the exact same formula upon query, update, and deletion of the same object. Moreover, users need to manage formulas explicitly and make sure to save and retrieve them between system interruptions.
- *Overlay Reconstruction*: The introduction of overlays makes property-based searching more complicated as it requires knowledge about (i) which overlays include a specific property and (ii) how to reconstruct them, in order to locate the objects. This requires access to all formulas across the system that include a specific property.

It becomes clear that the system needs to store formulas and make their retrieval on secondary attributes efficient. But formulas are themselves dynamic objects, therefore the system can store and query them using the schemes already described. It can use indexable dimensions to avoid indexing metadata that are stored along with the formulas such as inverted indices. It can also use several overlays to accelerate operations on objects that have the structure of a formula. The next few paragraphs explain the details.

Identifiers First, formulas get a property named ID. Its value is unique and is used to distinguish between different formulas. IDs can be thought as distributed pointers for naming formulas: normal operations are overloaded to also accept a string in place of a formula argument, which is used to locate and retrieve the formula.

Identical IDs mean identical sets of properties for the formula object. Users can assign human-meaningful IDs such as "Car", or "specialCarInstance". If not provided by the user, IDs are generated by the system using the formula's property names as input. In both cases, users can query or update them similar to any other object. The system also optimizes ID-based operations by using a dedicated space overlay with a single dimension.

The following example shows the use of formula IDs. Suppose we store the following formula:

```
var cf = {id: "cf_user", spaces: {...}};
put(cf); // insert formula to DB
```

Then the following two statements are semantically equivalent:

```
put(obj, cf); put(obj, "cf_user");
```

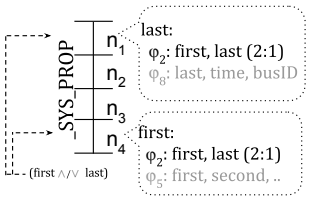
The first will run as if the formula was given verbatim. The second will first retrieve the formula and then run the operation.

Inverted Formulas To facilitate quick lookup of formulas by property, the system maintains a distributed map from object-properties to formulas containing these properties. It partitions this map by object-property name on a single dimension (_SYS_PROP on Fig. 3 left). By retrieving formulas, the system can reconstruct each space overlay with its own dimensions, coordinates, and weights.

Inverted formulas are particularly useful for searching for *all* objects that contain a specific property, *independently* of the formula used to store them. For example, the following operation will return all objects that include a property named first regardless of formula used:

```
get({first: ANY});
```

Structural Queries Since the system is already storing compression formulas, we can use the inverted formula space to efficiently answer union, intersection, and negation queries on dynamic data. These queries now amount to getting all the formulas that include the properties needed and running a union or intersection on *them*. This returns only the spaces that are guaranteed to contain the properties the user




```
function union(propertyList):
 formulaSet = new Set();
 for (p in propertyList):
   F = get({_SYS_PROP: p}, "-1");
   for (f in F):
     if (f.containsAny(propertyList)):
       formulaSet.add(f)
 return formulaSet;
```

```
function intersection(propertyList):
 formulaSet = new Set();
 for (p in propertyList):
   F = get({_SYS_PROP: p}, "-1");
   for (f in F):
     if (f.containsAll(propertyList)):
       formulaSet.add(f)
 return formulaSet;
```

Figure 3: Use of inverted formulas and queries on structure. Union returns ϕ_2, ϕ_8, and ϕ_5; intersection returns ϕ_2.

cares about. The pseudocode in Fig. 3 shows how union and intersection queries are handled at the formula space.

This is a lot more efficient than querying the data objects for several reasons: (i) there is a smaller number of formulas, as they get reused for multiple objects (*e.g.*, all objects that look like "car" share the same formula); (ii) formulas are much smaller than the data objects they describe (*i.e.*, on the order of an object's queryable property *names* only); (iii) the resulting object is guaranteed to have the requested structure.[3]

6 PROTOTYPE IMPLEMENTATION

Since the focus of this work is *dynamic* data, we chose to implement our prototype in JavaScript, a widely used dynamic programming language built around prototypes. JavaScript has first-class support for JSON for data interchange, which we use to serialize, store, and query data.

We built our prototype on top of Andromeda [23], a system aimed at simplifying the development of large-scale, distributed, general-purpose applications. The hosted version of Andromeda runs each node as a userspace process. Node management, synchronization, and communication are handled by Andromeda's built-in services. Low-level internals are handled by Node.js [4], a runtime that bundles (i) Google's V8, a fast JIT compiler, (ii) libUV, cross-platform wrappers for file-system and network operations, and (iii) several standard libraries, including OpenSSL used for hashing (SHA512). Andromeda currently exposes a variant of rendez-vous hashing [22] instead of the more common consistent hashing [10].

Excluding all Andromeda code, the current prototype is approximately 900 lines of code. It uses TCP for both local and remote communication. It exposes four methods (`put`, `get`, `patch`, and `del`) that follow an asynchronous programming style accessible through service named `uni` (*i.e.*, `andromeda.uni.*`).

Unispace hashing (§3) and the queryable-dimensions aspect of formulas (§4) are implemented as described in this paper. Several other features are not completely implemented (*e.g.*, weight and overlay formulas (§4), and have only partial support for formula spaces (§5)). Transactional semantics is also handled by the application's control layer, not the storage layer.

A challenge is race conditions potentially affecting the formula storage. Insertion, update, and delete operations on data have the potential of updating the stored formulas, but care needs to be taken as formula updates need to be isolated and atomic. Instead of locking, reading, updating, and unlocking the relevant inverted map state, remote nodes send functions that get interleaved using a cooperative multi-tasking scheme. The cooperative scheduling scheme guarantees isolation and atomicity, and minimizes associated overheads. In practice, inverted maps get only a fraction of the updates that data objects get.

7 PRELIMINARY EVALUATION

We explored (i) the sources of potential overheads and their impact, and (ii) the characteristics of dynamic workloads.

Evaluation Setup: The development version of Andromeda that was used came with Node.js 6.9.1 and was bundled with V8 v.5.1.281.84, libUV v.1.9.1, and OpenSSL v.1.0.2j. Experiments were run on a Linux server with 512GB of memory and 160 hyper-threaded cores running at 2.27 GHz. For all the experiments, we spawn 32 nodes as processes on the same machine, and store data in a memory-backed file-system.

To understand the sources of overheads, we synthesize read/write micro-benchmarks with formulas of increasing complexity (1-5 dimensions). We use 1M calls of randomly-generated objects that contain an average of five properties per object, 10-character property keys and 100-character property values.

To understand the properties of more dynamic workloads and see whether our proposed system would be a good fit, we use DBLP's 2GB bibtex database [12] as a more realistic, dynamic workload. We convert 307,780 bibtex entries to JSON objects and replace all multi-word property values with single-word equivalents to simplify query generation.

Micro-benchmarks: With a single dimension, similar to single-dimension key-value stores, our prototype system averages a throughput of 125950.78 and 154660.45 objects

[3] In general, most dynamically-typed languages behave like structurally-typed languages. Under a structural-subtyping [16] lens then, a more precise statement would be that these queries return all the objects that are structural subtypes of or structurally-equivalent with the query object.

per second for writes and reads (*i.e.*, fully-specified queries), respectively. Average latencies are 12.57ms and 10.26ms at full throughput, respectively.

Each extra indexable dimension adds a 0.1–0.2% throughput and 0.5–0.8% latency overhead. To highlight potential bottlenecks, we used statistical CPU profiling (DTrace) in the five-dimensional case. It revealed that most of the time is spent on IPC/network functions (76.18%) and not in database activity.[4]

Within a single node and excluding IPC/network and serialiation/deserialization overheads, microbenchmarks indicate that hashing is a considerable source of overhead (18.26%). Since SHA512 is a *cryptographic* hash function, it wastes too many resources for guarantees that are not useful in our setting. We tried a portable, JavaScript-only version of SipHash [1] as a candidate for a fast, collision-resistant, non-cryptographic hash function. Unfortunately, the results of hashing 23588700 strings (averaging a length of 9.569 characters each) were not as promising: the built-in SHA512 takes a total of 655.34ms, whereas SipHash takes 2825.230ms. As a point of comparison, TweetNaCl's [3] JavaScript-only SHA512 takes 4364.860ms. The big takeaway here is that crypto-to-non-crypto improvement is only 1.5× when the JITed-to-built-in is 4.3×. The main drawback of adding a native non-crypto function is a serious loss of portability.

Another approach would be to offload computation to clients. Most of the computation of identifying which node is responsible for a value (*e.g.*, tessellation and hashing) is pure (§4). We isolated parts of the code that compute the tessellation and coordinate results (roughly 500LoC of JavaScript) and wrapped them with a small RESTful interface (another 100LoC). Using this client library, different clients can independently compute the result of which server to contact. The results seem promising: using the previous experiment, the server goes from fully-saturated to non-saturated. The main challenge of client offloading is keeping in-sync with changes in the node topology.

DBLP Bibliography Database: The DBLP workload highlighted a number of interesting properties. First, *most* of the entries are *mostly* the same: 61% of the properties of each object (roughly 6) are common between 99.99% of objects; 28% of them are common between 95% of the objects; and 11% is only rarely shared between objects. Moreover, all of the objects have a "key" property that can be used to uniquely identify each entry. Finally, the union turns out to have more than 30 properties – a prohibitive amount for any system based on single-space, multi-dimensional hashing.

A problem with *static* multi-dimensional hashing à la Hyperdex [6] is that it requires going through the whole dataset and identifying the union of all properties. This is not possible when objects are being added dynamically, as in DBLP. Even in the case of our experiment with only a static snapshot, processing the dataset to calculate the union takes several seconds. Moreover, although Hyperdex supports subspaces, there is no way to set "don't-care" properties or "upgrade" to more dimensions. A system such as Redis [18] has none of these problems as it supports dynamic data without any special configuration, but does not support efficient queries on secondary properties, pervasive in the case of DBLP.

[4] As a point of comparison, raw process-to-process data exchange via TCP averages a throughput of 188.1MB/s and a latency of 5.8 – 12.6ms on the host where the experiments were run.

8 RELATED WORK

High-dimensional database techniques have been of interest to the data-base and data-structure communities for decades (*e.g.*, k-d trees [2], R-trees [8], the grid file [14], z-order [15], R+-trees [19]). They either require a static object (*i.e.*, table) structure [14, 15] or, more commonly, adapt on the data they see [2, 8], which makes straightforward distribution very difficult.

Other lines of work [7, 9, 13] focus on reducing dimensionality down to a single dimension to then enable single-dimension partitioning schemes. Space-filling curves [13] do this, for example, by tracking a single curve through all regions of a multi-dimensional space. This is a promising direction for dealing with very high dimensionality, and is somewhat similar in spirit to mapping a multidimensional matrix into a single region of physical memory, as done in our system. However, scalability may be hindered by (i) multi-dimensional queries partitioned into single-dimensional ranges of different sizes, and (ii) space regions falling under multiple nodes.

Locality-preserving hashing [9] and locality-sensitive hashing [7] offer multi-dimensional range and nearest-neighbor (*i.e.*, fuzzy, wildcard) queries. However, since they are sensitive to data for preserving locality, they do not offer good load-balancing features. In some cases, space partition is based on previously-seen data – so different nodes might have inconsistent views of the space – and offer only approximate guarantees [7].

On the other hand, distributed hashing techniques provide uniform load balancing without depending on previously-seen data. Somewhat more complex hashing schemes (*e.g.*, rendez-vous hashing [22], consistent hashing [10]) can minimize data shuffling in dynamically-changing systems. These ideas have been used in various settings (*e.g.*, P2P [17, 20]) and led to the revolution of distributed, single-dimensional (*i.e.*, key-value) NoSQL storage systems (*e.g.*, Dynamo [5], Redis [18], Cassandra [11]). Hyperdex [6] improved on the idea in many ways (§2). Our work extends Hyperdex's partitioning technique, namely hyperspace hashing, for data whose structure is not known beforehand and can change

during runtime. Replication and consistency are orthogonal issues and can be served by schemes complementary to ours (*e.g.*, Hyperdex [6], Replex [21]).

9 CONCLUSION

This paper addresses the problem of using dynamic data in distributed storage settings where query efficiency on secondary properties is a primary concern. It uses three complementary techniques that together aim to support dynamic data similar to single-dimension NoSQL, efficient data queries on secondary properties similar to multi-dimension NoSQL, and novel queries on the structure of stored data.

Acknowledgements. This work would not have been possible without Ben Karel's gentle prodding. Several other discussions provided helpful feedback: input from Luke Valenta improved the exposition of the two-step hashing; input from Radoslav Ivanov confirmed several flexibilities of formulas; input from Jocelyn Quaintance ruled out projection-based dimension compression. We would also like to thank the anonymous reviewers for their comments and suggestions. This research was funded in part by National Science Foundation grant CNS-1513687. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Jean-Philippe Aumasson and Daniel J Bernstein. 2012. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*. Springer, 489–508. https://131002.net/siphash/

[2] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517. https://doi.org/10.1145/361002.361007

[3] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. 2014. TweetNaCl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 64–83. https://tweetnacl.cr.yp.to/

[4] Ryan Dahl and the Node.js Foundation. 2009. Node.js. (2009). https://nodejs.org Accessed: 2017-06-11.

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[6] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A Distributed, Searchable Key-value Store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2342356.2342360

[7] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 518–529. http://dl.acm.org/citation.cfm?id=645925.671516

[8] Antonin Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*. ACM, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[9] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. 1997. Locality-preserving Hashing in Multidimensional Spaces. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC '97)*. ACM, New York, NY, USA, 618–625. https://doi.org/10.1145/258533.258656

[10] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC '97)*. ACM, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[11] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[12] Michael Ley. 2002. The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives. In *String Processing and Information Retrieval, 9th International Symposium, SPIRE 2002, Lisbon, Portugal, September 11-13, 2002, Proceedings*. 1–10. https://doi.org/10.1007/3-540-45735-6_1

[13] Bongki Moon, H. v. Jagadish, Christos Faloutsos, and Joel H. Saltz. 2001. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE Transactions on Knowledge and Data Engineering* 13, 1 (Jan. 2001), 124–141. https://doi.org/10.1109/69.908985

[14] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.* 9, 1 (March 1984), 38–71. https://doi.org/10.1145/348.318586

[15] Jack A. Orenstein. 1986. Spatial Query Processing in an Object-oriented Database System. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD '86)*. ACM, New York, NY, USA, 326–336. https://doi.org/10.1145/16894.16886

[16] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press. https://www.cis.upenn.edu/~bcpierce/tapl/

[17] Antony I. T. Rowstron and Peter Druschel. 2001. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg (Middleware '01)*. Springer-Verlag, London, UK, UK, 329–350. http://dl.acm.org/citation.cfm?id=646591.697650

[18] Salvatore Sanfilippo and Pieter Noordhuis. 2009. Redis. (2009). https://redis.io Accessed: 2016-09-30.

[19] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '87)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 507–518. http://dl.acm.org/citation.cfm?id=645914.671636

[20] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*. ACM, New York, NY, USA, 149–160. https://doi.org/10.1145/383059.383071

[21] Amy Tai, Michael Wei, Michael J. Freedman, Ittai Abraham, and Dahlia Malkhi. 2016. Replex: A Scalable, Highly Available Multi-Index Data Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, USA, 337–350. https://www.usenix.org/conference/atc16/technical-sessions/presentation/tai

[22] D. G. Thaler and C. V. Ravishankar. 1998. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking* 6, 1 (Feb 1998), 1–14. https://doi.org/10.1109/90.663936

[23] Nikos Vasilakis, Ben Karel, and Jonathan M. Smith. 2015. From Lone Dwarfs to Giant Superclusters: Rethinking Operating System Abstractions for the Cloud. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/vasilakis