HyperDex: Next Generation NoSQL

Emin Gün Sirer
Robert Escriva Bernard Wong

CloudPhysics
June 28, 2013

From RDBMS to NoSQL

- RDBMS have difficulty with scalability and performance
- NoSQL systems emerged to fill the gap

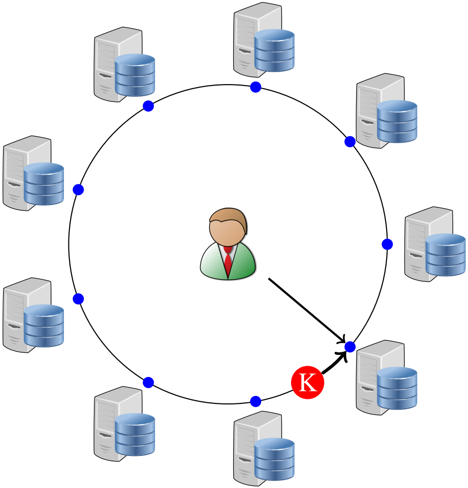
Problems Typical of NoSQL

Lack of ...

- Search
- Consistency
- Fault-Tolerance

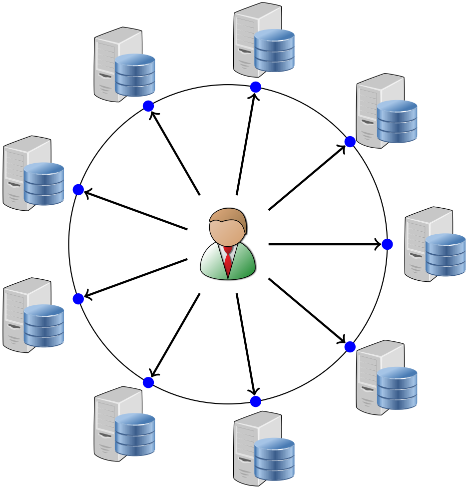
Specifics vary between systems

Typical NoSQL Architecture



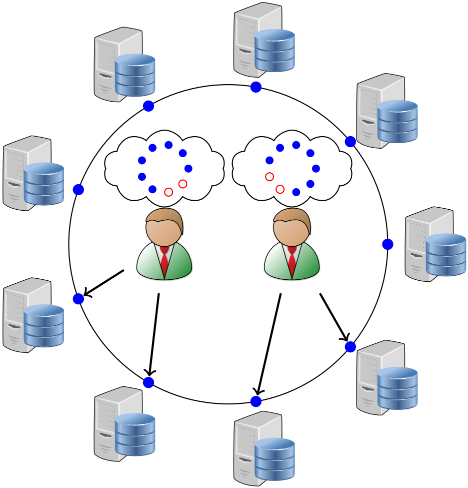
Consistent hashing maps each key to a server

The Search Problem



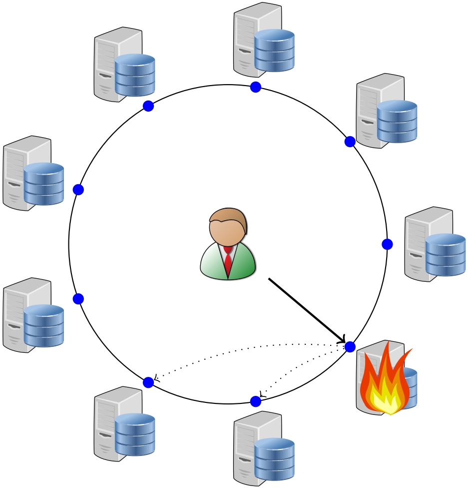
Searching for objects without the key involves many servers

The Consistency Problem



Clients may read inconsistent data and writes may be lost

The Fault-Tolerance Problem



Many systems' default settings consider a write complete after writing to just one node

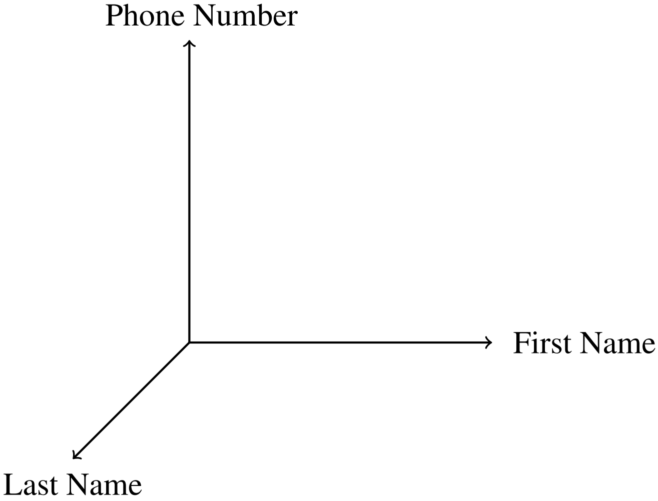
HyperDex: An Overview

- Hyperspace hashing
- Value-dependent chaining
- ACID Transactions

$$\Downarrow$$

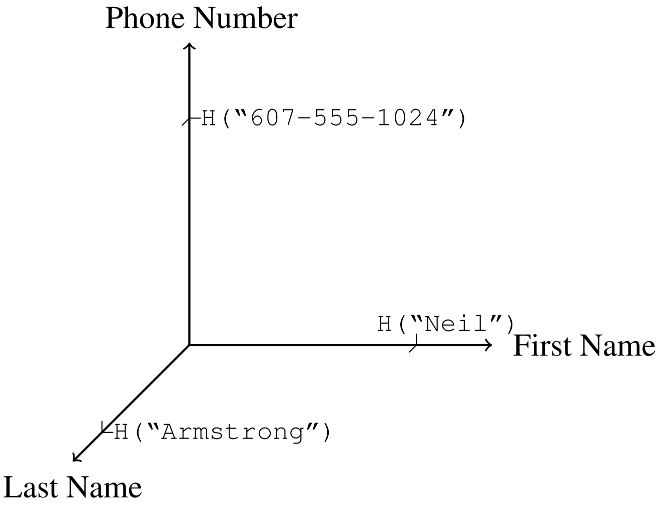
- High-Performance: High throughput with low variance
- Strong Consistency: Strong safety guarantees
- Fault Tolerance: Tolerates a threshold of failures
- Scalable: Adding resources increases performance
- Rich API: Support for complex datastructures and search

Attributes map to dimensions in a multidimensional hyperspace

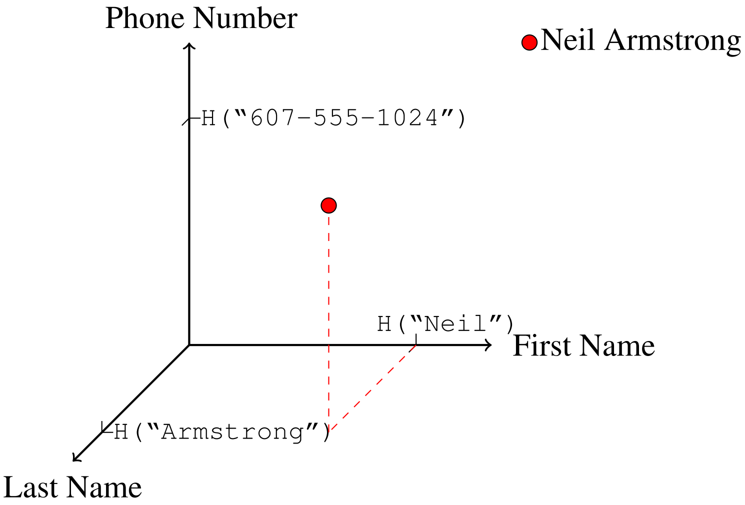


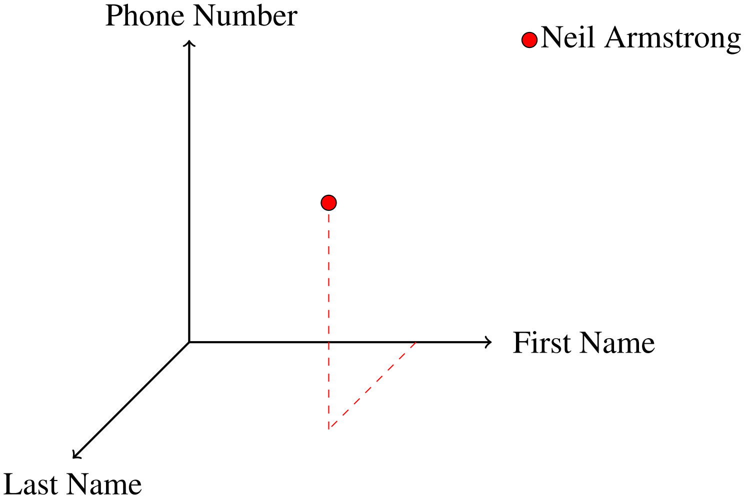
Attribute values are hashed independently

Any hash function may be used

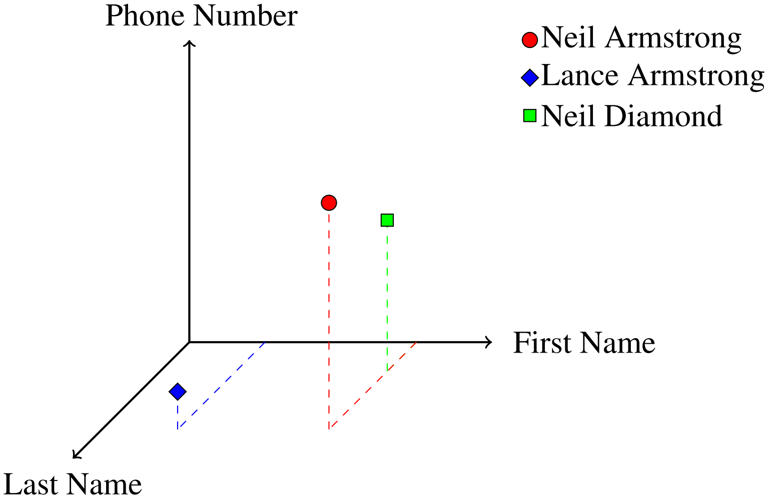


Objects reside at the coordinate specified by the hashes

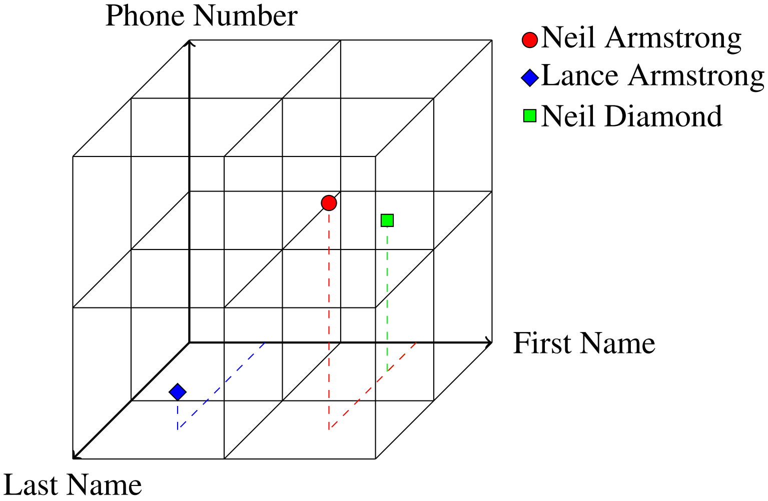



Phone Number
Neil Armstrong
First Name
Last Name

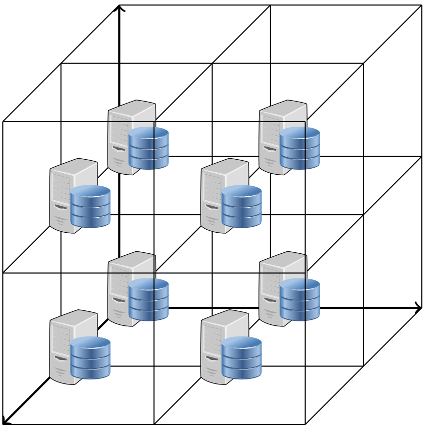
Different objects reside at different coordinates



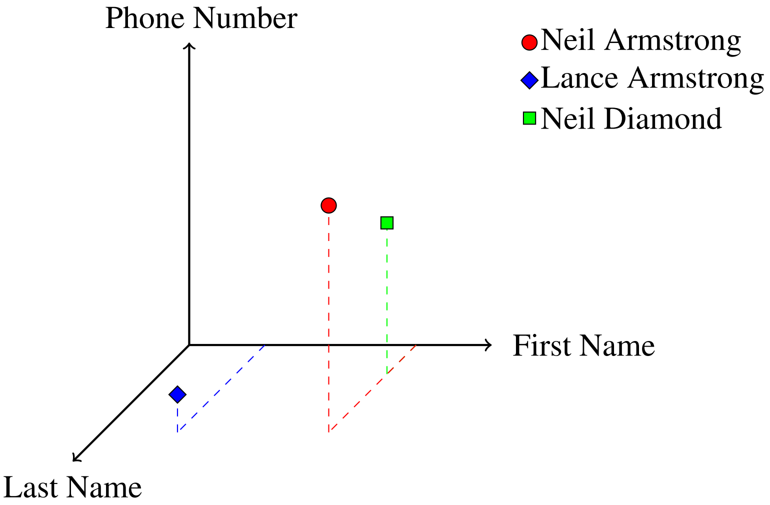
The hyperspace is divided into **regions** where each object resides in exactly one region



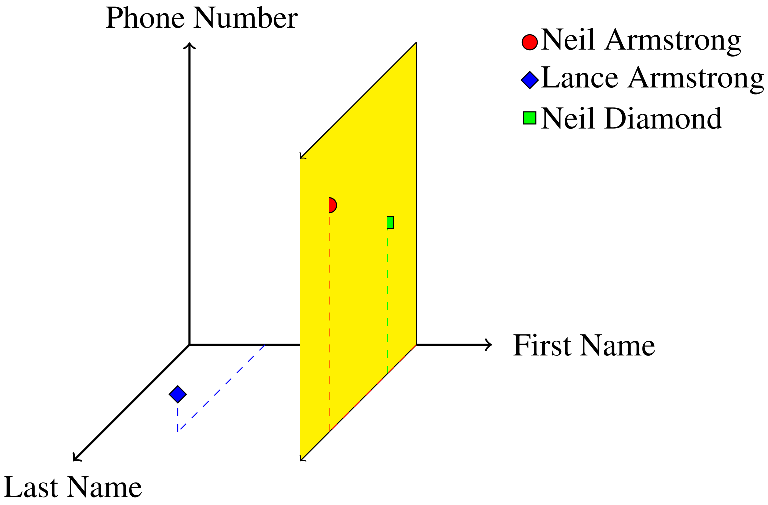
Each server is responsible for a region of the hyperspace



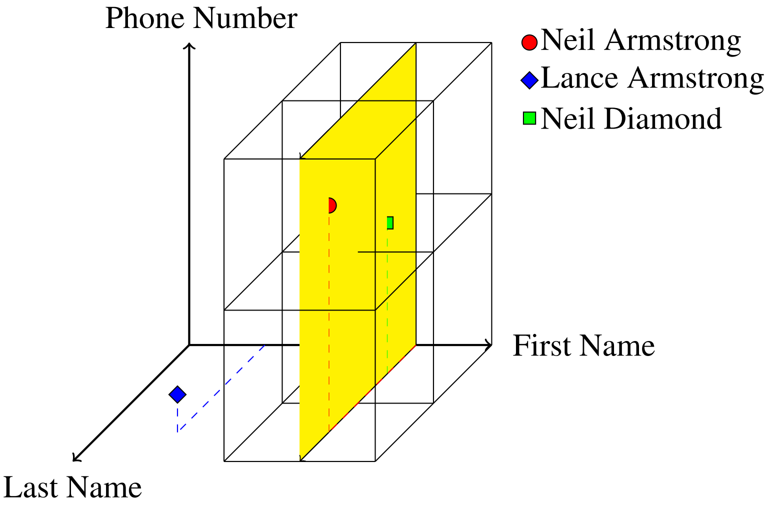
Each search intersects a subset of regions of the hyperspace



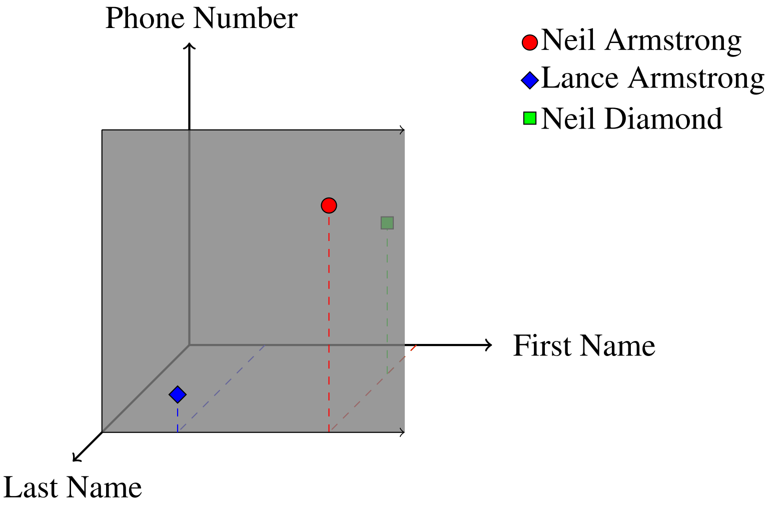
All people named Neil are mapped to the yellow plane



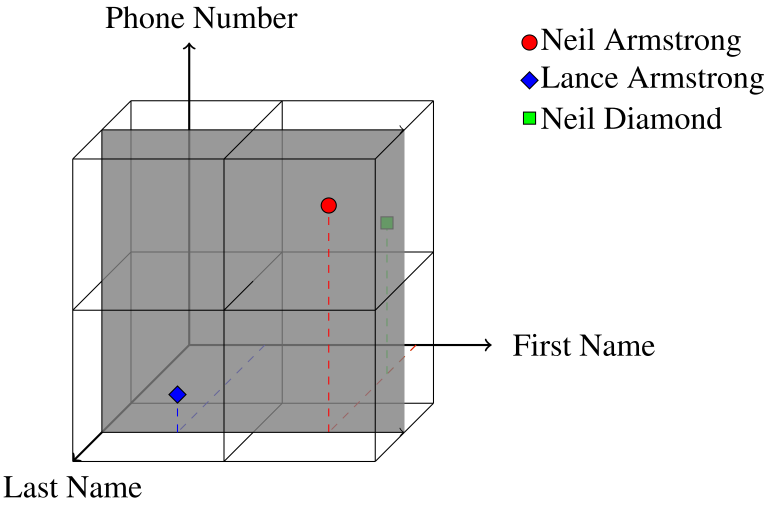
All people named Neil are mapped to the yellow plane



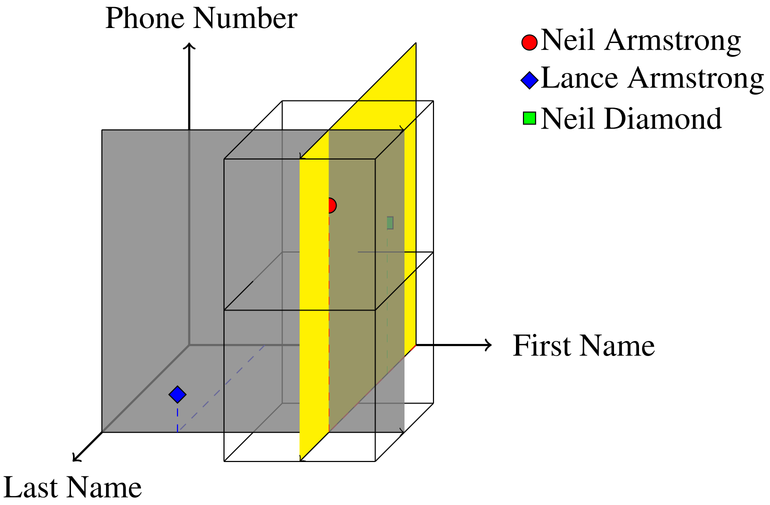
All people named Armstrong are mapped to the gray plane



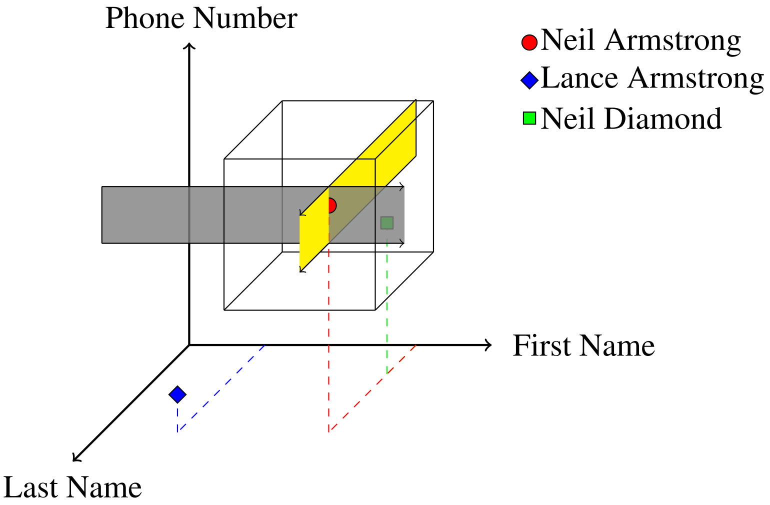
All people named Armstrong are mapped to the gray plane



A more restrictive search for Neil Armstrong contacts fewer servers



Range searches are natively supported



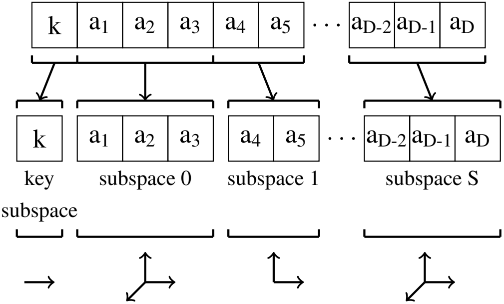
Space Partitioning

- In a naive implementation, the hyperspace would grow exponentially in the number of dimensions
- *Space partitioning* prevents exponential growth in the number of searchable attributes

k	a_1	a_2	a_3	a_4	a_5	$\cdots$	a_{D-2}	a_{D-1}	a_D

Space Partitioning

- In a naive implementation, the hyperspace would grow exponentially in the number of dimensions
- *Space partitioning* prevents exponential growth in the number of searchable attributes



- A search is performed in the most restrictive subspace

Space Partitioning

- In a naive implementation, a 9-dimensional space could require 512 machines
- HyperDex can store this space on just 24 machines using three subspaces

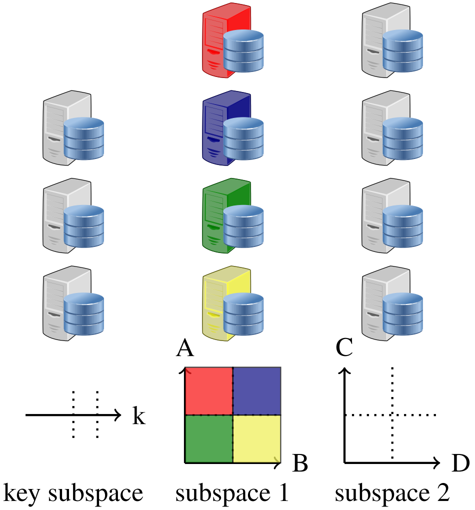
Hyperspace Hashing Implications

- `search`es are efficient
- Hyperspace hashing is a mapping, not an index
  - No per-object updates to a shared datastructure
  - No overhead for building and maintaining B-trees
  - Functionality gained solely through careful placement

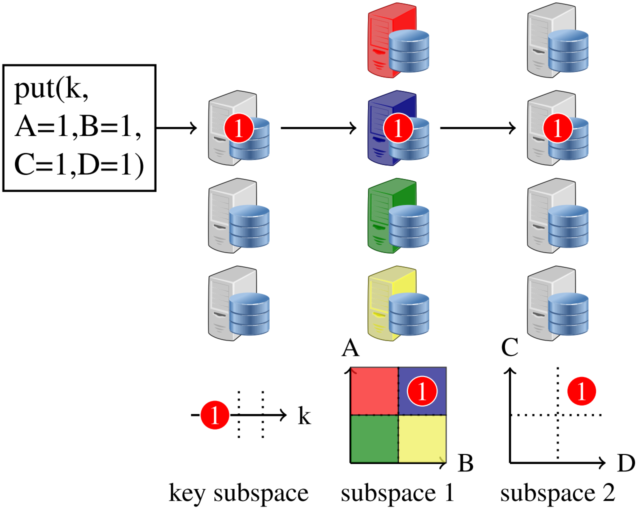
Replication

- As an object changes, so too must the set of servers holding it

Value-Dependent Chaining

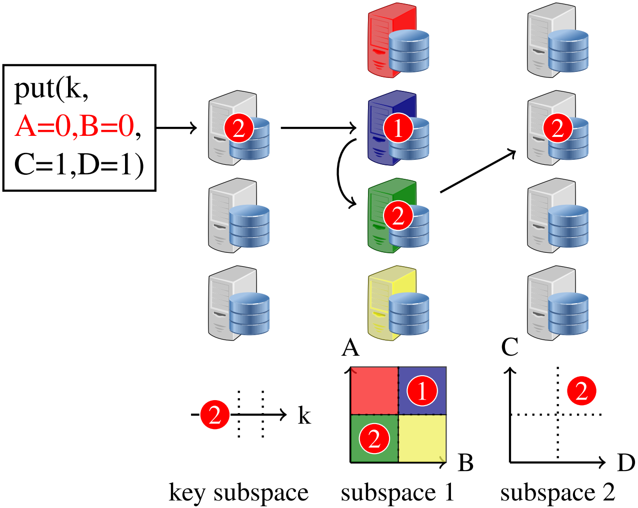


Value-Dependent Chaining



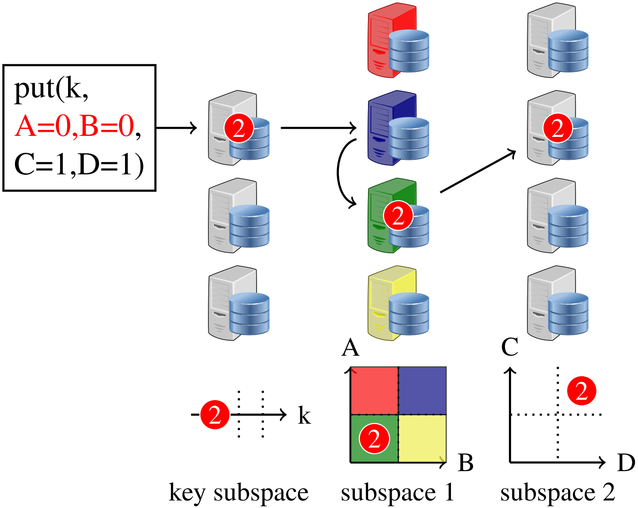
A `put` includes one node from each subspace

Value-Dependent Chaining



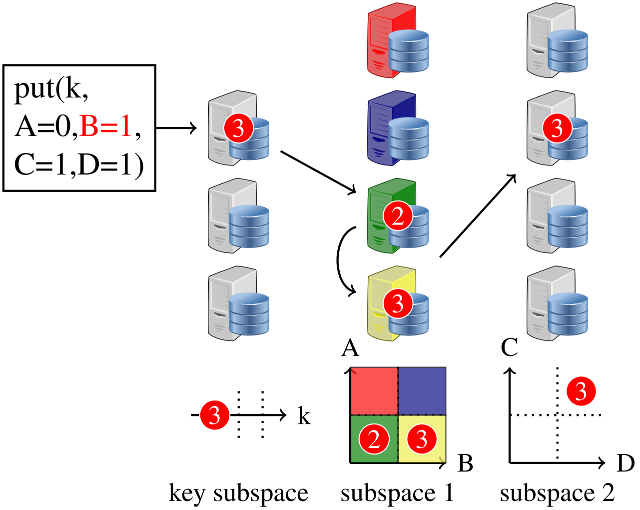
When updating an object, the value-dependent chain includes the servers which hold the old and new versions of the object

Value-Dependent Chaining



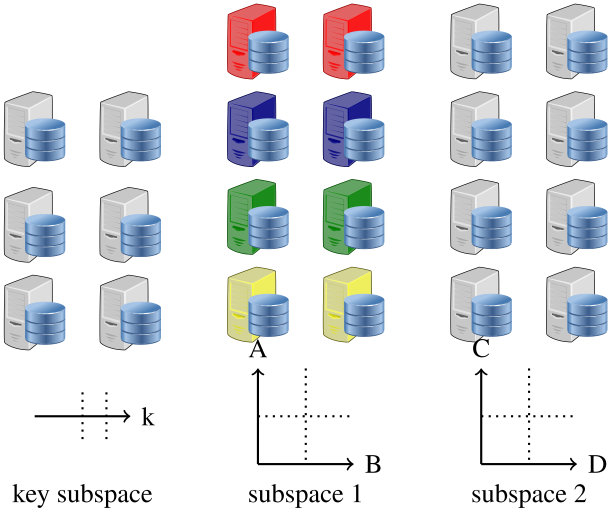
Each `put` removes all state from the previous `put`

Value-Dependent Chaining



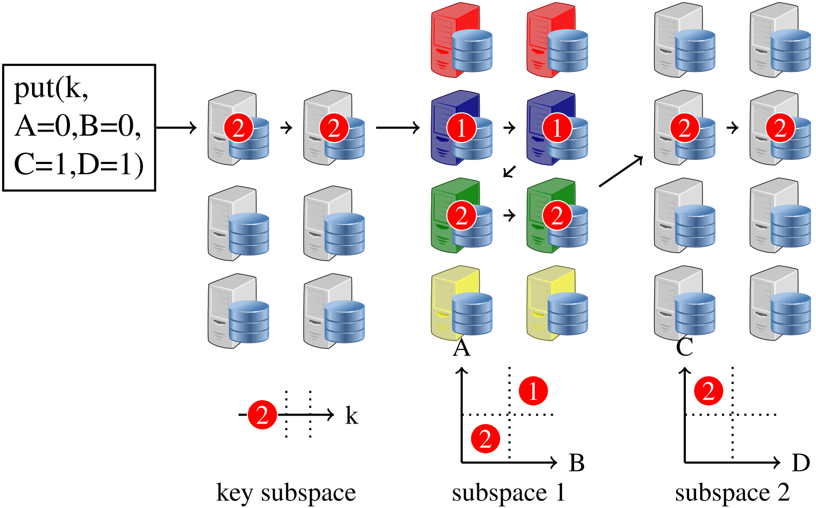
Subsequent operations involve solely the most recent nodes

Value-Dependent Chaining



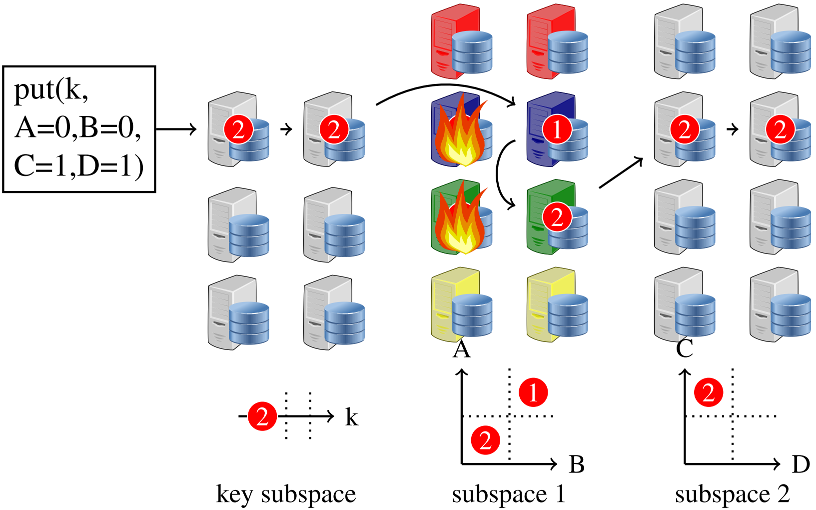
Servers are replicated in each region to provide fault tolerance

Value-Dependent Chaining



The value-dependent chain includes all replicas

Value-Dependent Chaining



Failed nodes are removed from the chain

Value-Dependent Chaining Implications

No extra mechanism is necessary to provide

- Atomicity
- Ordering
- Replication
- Relocation

Multikey Transactions

- Hyperspace hashing enables HD to locate data quickly
- Value-dependent chaining enables HD to replicate data
- And this is sufficient for many applications
- But some apps require atomic, consistent updates to multiple items

Options are:

- Spray and pray
- Use a heavyweight algorithm (e.g. Paxos) for ordering
- HyperDex Warp

Warp Properties

Warp is a novel, optimistic, concurrent, distributed algorithm for ensuring isolated updates to a data store.

- Atomic – multiple operations on multiple keys are indivisable
- Consistent – application invariants are preserved
- Isolated – one copy serializability
- Durability – all transactions are propagated to f+1 replicas

Consistency

- **Key Operations and Transactions:** One copy serializability
- **Search Consistency:** All `search` operations observe all `put` operations that completed prior to the search

Implementation

- Fully implemented system with 52,000 LOC
- Bindings for C, C++, Python, Java, Ruby, Node.JS
- Open sourced under a BSD-like license
- Active user community with many contributors
- Implementation tricks:
  - Hyperspace hashing maps objects to locations on disk
  - Paxos-based RSM maintains the hyperspace mapping

Inserting/Retrieving an Object

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

Performing A Search

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

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

Atomic Operations

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

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

Atomic Operations

```
>>> c.atomic_add('phonebook', 'jsmith',
...                 {'phone': 1})
True
```

```
>>> c.get('phonebook', 'jsmith')
{'first': 'John', 'last': 'Smith',
 'phone': 6075552049}
```

Asynchronous Operations

```
>>> d = c.async_put('phonebook', 'jsmith',
...     {'first': 'John', 'last': 'Smith',
...      'phone': 6075551024})
>>> d
<hyperclient.DeferredInsert object at 0x7f2252c
>>> do_work()
>>> d.wait()
True
```

A Space with Datastructures

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

Manipulating Lists

```
>>> c.list_rpush('socialnetwork', 'jsmith',
...                {'pending': 'bjones1'})
True

>>> c.get('socialnetwork', 'jsmith')['pending']
['bjones1']
```

Transactions

```
>>> t = c.begin_transaction()
>>> amount1 = t.get('account', 'egs')['balance'
>>> amount2 = t.get('account', 'rob')['balance'
>>> amount1 -= 1000
>>> amount2 += 1000
>>> t.put('account', 'egs', {'balance': amount1
>>> t.put('account', 'rob', {'balance': amount2
>>> t.commit()
```

Introduction

Design and Implementation

Evaluation

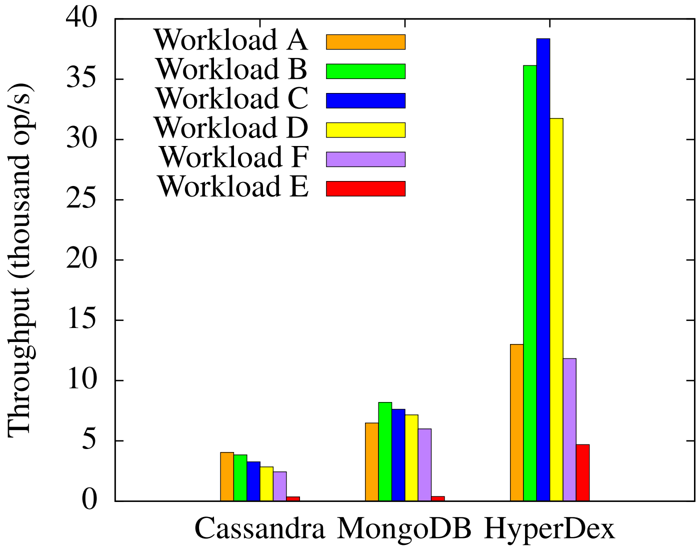
Perspective

Conclusion

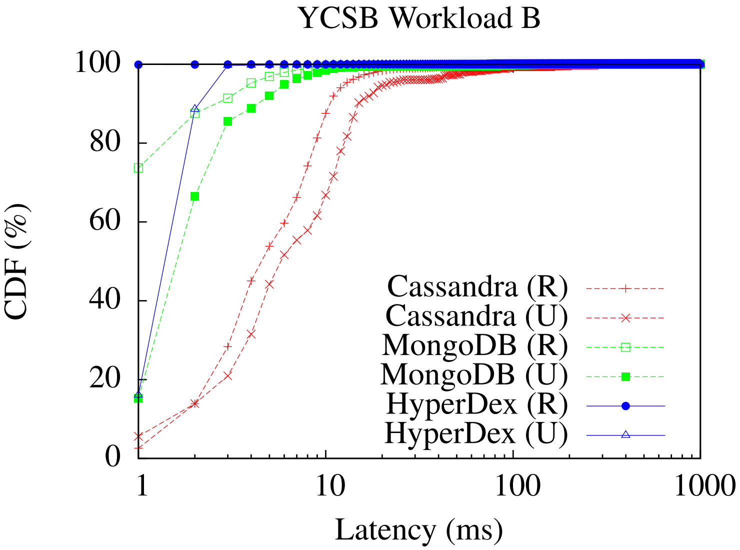
Experimental Setup

- Use the Yahoo! Cloud Serving Benchmark
- Each system makes two replicas of the data
- **MongoDB:** Writes to the client's outgoing socket buffer
- **Cassandra:** Writes to one storage node's filesystem
- **HyperDex:** Writes to both replicas in three subspaces

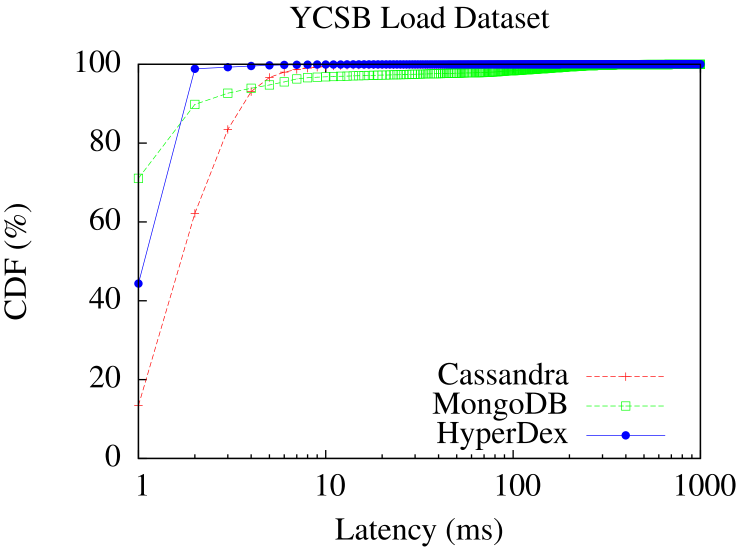
YCSB Throughput



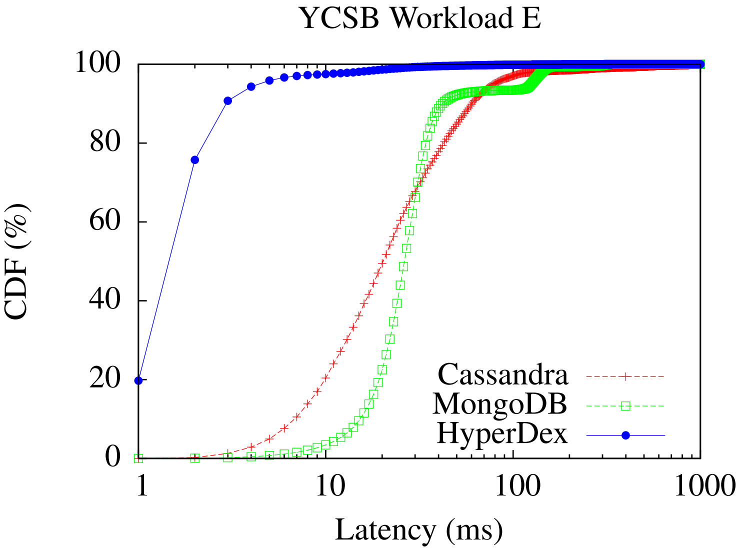
95% `get` / 5% `put` Latency



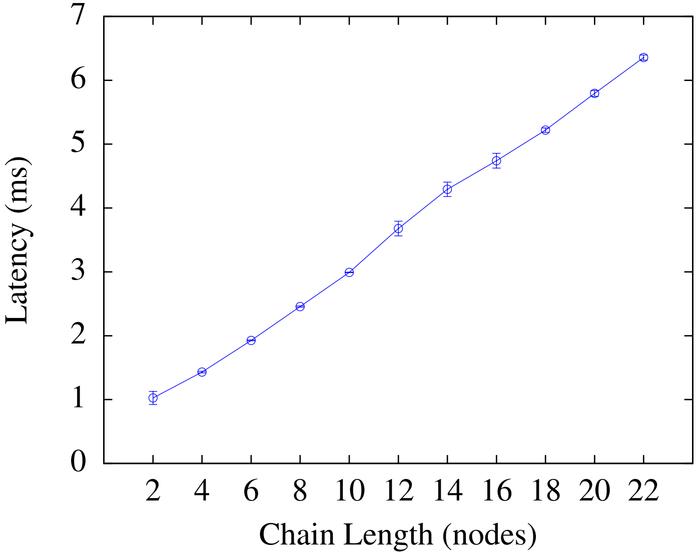
100% `put` Latency



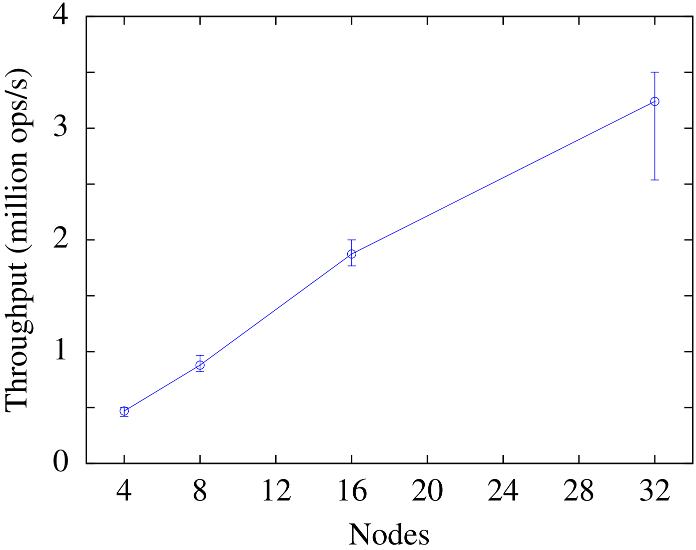
`search` Latency



Chain Length vs. Put Latency



Scalability



Performance Summary

- Outperforms other systems by 2–4$\times$ for `get/put`
  - While offering stronger consistency and fault tolerance
- Outperforms other systems by 12–13$\times$ for `search`
  - Despite operating solely on secondary attributes
- Latency for chain-operations is predictable
- Scales as resources are added

The CAP Theorem

- What CAP is simplified to:
  - You must always give something up
- What the CAP theorem really says:
  - If you cannot limit the number of faults
  - and requests can be directed to any server
  - and you insist on serving every request
  - then you cannot possibly be consistent
- Most NoSQL systems are proud to preemptively give up desirable properties like consistency in the name of CAP — even in the case of no failures
- HyperDex allows for f failures without sacrificing consistency or availability

Conclusion

- HyperDex is a next generation NoSQL system
- Novel Techniques
  - Hyperspace Hashing
  - Value-Dependent Chaining
  - ACID Transactions
- The next-generation of NoSQL systems should explore alternative designs that offer both an expanded API and strong guarantees
- `http://hyperdex.org/`

YCSB Benchmark Workloads

Name	Workload	Key Choice	Application
A	50% R 50% U	Zipf	Session Store
B	95% R 5% U	Zipf	Photo Tagging
C	100% R	Zipf	Profile Cache
D	95% R 5% I	Temporal	Status Updates
E	95% S 5% I	Zipf	Threads
F	50% R 50% R-M-U	Zipf	User Database

R = Read, U = Update, I = Insert, S = Scan/Search

Hash Functions and Load Balancing

- Out of the box, HyperDex supports hashing strings and integers
- What about non-uniform inputs?
  - Select a better hash function
  - Use forwarding pointers
  - Create multiple dimensions in the hyperspace for a single attribute
- The default hash functions work well for workloads that we've seen in practice

Experimental Setup

Lab Cluster

- 14 Machines
- Intel Xeon 2.5 GHz E5420 $\times$ 2
- 16 GB RAM
- 500 GB SATA HDD
- Debian 6.0
- Linux 2.6.32

VICCI Cluster

- 70 Machines
- Intel Xeon 2.66 GHz X5650 $\times$ 2
- 48 GB RAM
- 1 TB SATA HDD $\times$ 3
- Virtualized Fedora 12
- Linux 2.6.32

Cluster Size

- Netflix: App-specific clusters of 6-48 Cassandra instances
- Google BigTable:
  - 66% of clusters $<$ 20 tablet servers
  - 84% of clusters $<$ 100 tablet servers
  - 96% of clusters $<$ 500 tablet servers
- Justin Sheehy, Basho Inc.:
  - Typical cluster is 6-12 Riak nodes
  - Largest clusters $<$ 100 Riak nodes

Related Work

- Multi-dimensional database systems on a single host
  - Grid File, KD-Tree, Multi-dimensional BST, Quad-Tree, R-Tree, Universal B-Tree
- Distributed database systems maintain distributed indices
  - Distributed B-Tree, P-Tree, Sinfonia
- Peer-to-peer systems are only eventually consistent
  - Arpeggio, CAN, Chord, Consistent Hashing, Mercury, MURK, Pastry, SkipIndex, SWAM-V, Tapestry
- Space-filling curves suffer from the curse of dimensionality
  - MAAN, SCRAP, Squid, ZNet
- NoSQL systems/key-value stores give up search, consistency or fault-tolerance
  - CouchDB, MongoDB, Neo4j, PNUTS, Redis, TXCache, BigTable, Cassandra, COPS, Distributed Data Structures, Dynamo, Fawn KV, HBase, HyperTable, LazyBase, Masstree, Memcached, RAMCloud, Riak, SILT, Spanner, Spinnaker, TSSL, Voldemort