HyperDex: The Next Generation NoSQL Data Store
Speed, Search and ACID

Emin Gün Sirer
Robert Escriva    Bernard Wong

Bitly Betaworks Technical Talks
February 25, 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
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

↓

- 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
Introduction

Design and Implementation
Hyperspace Hashing
Value-Dependent Chaining

Evaluation

Perspective

Conclusion
Attributes map to dimensions in a multidimensional hyperspace.
Attribute values are hashed independently
Any hash function may be used

First Name

Phone Number

H("607-555-1024")

H("Neil")

H("Armstrong")

Last Name

First Name
Objects reside at the coordinate specified by the hashes

Phone Number

\[ H(“607-555-1024”) \]

First Name

\[ H(“Neil”) \]

Last Name

\[ H(“Armstrong”) \]

Neil Armstrong
First Name

Phone Number

Last Name

Neil Armstrong
Different objects reside at different coordinates

- Neil Armstrong
- Lance Armstrong
- Neil Diamond
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

- Neil Armstrong
- Lance Armstrong
- Neil Diamond

Phone Number

First Name

Last Name
All people named Neil are mapped to the yellow plane
All people named Armstrong are mapped to the gray plane

- Neil Armstrong
- Lance Armstrong
- Neil Diamond
All people named Armstrong are mapped to the gray plane.
A more restrictive search for Neil Armstrong contacts fewer servers

First Name

Phone Number

Last Name

Neil Armstrong
Lance Armstrong
Neil Diamond

HyperDex: The Next Generation NoSQL Data Store
http://hyperdex.org/
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 | a1 | a2 | a3 | a4 | a5 | ... | 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

- searches 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
Introduction

Design and Implementation
  Hyperspace Hashing
  Value-Dependent Chaining

Evaluation

Perspective

Conclusion
Replication

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

key subspace

subspace 1

subspace 2

put(k, A=1, B=1, C=1, D=1)

put(k, A=0, B=0, C=1, D=1)

put(k, A=0, B=1, C=1, D=1)
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

\[ \text{put}(k, A=0, B=0, C=1, D=1) \]

The value-dependent chain includes all replicas.
Value-Dependent Chaining

\[
\text{put}(k, A=0, B=0, C=1, D=1)
\]

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 indivisible
- 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

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

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

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

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

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

```python
>>> 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 0x7f2252d8>
>>> 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

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

>>> c.get('socialnetwork', 'jsmith')['pending'][0]
['bjones1']
```
Transactions

```python
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

Throughput (thousand op/s)

Workload A
Workload B
Workload C
Workload D
Workload F
Workload E

Cassandra  MongoDB  HyperDex
95% get / 5% put Latency

YCSB Workload B

CDF (%) vs Latency (ms)

- Cassandra (R)
- Cassandra (U)
- MongoDB (R)
- MongoDB (U)
- HyperDex (R)
- HyperDex (U)
100% put Latency

YCSB Load Dataset

CDF (%) vs. Latency (ms)

- Cassandra
- MongoDB
- HyperDex
YCSB Workload E

CDF (%) vs Latency (ms)

- Cassandra
- MongoDB
- HyperDex

Latency (ms)

CDF (%)
Chain Length vs. Put Latency

![Graph showing the relationship between chain length and put latency.](image)
Scalability

Throughput (million ops/s) vs Nodes
Performance Summary

- Outperforms other systems by 2–4× for get/put
  - While offering stronger consistency and fault tolerance
- Outperforms other systems by 12–13× 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Workload</th>
<th>Key Choice</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>50% R</td>
<td>Zipf</td>
<td>Session Store</td>
</tr>
<tr>
<td></td>
<td>50% U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>95% R</td>
<td>Zipf</td>
<td>Photo Tagging</td>
</tr>
<tr>
<td></td>
<td>5% U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>100% R</td>
<td>Zipf</td>
<td>Profile Cache</td>
</tr>
<tr>
<td>D</td>
<td>95% R</td>
<td>Temporal</td>
<td>Status Updates</td>
</tr>
<tr>
<td></td>
<td>5% I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>95% S</td>
<td>Zipf</td>
<td>Threads</td>
</tr>
<tr>
<td></td>
<td>5% I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>50% R</td>
<td>Zipf</td>
<td>User Database</td>
</tr>
<tr>
<td></td>
<td>50% R-M-U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 × 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 × 2
- 48 GB RAM
- 1 TB SATA HDD × 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