HyperDex
A New Era in High Performance Data Stores for the Cloud

Emin Gün Sirer†

Robert Escriva†  Bernard Wong‡

†Department of Computer Science
Cornell University

‡School of Computer Science
University of Waterloo

Twitter, April 27, 2012
My group recently built a new datastore called HyperDex
My group recently built a new datastore called HyperDex

- **Consistent**: linearizable. GET returns latest PUT, always
My group recently built a new datastore called HyperDex

- **Consistent**: linearizable. GET returns latest PUT, always
- **Available**: in the presence of $\leq f$ failures
My group recently built a new datastore called HyperDex

- **Consistent**: linearizable. GET returns latest PUT, always
- **Available**: in the presence of $\leq f$ failures
- **Partition-Tolerant**: for partitions with $\leq f$ nodes

Thought to be Impossible!
My group recently built a new datastore called HyperDex

- **Consistent**: linearizable. GET returns latest PUT, always
- **Available**: in the presence of $\leq f$ failures
- **Partition-Tolerant**: for partitions with $\leq f$ nodes
- **Scalable**: scales with servers
- **High-Performance**: high throughput, low variance
- **Searchable**: expressive API

Thought to be Impossible!
### DBMS Properties

- **Very strong properties**
  - Atomicity
  - Consistency
  - Isolation
  - Durability

- **But missing some critical ones**
  - Scalability
  - Performance
The NoSQL Revolution

Root Causes of DBMS problems

- DBMS architecture not amenable to large scale

NoSQL Approach

- Distribute work across many machines
- Scale the service up by adding more machines
- Abandon SQL for simpler interface
- Weaken ACID, perhaps all the way to BASE
NoSQL Systems

Web Servers

Storage Servers
NoSQL Systems
NoSQL Systems
NoSQL Systems
NoSQL Systems
NoSQL Systems
NoSQL Systems
NoSQL Systems
First name: John, Last name: Smith, Phone: (555) 123-4567
NoSQL Systems

FIRST="John" LAST="Smith", PHONE=(555) 123-4567

FIRST="John" LAST="Doe", PHONE=(555) 890-2028

FIRST="Jane" LAST="Baker", PHONE=(555) 828-3156
NoSQL Systems

FIRST="John" LAST="Smith", PHONE=(555) 123-4567
NoSQL Systems

- FIRST="John" LAST="Smith", PHONE=(555) 123-4567
- FIRST="John" LAST="Doe" PHONE=(555) 890-2028
NoSQL Systems

FIRST="John" LAST="Smith", PHONE=(555) 123-4567

FIRST="John" LAST="Doe" PHONE=(555) 890-2028
NoSQL Systems

FIRST="John" LAST="Smith", PHONE=(555) 123-4567

FIRST="John" LAST="Doe" PHONE=(555) 890-2028
NoSQL Systems

- FIRST="John" LAST="Smith", PHONE=(555) 123-4567
- FIRST="John" LAST="Doe" PHONE=(555) 890-2028
- FIRST="Jane" LAST="Baker" PHONE=(555) 828-3156
NoSQL Systems
NoSQL Systems

FIRST="John" LAST="Smith", PHONE=(555) 123-4567

FIRST="John" LAST="Doe" PHONE=(555) 890-2028

FIRST="Jane" LAST="Baker" PHONE=(555) 828-3156
NoSQL Systems

- FIRST="John" LAST="Smith", PHONE=(555) 123-4567
- FIRST="John" LAST="Doe" PHONE=(555) 890-2028
- FIRST="Jane" LAST="Baker" PHONE=(555) 828-3156
NoSQL problems

NoSQL Systems suffer from many problems

- Limited API: Can only recall by key
  - No other technology, besides NoSQL, is as proud of its lack of features
  - This is not a good thing
- Consistency and fault-tolerance are difficult to achieve
  - Inconsistent, eventually consistent
  - Not available
  - Not partition-tolerant
HyperDex

1. Design and Implementation

2. Evaluation

3. Perspective

4. Conclusions
Limitation: Expressiveness

- Efficient object retrieval requires the key
- Retrieving objects using secondary attributes requires a linear scan

Can we build a key-value store which supports efficient search?
HyperDex
- Design and Implementation
- Hyperspace Hashing

Phone Number

First Name

Last Name

John Smith
HyperDex

Design and Implementation

Hyperspace Hashing

Phone Number

First Name

Last Name

Smith

John
Hyperspace Hashing Implications

- SEARCHes are efficient
  - Every search term introduces a hyperplane
  - Sought items must lie at the intersection
  - Need only contact servers whose zones intersect the search hyperplane
- SEARCHes permit efficient range queries
- No indices!
  - No overhead for building and maintaining B-trees
  - Such distributed trees are difficult to make consistent
  - Functionality gained solely through careful placement
The volume of the hyperspace grows exponentially with the number of searchable attributes
### Data Partitioning

<table>
<thead>
<tr>
<th>k</th>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$v_3$</th>
<th>$v_4$</th>
<th>$v_5$</th>
<th>...</th>
<th>$v_{D-1}$</th>
<th>$v_{D-1}$</th>
<th>$v_D$</th>
</tr>
</thead>
</table>

The diagram illustrates a key-value data structure. Each key is associated with a subspace, where subspaces are divided into partitions $v_1$ to $v_D$. The structure is designed to optimize access and retrieval times in distributed systems.
HyperDex

Design and Implementation

Data Partitioning

![Diagram of data partitioning in HyperDex]

- Key subspace
- Subspace 0
- Subspace 1
- Subspace D

- Data partitioning diagram with key and subspaces.
HyperDex

Data Partitioning

Design and Implementation

k
v_1  v_2  v_3  v_4  v_5  \ldots  v_{D-1}  v_{D-1}  v_D

v_1  v_2  v_3

v_4  v_5

v_{D-1}  v_{D-1}  v_D

subspace 0

subspace 1

subspace D

key subspace
A single hyperspace with 9 attributes requires 512 machines to completely cover the space.

Splitting the same hyperspace into three subspaces reduces the requirement to just 24 machines.
Node failures are common at large scale, network partitions are possible

NoSQL systems replicate data

Coordinating replicas is difficult

- Option #1: Spray updates $\Rightarrow$ Fast but inconsistent
- Option #2: Use consensus protocol $\Rightarrow$ Consistent but slow

Many NoSQL systems choose Option #1 and forego consistency

- Even on a good day with no failures!

Can we replicate data so it is available and the network is resilient to partitions, without compromising consistency?
Each object is mapped to multiple nodes

Changes are applied to each object in a consistent fashion

As an object changes, so does the set of nodes to which the object maps

HyperDex employs a novel technique called *value-dependent chaining* which constructs replica sets dynamically as objects move about the hyperspace
Value-Dependent Chaining
Value-Dependent Chaining

update \( u_1 \)
update \( u_2 \)
update \( u_3 \)

key
subspace

subspace 0

subspace 1
Value-Dependent Chaining

- update $u_1$
- update $u_2$
- update $u_3$

- key
- subspace
- subspace 0
- subspace 1
Value-Dependent Chaining

h₁ → h₂
h₁ → h₃
h₁ → h₄
h₂ → h₅
h₂ → h₆
h₃ → h₅
h₃ → h₆

update u₁
update u₂
update u₃

key subspace
subspace 0
subspace 1
Value-Dependent Chaining

- update $u_1$
- update $u_2$
- update $u_3$

- key subspace
- subspace 0
- subspace 1
Value-Dependent Chaining

- Key
  - Subspace
  - Subspace 0
  - Subspace 1

- Updates
  - $u_1$
  - $u_2$
  - $u_3$

- Hash Functions
  - $h_1$
  - $h_2$
  - $h_3$
  - $h_4$
  - $h_5$
  - $h_6$
Value-Dependent Chaining

update $u_1$
update $u_2$
update $u_3$

key subspace
subspace 0
subspace 1
Value-Dependent Chaining

HyperDex
- Design and Implementation
- Replication and Consistency

update $u_1$
update $u_2$
update $u_3$

key subspace
subspace 0
subspace 1
Value-Dependent Chaining

HyperDex
- Design and Implementation
- Replication and Consistency

update $u_1$
update $u_2$
update $u_3$

subspace 0
subspace 1

key subspaces
h1
h2
h3
h4
h5
h6

replication and consistency
Consistency and Failures

- Chaining enables data to be replicated yet kept consistent
- Failed nodes are skipped
- New nodes are inserted at chain tail
- Strict ordering eliminates doubt as to which nodes have seen updates or which updates have been committed
Consistency

Key Consistency  All key-based operations are linearizable

Search Consistency  All SEARCH operations observe all PUT operations that were completed prior to the search and that are not concurrent with another PUT
Node Configuration and Management

- A logically centralized coordinator manages global state.
- Global state is specified in a configuration.
- Failure information is decided upon by the coordinator and disseminated via a new configuration.
- The coordinator is Paxos-replicated to tolerate failures.
Implementation

- Implementation contains:
  - HyperDex Server
  - HyperDex Client
  - Coordinator
  - HyperDisk
Apply the hyperspace hashing trick recursively
HyperDisk

- Apply the hyperspace hashing trick recursively

\[ k_0 \]

<table>
<thead>
<tr>
<th>Key</th>
<th>$\alpha$</th>
<th>$\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\zeta$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\eta$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Secondary Attributes</th>
</tr>
</thead>
</table>
Apply the hyperspace hashing trick recursively
Apply the hyperspace hashing trick recursively.

\[ k_0 \]

Key

Secondary Attributes

\[ \alpha \quad \beta \]
\[ \delta \quad \gamma \]
\[ \zeta \quad \eta \]
Apply the hyperspace hashing trick recursively

Key

Secondary Attributes

$\kappa_0$, $\alpha$, $\beta$, $\delta$, $\zeta$, $\eta$, $\gamma$
Apply the hyperspace hashing trick recursively
Creating a Space

```bash
$ hyperdex-coordinator-control \
   --host 127.0.0.1 \
   --port 6970
add-space << EOF
space phonebook
dimensions username, first, last, phone (int64)
key username auto 1 3
subspace first, last, phone auto 2 3
EOF
```
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'}]
```
>>> 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

```python
>>> 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']['bjones1']
```
1 Design and Implementation

2 Evaluation

3 Perspective

4 Conclusions
## 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
HyperDex Evaluation

YCSB Throughput

Throughput (thousand op/s)

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

Cassandra MongoDB HyperDex
Chain Length vs. Latency

- **X-axis**: Chain Length (nodes)
- **Y-axis**: Latency (ms)

The graph shows a linear increase in latency as the chain length increases.
Throughput (million ops/s) vs. Nodes

- Nodes: 4, 8, 12, 16, 20, 24, 28, 32
- Throughput (million ops/s): 0, 1, 2, 3, 4

Graph shows a linear increase in throughput with an increase in the number of nodes.
1 Design and Implementation
2 Evaluation
3 Perspective
4 Conclusions
What about CAP?

CAP Background
- Eric Brewer’s conjecture from 2001 PODC keynote
- Later proven by Gilbert and Lynch two classes of networks

CAP Theorem, the pop formulation
- “Consistency, Availability, Partition Tolerance: Pick any two”

The pop formulation of the theorem has been used to justify many bad system design decisions.
CAP Basics

CAP is not what you think it is

- What practitioners think CAP says:
  - you must give up something (like Lent)

- What CAP really says:
  - if you cannot constrain your faults in any way,
  - and your requests can be directed at any server,
  - and you insist on serving every request,
  - then you cannot possibly be consistent.

- No shortage of systems that preemptively give up on C, A, and P, especially C.
CAP is not Desirable

Do you really care for C, A and P as defined?

- CAP is essentially a tautology
  - In the same class as “no system can work if all of its servers are down”
- Is P, as defined, something you want?
  - Does your system have to work if every server is partitioned from every other?
- Is A, as defined, something you want?
  - Should any server really be able to serve any request?
  - Can you redirect the clients somewhere else?
- Is C something we should abandon readily?
  - Most systems cannot provide consistent answers even when there are no failures!
CAP Theorem is easily avoidable

Working around the theorem

- Constrain the failure size
- Redirect clients to majority partition
- Profit: Retain all of C, A, P
- Realistic for a modern data center

CAP is Dead, Long Live CAP

- CAP misses the point
- The real tradeoff is between C, A, and Performance
1 Design and Implementation
2 Evaluation
3 Perspective
4 Conclusions
Summary

- Hyperspace hashing and value-dependent chaining are powerful techniques.
- CAP is achievable for failures below a threshold *with high-performance*.

http://hyperdex.org/

Example app: http://gibbr.org/