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

University of Buffalo, April 13, 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
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
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!**
In the Beginning of Time...

- Data used to be managed by applications
- This led to chaos and haphazard treatment of data
Abridged History of Data Management

In the Beginning of Time...
- Data used to be managed by applications
- This led to chaos and haphazard treatment of data

Database Doctrine/Dogma was born
- Factor data management out to a DBMS
- DBMS provides ACID properties
- Interact with the server through SQL, declare intent
- A query optimizer retrieves what you want, fast
DBMS-backed Web Sites

Web Servers

DBMS
DBMS-backed Web Sites
DBMS-backed Web Sites
DBMS-backed Web Sites
DBMS-backed Web Sites
DBMS Problems

Database dogma leads to problems

- Traditional DBMSs do not scale well
  - Big Data can exceed server capacity
  - Data is growing exponentially, faster than server capacity
  - Well-publicized case-studies: Google, Amazon, Twitter, Facebook, LinkedIn, NetFlix, ...

- Not fast
  - Slow to insert data, slow to build indices
  - JOINs and nested queries pose problems for query optimizers
  - Not even *predictably* slow performance
  - Father does not always know best!
The NoSQL Revolution

Root Causes of DBMS problems

- DBMS architecture not amenable to large scale
- SQL puts query planning and execution burden on DBMS
- ACID properties are overkill for many applications

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

...  

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

...
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) 890-1928
- FIRST="Jane" LAST="Baker" PHONE=(555) 828-3156
NoSQL Systems

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

FIRST="John" LAST="Doe" PHONE=(555) 890-1928
NoSQL Systems

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

FIRST="John" LAST="Doe" PHONE=(555) 890-1928
NoSQL Systems

- FIRST="John" LAST="Smith", PHONE=(555) 123-4567
- FIRST="John" LAST="Doe" PHONE=(555) 890-1928
- FIRST="Jane" LAST="Baker" PHONE=(555) 828-3156
FIRST="John" LAST="Smith", PHONE=(555) 123-4567
FIRST="John" LAST="Doe" PHONE=(555) 890-1928
FIRST="Jane" LAST="Baker" PHONE=(555) 828-3156
NoSQL Systems

FIRST="John" LAST="Smith", PHONE=(555) 123-4567
FIRST="John" LAST="Doe" PHONE=(555) 890-1928
FIRST="Jane" LAST="Baker" PHONE=(555) 828-3156
1. Design and Implementation

2. Evaluation

3. Perspective

4. Conclusions
NoSQL Problem #1

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
First Name: John
Last Name: Smith
Phone Number: 

Diagram:

- First Name axis
- Last Name axis
- Phone Number axis
John Smith

Phone Number

Last Name

First Name

Smith
HyperDex

Design and Implementation

Hyperspace Hashing

Phone Number

First Name

Last Name
### 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

- HyperDex
- Design and Implementation

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

- k
- v_1, v_2, v_3, v_4, v_5, \ldots
- v_D-1, v_D

- k
- v_1, v_2, v_3
- v_4, v_5

- v_D-1, v_D

- v_D-1, v_D

- v_D

- v_D-1, v_D

- v_D

- v_D
### Without Partitioning

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

### With Partitioning

Splitting the same hyperspace into three subspaces reduces the requirement to just 24 machines.
NoSQL Problem #2

Limitation: Fault-tolerance, Availability and Partition-Tolerance

- 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
- Extant 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?
HyperDex’s Replication Protocol

- 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

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

key subspace

subspace 0

subspace 1
Value-Dependent Chaining

update $u_1$ → $h_1$ → $h_2$ → $h_3$
update $u_2$ → $h_2$
update $u_3$ → $h_1$

$h_4$ → $h_5$ → $h_6$

key

subspace

subspace 0

subspace 1
Value-Dependent Chaining

- **h₁**: update u₁
- **h₂**: update u₂
- **h₃**: update u₃

- **h₄**: key
- **h₅**: subspace 0
- **h₆**: subspace 1

- Subspace 0 is connected to subspace 1 via dashed lines.
Value-Dependent Chaining
Value-Dependent Chaining

update $u_1$ → h_1
update $u_2$ → h_2
update $u_3$ → h_3

h_4 → h_5 → h_6

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

Key

Subspace

Update u1
Update u2
Update u3
Value-Dependent Chaining

Key

Subspace

Subspace 0

Subspace 1

Update u₁
Update u₂
Update u₃

h₁
h₂
h₃

h₄
h₅
h₆
Value-Dependent Chaining

- **key subspace**
- **subspace 0**
- **subspace 1**

- update $u_1$
- update $u_2$
- update $u_3$
Value-Dependent Chaining

Key:
- update $u_1$
- update $u_2$
- update $u_3$

Hyperspace:
- $h_1$
- $h_2$
- $h_3$
- $h_4$
- $h_5$
- $h_6$

Subspaces:
- Subspace 0
- Subspace 1
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
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

- ~29,800 lines of C++
- Implementation contains:
  - HyperDisk
  - Hyperspace Hashing
  - HyperDex Server
  - HyperDex Client
  - Coordinator
HyperDisk

"HyperDex Design and Implementation"

Implementation

Diagram illustrating the relationship between key (k₀) and secondary attributes (α, β, δ, γ, ζ, η).
HyperDisk

- Design and Implementation
- Implementation

HyperDex

\[ k_0 \]

\[ \begin{array}{c|c|c}
\alpha & \bullet & \beta \\
\hline
\delta & & \\
\hline
\zeta & & \\
\hline
\eta & & \\
\end{array} \]

Key

Secondary Attributes
HyperDisk

HyperDex
Design and Implementation
Implementation

\[ \alpha \cdot \beta \]
\[ \delta \]
\[ \zeta \]
\[ \eta \]

[Diagram showing a grid with keys and secondary attributes]
Hyperspace Hashing

\[
\begin{align*}
  h(v_3) &= 01101110 \\
  h(v_2) &= 01100010 \\
  h(v_1) &= 10010100 \\

  h(\langle v_1, v_2, v_3 \rangle) &= 10001101
\end{align*}
\]
## 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 E
- Workload F

Cassandra MongoDB HyperDex
HyperDex Evaluation

YCSB Workload B Latency

CDF (%) vs. Latency (ms)

- Cassandra (R)
- Cassandra (U)
- MongoDB (R)
- MongoDB (U)
- HyperDex (R)
- HyperDex (U)
What about CAP?

**CAP Background**

- Eric Brewer’s conjecture from 2001 PODC keynote
- Later proven by Fischer 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 and misused to justify many bad system design decisions.
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?
  - If you are partitioned, do you sell tickets from both halves of your datacenter?
  - If you have an axe in your head, do you show up to work?
- Is A, as defined, something you want?
  - Should any server really be able to serve any request?
  - Isn’t it better to 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
Design and Implementation

Evaluation

Perspective

Conclusions
Hyperspace hashing is a novel way to organize data
HyperDex realizes this design efficiently
CAP is achievable for failures below a threshold
with high-performance

hyperdex-team@systems.cs.cornell.edu
http://hyperdex.org/