MapReduce & HyperDex

Kathleen Durant PhD
Lecture 21 CS 3200
Northeastern University

Distributing Processing Mantra

- Scale “out,” not “up.”
- Assume failures are common.
- Move processing to the data.
- Process data sequentially and avoid random access.
- Hide system-level details from the application developer.
- Incorporate seamless scalability.

Drivers to MapReduce

- Our ability to store data is fast overwhelming our ability to process what we store
  - So you can write it you just can't use it for any calculations
- Increases in capacity are outpacing improvements in bandwidth
  - So you can write it you just can't read it back in a reasonable time

Introduction to Parallelization

- Writing algorithms for a cluster
  - On the order of 10,000 or more machines
  - Failure or crash is not an exception, but common phenomenon
  - Parallelize computation
  - Distribute data
  - Balance load
- Makes implementation of conceptually straightforward computations challenging

MapReduce

- Wanted: A model to express computation while hiding the messy details of the execution
- Inspired by map and reduce primitives in functional programming
  - Apply map to each input record to create a set of intermediate key-value pairs
  - Apply reduce to all values that share the same key (like GROUP BY)
- Automatically parallelized
- Re-execution as primary mechanism for fault tolerance

What is MapReduce?

- Programming model for expressing distributed computations on massive amounts of data

AND

- An execution framework for large-scale data processing on clusters of commodity servers

Typical MapReduce Application

MAP

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results

REDUCE

- Aggregate intermediate results
- Generate final outcome

Programming Model

- Transforms set of input key-value pairs to set of output key-value pairs
  - Map function written by user
  - Map: (k1, v1) → list (k2, v2)
  - MapReduce library groups all intermediate pairs with same key together
- Reduce written by user
  - Reduce: (k2, list (v2)) → list (v2)
  - Usually zero or one output value per group
  - Intermediate values supplied via iterator (to handle lists that do not fit in memory)

Execution Framework

- Handles scheduling of the tasks
  - Assigns workers to maps and reduce tasks
  - Handles data distribution
    - Moves the process to the data
  - Handles synchronization
    - Gathers, sorts and shuffles intermediate data
  - Handles faults
    - Detects worker failures and restarts
  - Understands the distributed file system

EXAMPLE: Count occurrences of each word in a document collection

```
Map( String key,
            String value ):
  // key: document name
  // value: document contents
  for each word w in value:
   EmitIntermediate( w, "1“ );
```

```
Reduce( String key,
            Iterator values ):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
     result += ParseInt( v );
     Emit( AsString(result) );
```

Distributing work to nodes

- Focuses on large clusters
  - Relies on existence of reliable and highly available distributed file system
- Map invocations
  - Automatically partition input data into M chunks (16-64 MB typically)
  - Chunks processed in parallel
- Reduce invocations
  - Partition intermediate key space into R pieces, e.g., using hash(key) mod R
- Master node controls program execution

Dealing with failing nodes

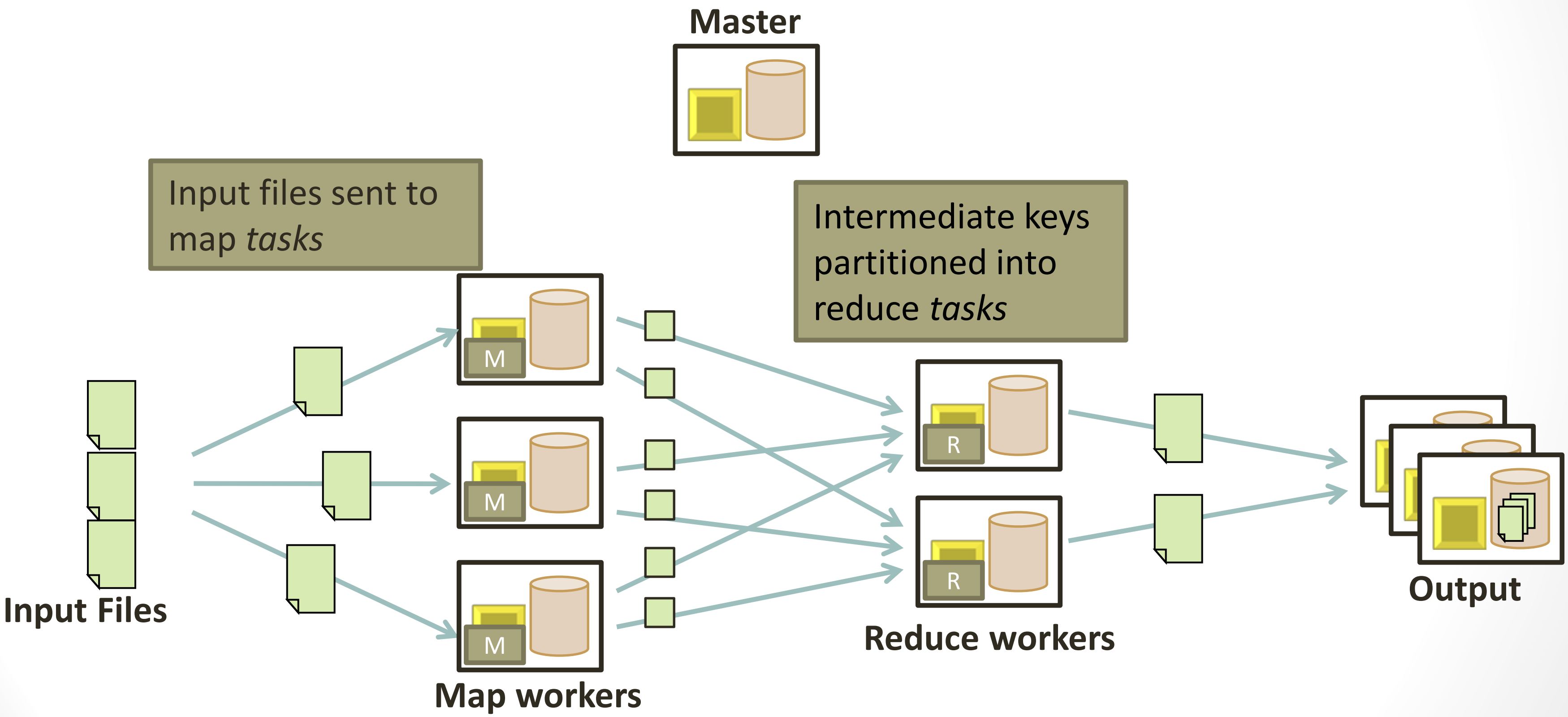
- Master monitors tasks on mappers and reducers: idle, in progress, completed
- Worker failure (common)
  - Master pings workers periodically
  - No response => assumes worker failed
    - Resets worker's map tasks, completed or in progress, to idle state (tasks now available for scheduling on other workers)
      - Completed tasks only on local disk, hence inaccessible
    - Same for reducer's in-progress tasks
      - Completed tasks stored in global file system, hence accessible
  - Reducers notified about change of mapper assignment
- Master failure (unlikely)
  - Checkpointing or simply abort computation

Other considerations

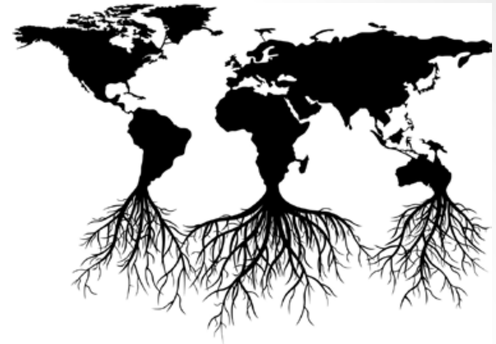
- Conserve network bandwidth (“Locality optimization”)
  - Distributed file system assigns data chunks to local disks
  - Schedule map task on machine that already has a copy of the chunk, or one “nearby”
- Choose M and R much larger than number of worker machines
  - Load balancing and faster recovery (many small tasks from failed machine)
  - Limitation: O(M+R) scheduling decisions and O(M*R) in-memory state at master
  - Common choice: M so that chunk size is 16-64 MB, R a small multiple of number of workers
- Backup tasks to deal with machines that take unusually long for last few tasks
  - For in-progress tasks when MapReduce near completion

MapReduce

- Execution flow



Map



- Interface
  - Input: <in_key, in_value> pair => <url, content>
  - Output: list of intermediate <key, value> pairs
    => list of <word, url>

key = http://url0.com
value = “every happy family is alike.”

<every, http://url0.com>
<happy, http://url0.com>
<family, http://url0.com>
…

map()

key = http://url1.com
value = “every unhappy family is unhappy in its own way.”

<every, http://url1.com>
<unhappy, http://url1.com>
<family, http://url1.com>
…

Map Input: <url, content>

Map Output: list of <word, url>

Shuffle

- MapReduce system
  - Collects outputs from all *map* executions
  - Groups all intermediate values by the same key

<every, http://url0.com>
<happy, http://url0.com>
<family, http://url0.com>
…

<every, http://url1.com>
<unhappy, http://url1.com>
<family, http://url1.com>
…

Map Output: list of <word, url>

every -> http://url0.com
http://url1.com

happy -> http://url0.com

unhappy ->
http://url1.com

family -> http://url0.com
http://url1.com

Reduce Input: <word, list of urls>

Reduce

- Interface
  - Input: <out_key, list of intermediate_value>
  - Output: <out_key, out_value>

Reduce Input	reduce()	Reduce Output
every -> http://url0.com http://url1.com	→	<every, “http://url0.com, http://url1.com”>
happy -> http://url0.com	→	<happy, “http://url0.com”>
unhappy -> http://url1.com	→	<unhappy, “http://url1.com”>
family -> http://url0.com http://url1.com	→	<family, “http://url0.com, http://url1.com” >

Reduce Input: <word, list of urls>

Reduce Output: <word, string of urls>

Parallel Database

- SQL specifies what to compute, not how to do it
  - Perfect for parallel and distributed implementation
  - “Just” need an optimizer that can choose best plan in given parallel/distributed system
  - Cost estimate includes disk, CPU, and network cost
- Recent benchmarks show parallel DBMS can significantly outperform MapReduce
  - But many programmers prefer writing Map and Reduce in familiar PL (C++, Java)
- Recent trend: High-level PL for writing MapReduce programs with DBMS-inspired operators

mySQL

```sql
SELECT
    Dim1, Dim2,
    SUM(Measure1) AS MSum,
    COUNT(*) AS RecordCount,
    AVG(Measure2) AS MAvg,
    MIN(Measure1) AS MMin
    MAX(CASE
      WHEN Measure2 < 100
      THEN Measure2
    END) AS MMax
FROM DenormAggTable
WHERE (Filter1 IN ('A','B'))
    AND (Filter2 = 'C')
    AND (Filter3 > 123)
GROUP BY Dim1, Dim2
HAVING (MMin > 0)
ORDER BY RecordCount DESC
LIMIT 4, 8
```

1. Grouped dimension columns are pulled out as keys in the map function, reducing the size of the working set.
2. Measures must be manually aggregated.
3. Aggregates depending on record counts must wait until finalization.
4. Measures can use procedural logic.
5. Filters have an ORM/ActiveRecord-looking style.
6. Aggregate filtering must be applied to the result set, not in the map/reduce.
7. Ascending: 1; Descending: -1

MongoDB

```javascript
db.runCommand({
mapreduce: "DenormAggCollection",
query: {
    filter1: { '$in': [ 'A', 'B' ] },
    filter2: 'C',
    filter3: { '$gt': 123 }
  },
map: function() { emit(
    { d1: this.Dim1, d2: this.Dim2 },
    { msum: this.measure1, recs: 1, mmin: this.measure1,
      mmax: this.measure2 < 100 ? this.measure2 : 0 }
  );},
reduce: function(key, vals) {
    var ret = { msum: 0, recs: 0, mmin: 0, mmax: 0 };
    for(var i = 0; i < vals.length; i++) {
      ret.msum += vals[i].msum;
      ret.recs += vals[i].recs;
      if(vals[i].mmin < ret.mmin) ret.mmin = vals[i].mmin;
      if((vals[i].mmax < 100) && (vals[i].mmax > ret.mmax))
        ret.mmax = vals[i].mmax;
    }
    return ret;
  },
finalize: function(key, val) {
    val.mavg = val.msum / val.recs;
    return val;
  },
out: 'result1',
verbose: true
});
db.result1.
  find({ mmin: { '$gt': 0 } }).
  sort({ recs: -1 }).
  skip(4).
  limit(8);
```

Revision 4, Created 2010-03-06
Rick Osborne, rickosborne.org

My SQL vs. MongoDB

SELECT ‘goalType’,
SUM(distancekm) as ‘totalkm’,
COUNT(*) AS ‘workouts’,
count(powerSOngAlbum) as “soungcount’,
avg(distancekm) as ‘avgkm’, max(distancekm) as maxkm,
Min(distancekm) as minkm from workouts group by goaltype;

```
db.runCommand({ mapreduce: "workouts", map: function () { emit( this.goalType, { '_cfcount': 1, 'distancekm_cfsum': isNaN(this.distancekm) ? null : this.distancekm, 'distancekm_cfnum': isNaN(this.distancekm) ? 0 : 1, 'powerSongAlbum_cfcount': (this.powerSongAlbum == null) ? 0 : 1, 'distancekm_cfmax': isNaN(this.distancekm) ? null : this.distancekm, 'distancekm_cfmin': isNaN(this.distancekm) ? null : this.distancekm } ); }, reduce: function (key,vals) { var ret = { 'distancekm_cfmax': null, 'distancekm_cfsum': null, 'distancekm_cfmin': null, 'distancekm_cfnum': 0, 'powerSongAlbum_cfcount': 0, '_cfcount': 0 }; for(var i = 0; i < vals.length; i++) { var v = vals[i]; ret['distancekm_cfnum'] += v['distancekm_cfnum']; if(!isNaN(v['distancekm_cfmax'])) ret['distancekm_cfmax'] = (ret['distancekm_cfmax'] == null) ? v['distancekm_cfmax'] : (ret['distancekm_cfmax'] > v['distancekm_cfmax']) ? ret['distancekm_cfmax'] : v['distancekm_cfmax']; ret['_cfcount'] += v['_cfcount']; if(!isNaN(v['distancekm_cfmin'])) ret['distancekm_cfmin'] = (ret['distancekm_cfmin'] == null) ? v['distancekm_cfmin'] : (v['distancekm_cfmin'] > ret['distancekm_cfmin']) ? ret['distancekm_cfmin'] : v['distancekm_cfmin']; ret['powerSongAlbum_cfcount'] += v['powerSongAlbum_cfcount']; if(!isNaN(v['distancekm_cfsum'])) ret['distancekm_cfsum'] = v['distancekm_cfsum'] + (ret['distancekm_cfsum'] == null ? 0 : ret['distancekm_cfsum']); } return ret; }, finalize: function (key,val) { return { 'totalkm' : val['distancekm_cfsum'], 'workouts' : val['_cfcount'], 'songcount' : val['powerSongAlbum_cfcount'], 'avgkm' : (isNaN(val['distancekm_cfnum']) || isNaN(val['distancekm_cfsum'])) ? null : val['distancekm_cfsum'] / val['distancekm_cfnum'], 'maxkm' : val['distancekm_cfmax'], 'minkm' : val['distancekm_cfmin'] }; }, out: "s2mr", verbose: true });
```

Database gurus have spoken out against MapReduce

Dave DeWitt, Michael Stonebraker

http://rickosborne.org/blog/2010/02/yes-virginia-thats-automated-sql-to-mongodb-mapreduce/

MapReduce Summary

- MapReduce = programming model that hides details of parallelization, fault tolerance, locality optimization, and load balancing
- Simple model, but fits many common problems
- Implementation on cluster scales to 1000s of machines and more
  - Open source implementation, Hadoop, is available
- Parallel DBMS, SQL are more powerful than MapReduce and similarly allow automatic parallelization of “sequential code”
  - Never really achieved mainstream acceptance or broad open-source support like Hadoop
- Recent trend: simplify coding in MapReduce by using DBMS ideas
  - (Variants of) relational operators and BI being implemented on top of Hadoop

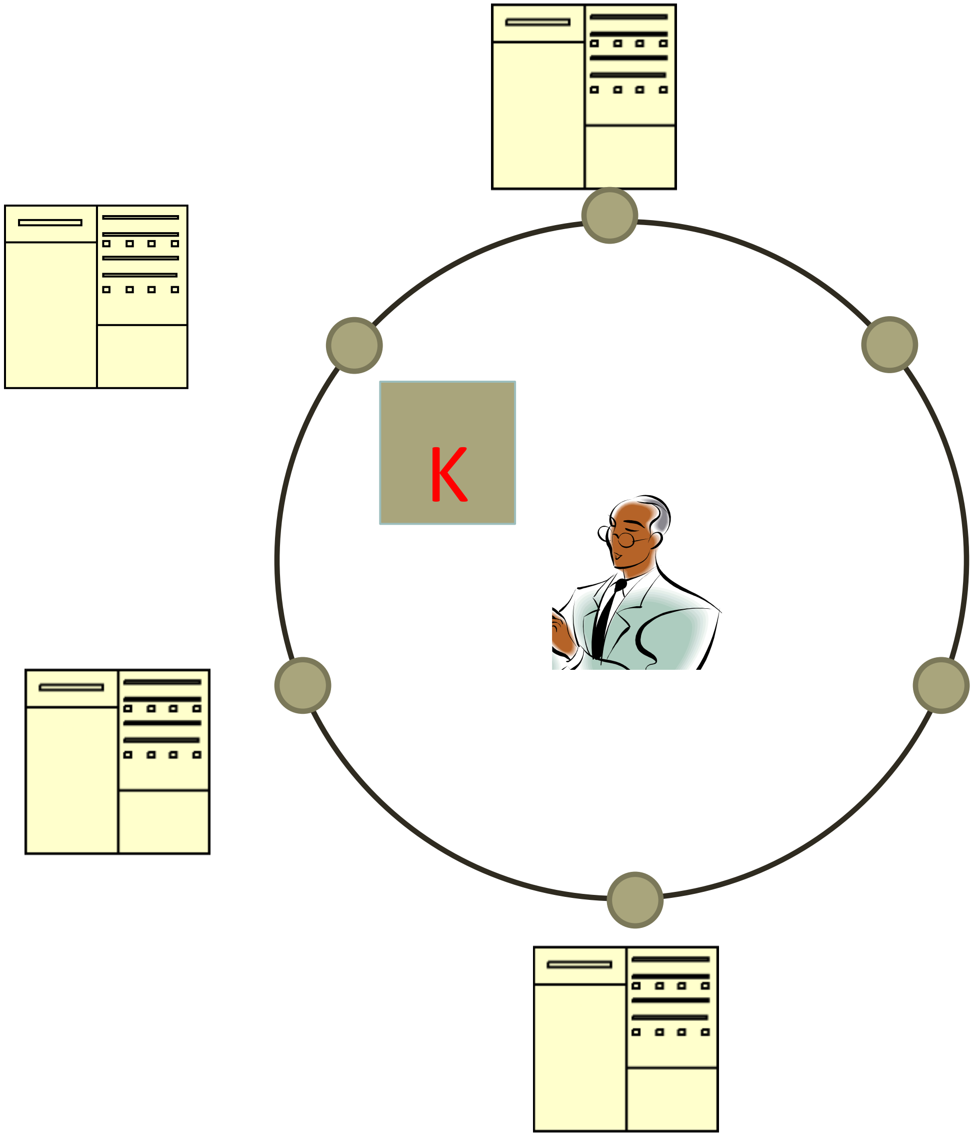
HyperDex

Adapted from
Hyperdex a Distributed, Searchable Key-value Store
Robert Escriva, Bernard Wong, Emin Gun Sirer
ACM SIGCOMM Conference, August 14, 2012

CAP Review

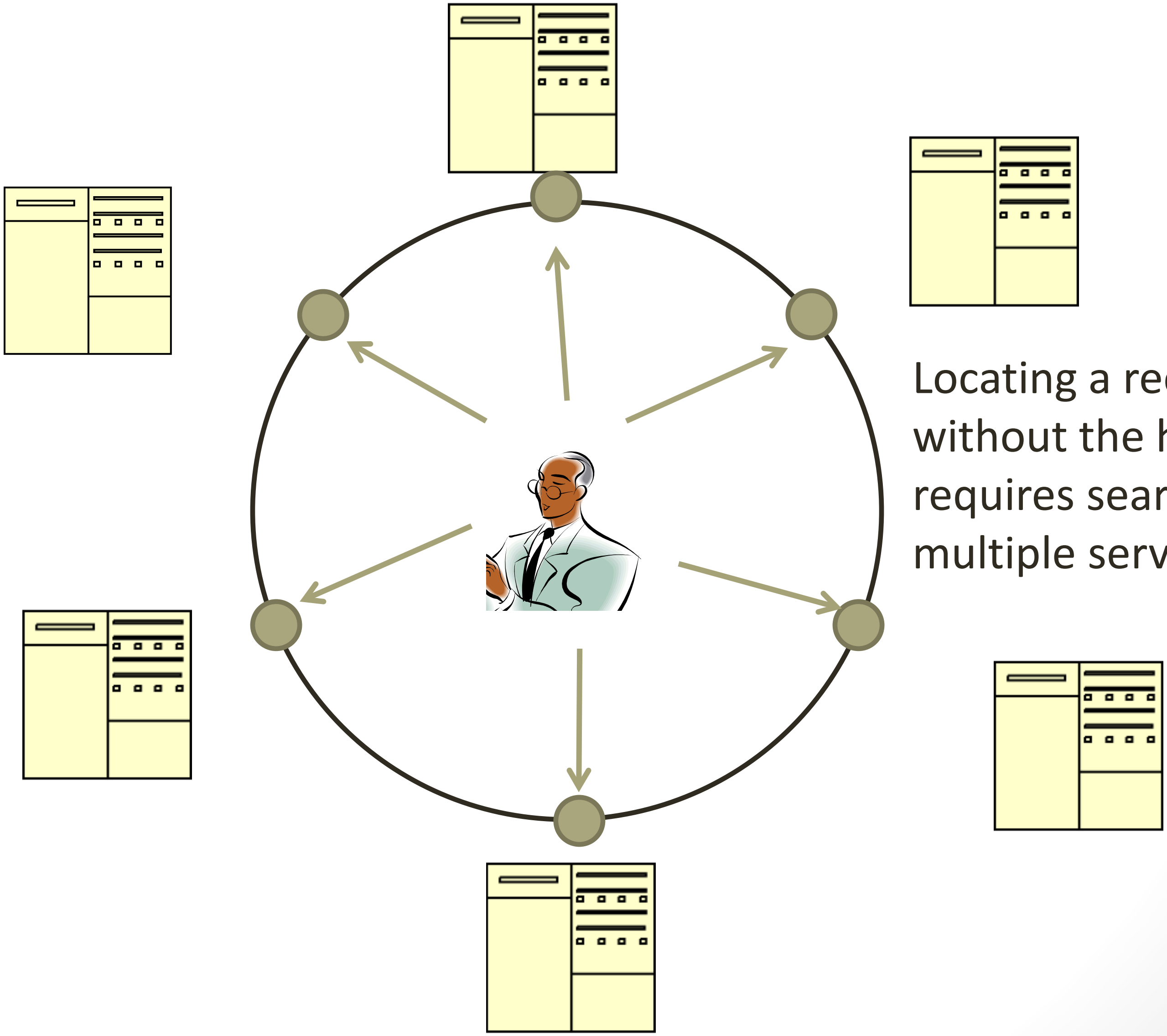
- Strong Consistency : all clients see the same view, even in the presence of updates
- High Availability : all clients can find some replica of the data, even in the presence of failures
- Partition-tolerance: the system properties hold even when the system is partitioned or not fully operational

Typical NoSQL architecture



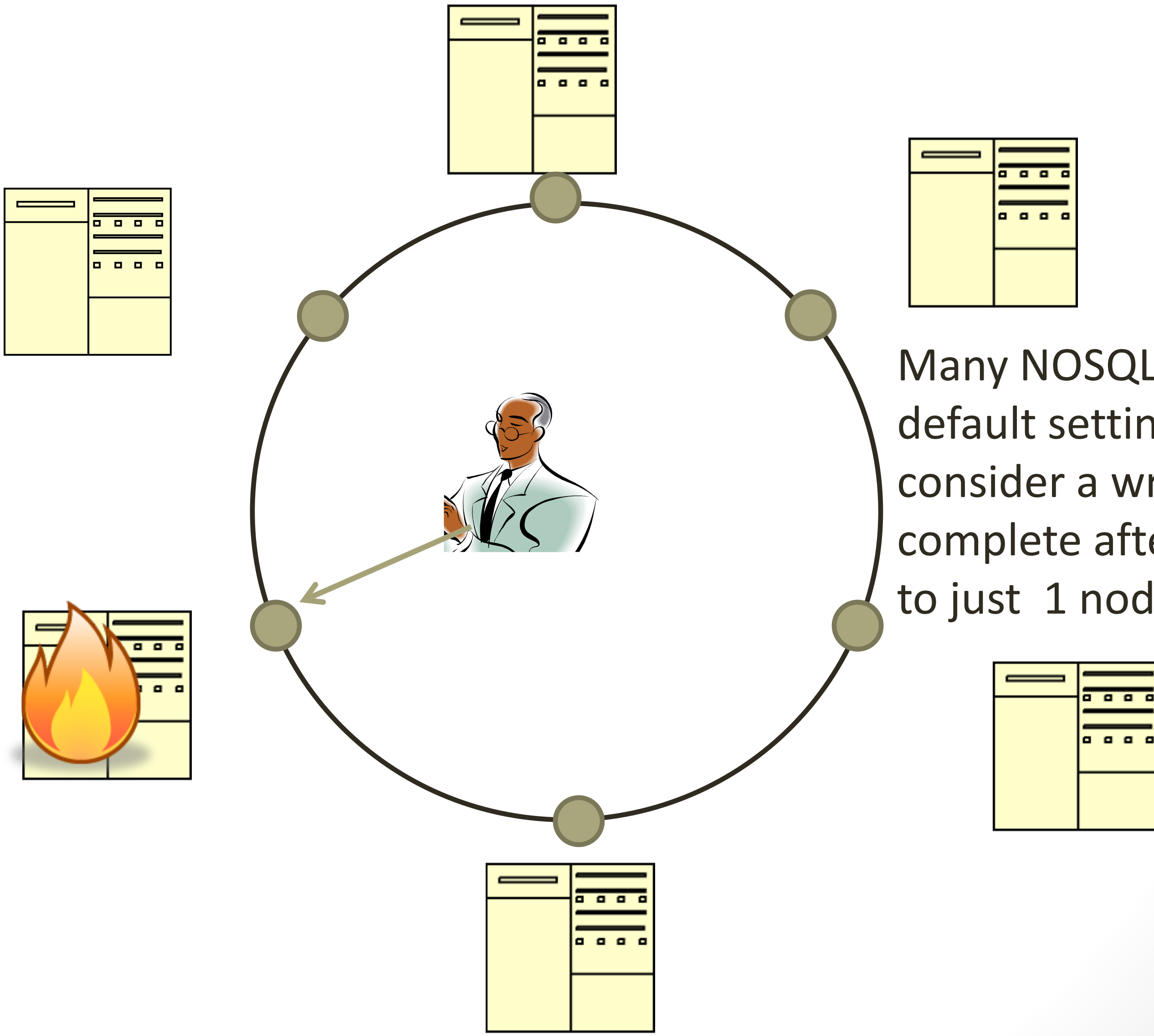
Hashing function maps each key to a server (node)

The search problem: No Hash key



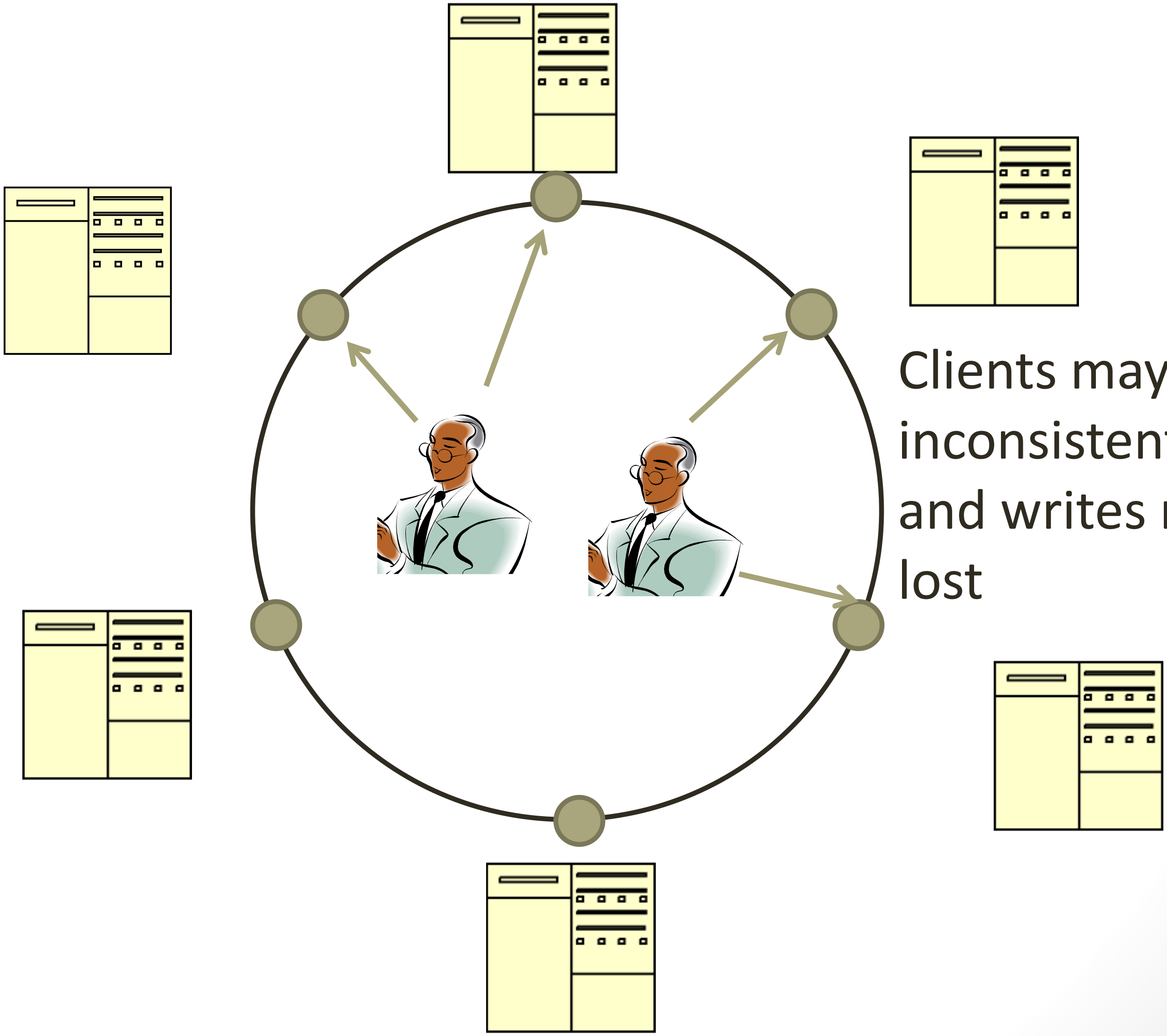
Locating a record without the hash key requires searching multiple servers

The Fault tolerance problem



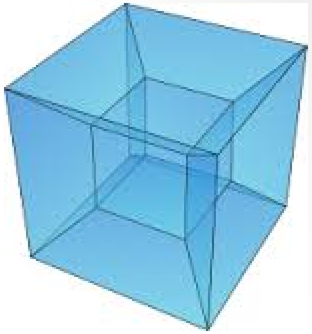
Many NOSQL system's default settings consider a write complete after writing to just 1 node

The consistency problem



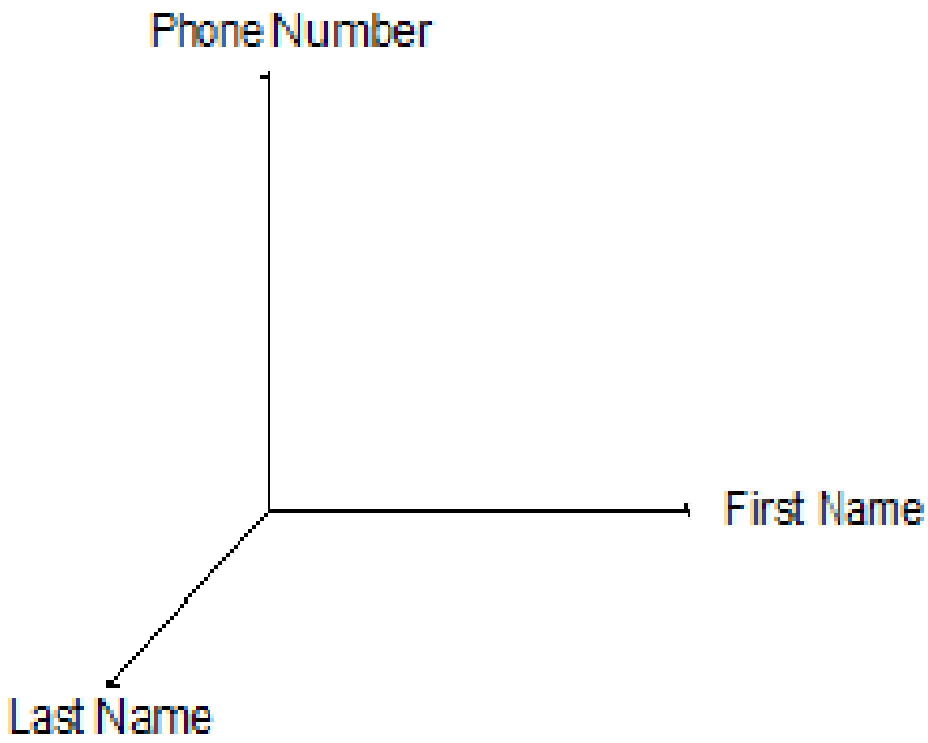
Clients may read inconsistent data and writes may be lost

HyperDex Key Points

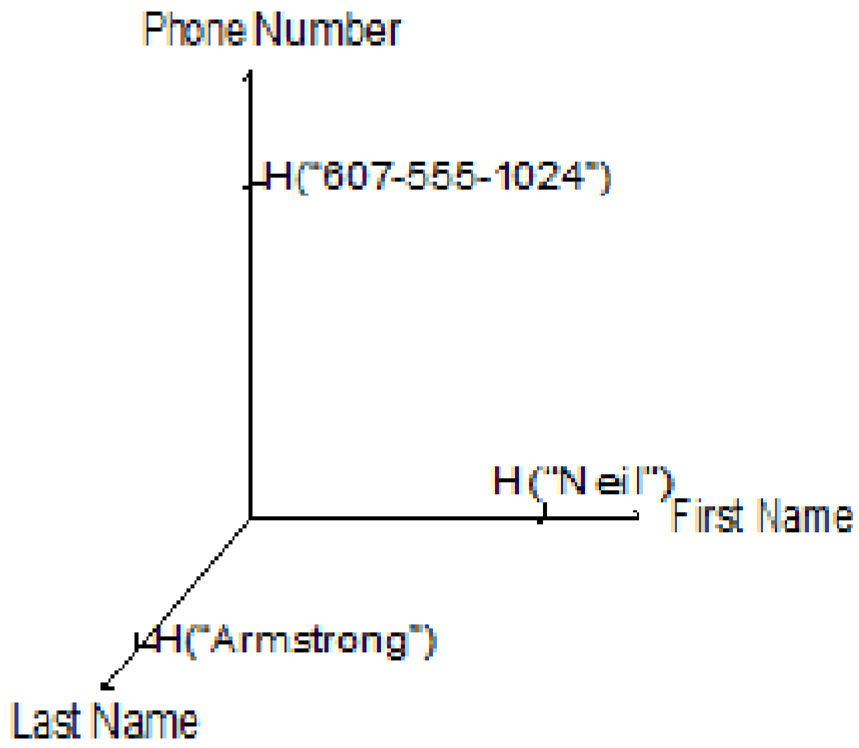


- Maps records to a Hypercube Space
  - object's key are stored in a dedicated one-dimensional subspace for efficient lookup
  - only need to contact the servers which match the regions of the hyperspace assigned for the search attributes
- Value-dependent chaining
  - Keeps replicas consistent without heavy overhead from coordination of servers
    - Uses the hypercube space
  - Appoints a point leader that contains the most recent update of a record
    - Other replicas are updated from the point leader

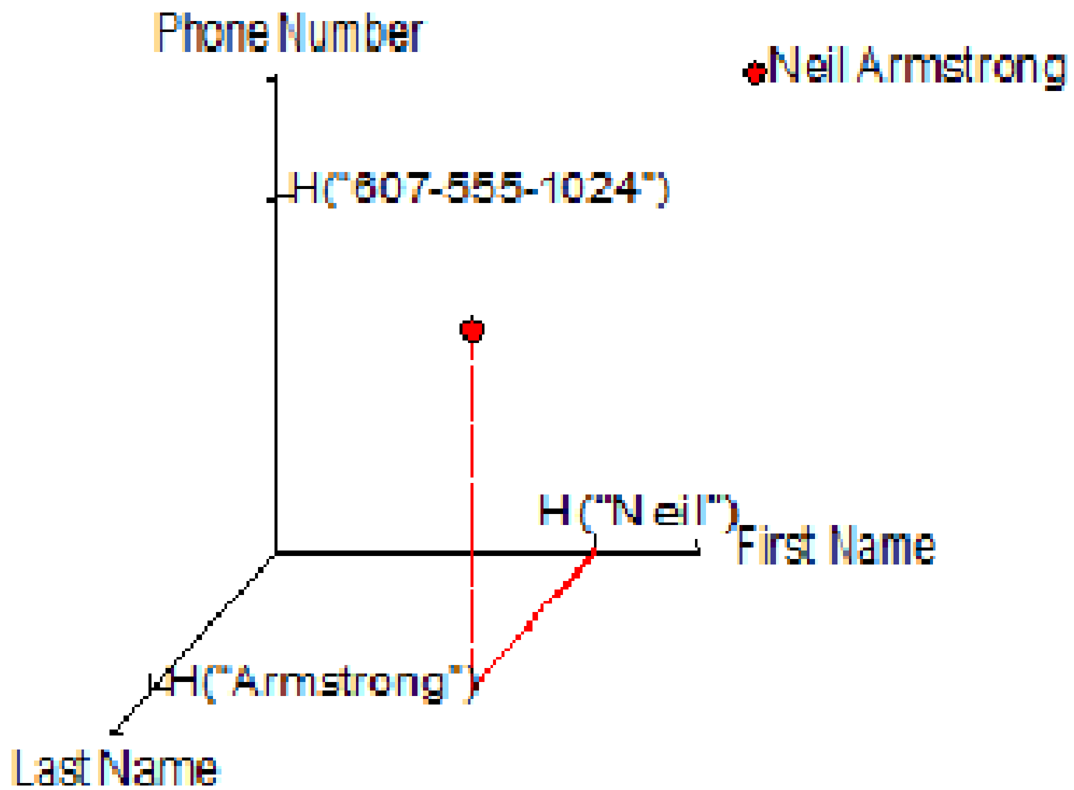
Attributes map to dimensions in a multidimensional hyperspace



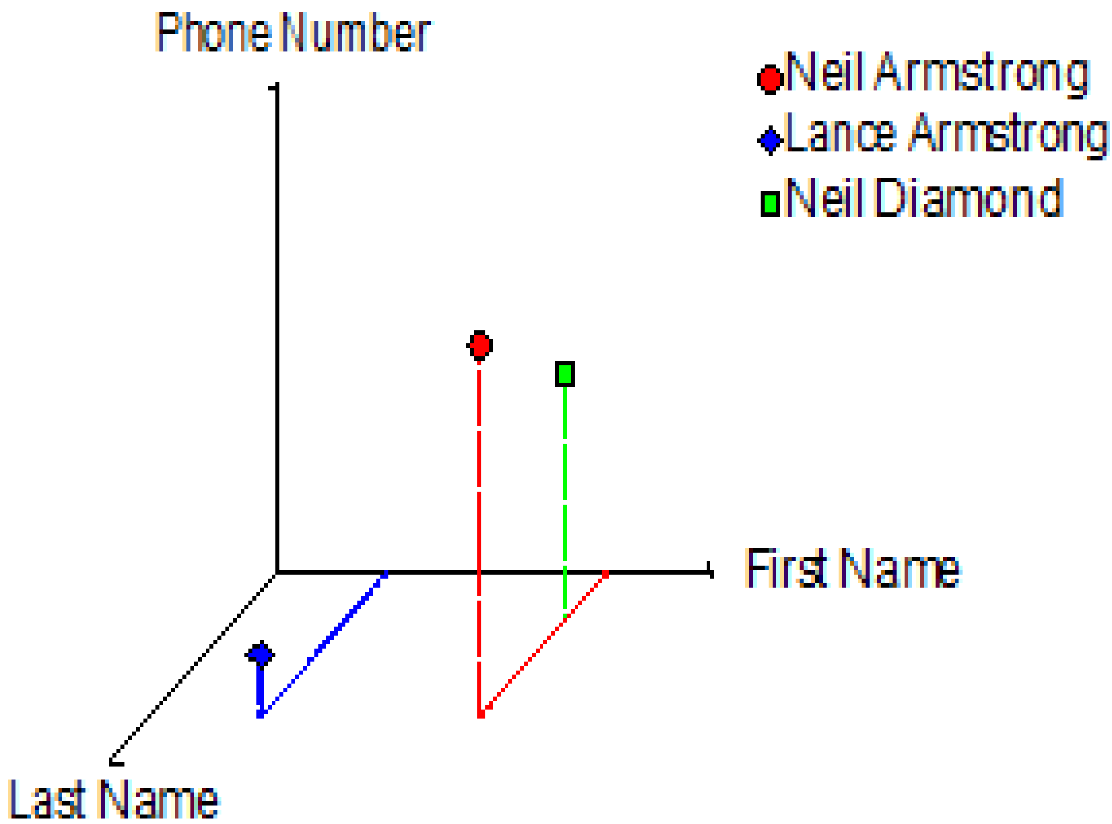
Attributes values are hashed independently
Any hash function may be used



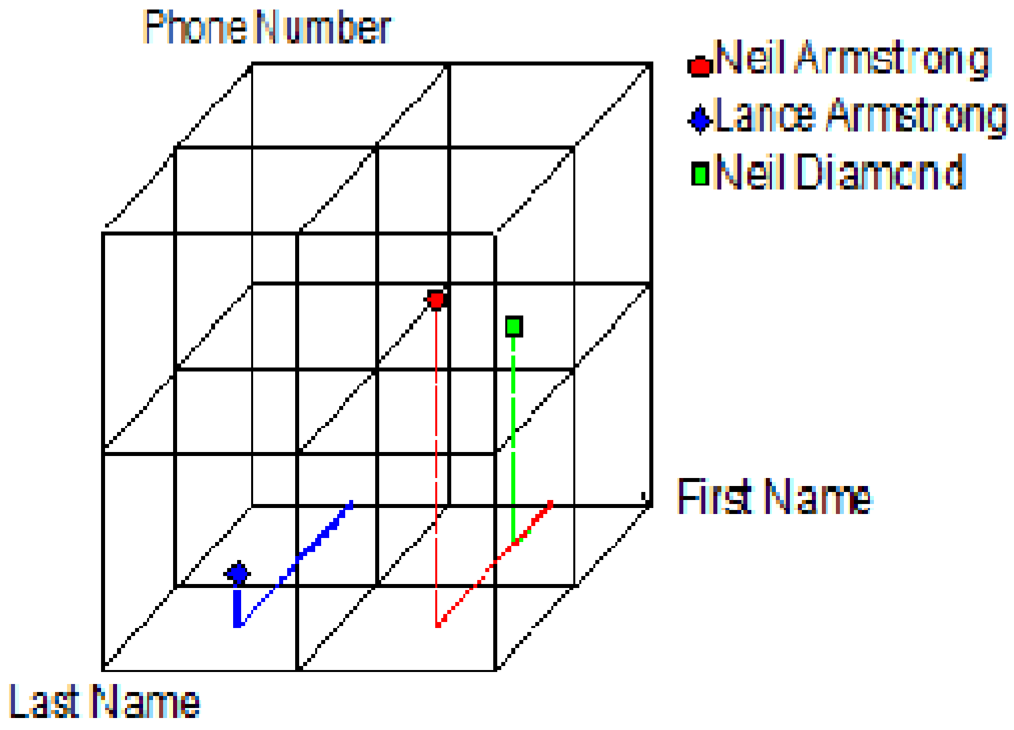
Objects reside at the coordinate specified by the hashes



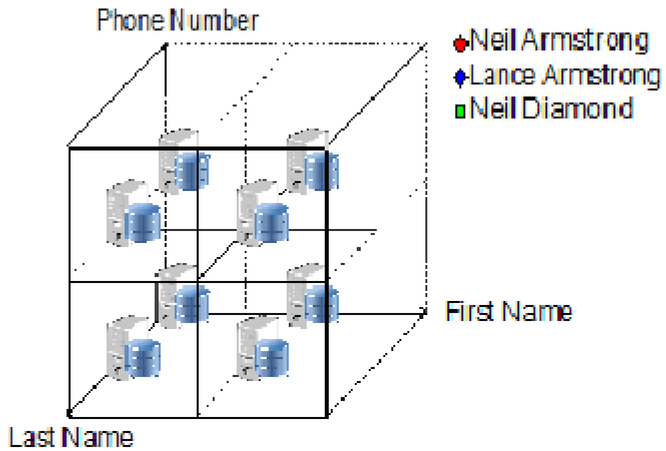
Different objects reside at different coordinates



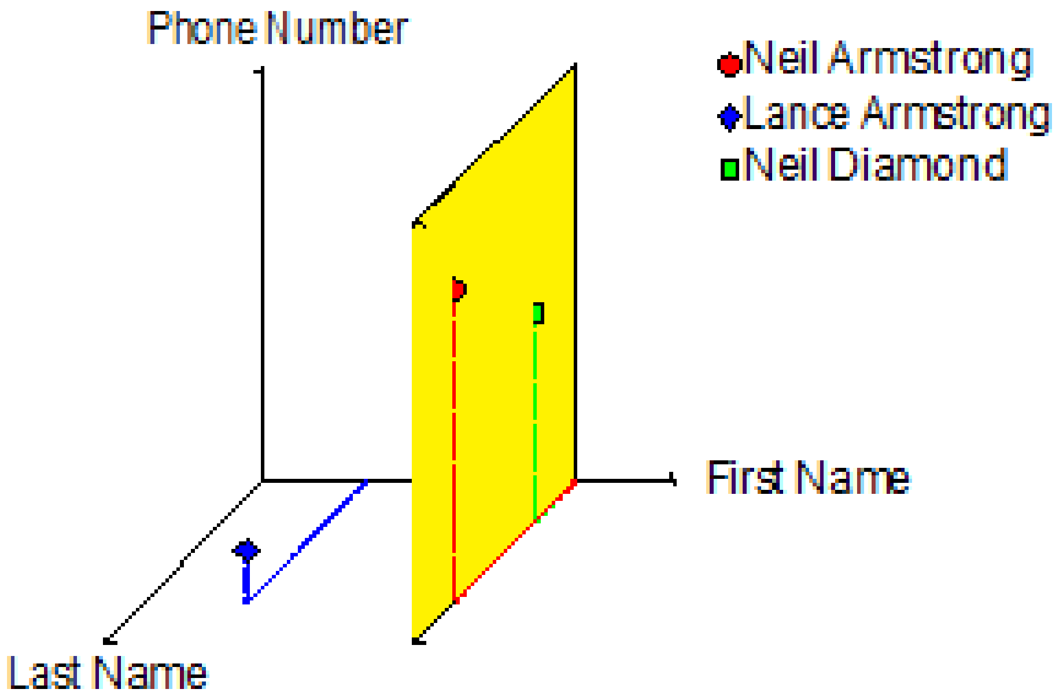
The hyperspace is divided into regions Each object resides in exactly one region



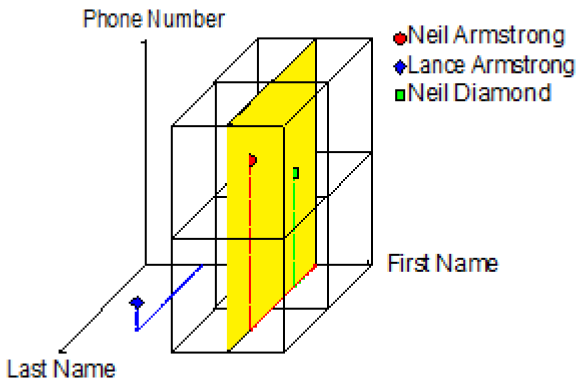
Each server is responsible for a region of the hyperspace



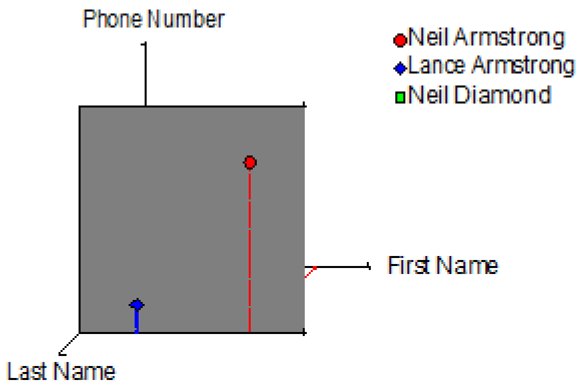
Each search intersects a subset of regions in the hyperspace



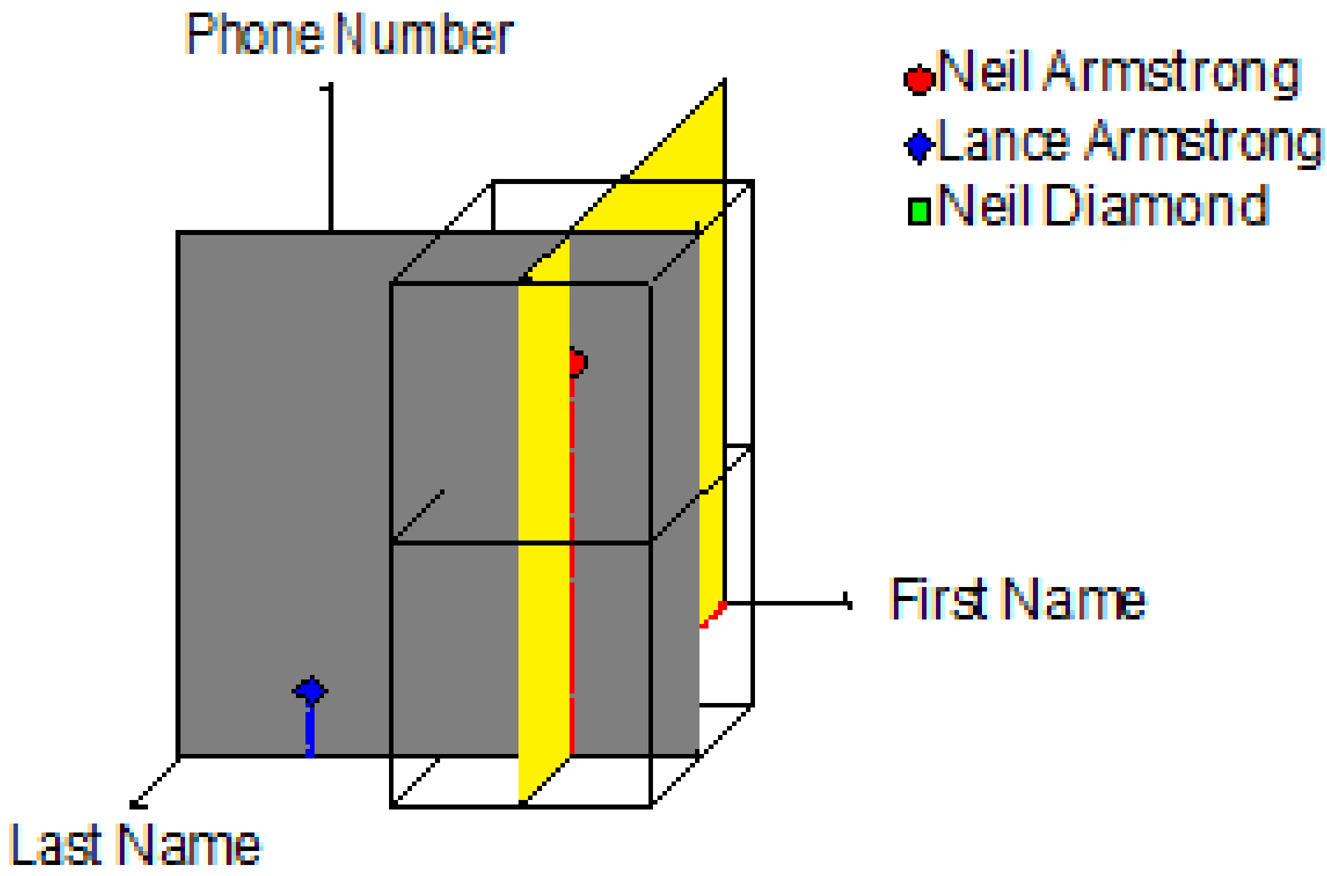
Example: All people name Neil mapped to the yellow plane



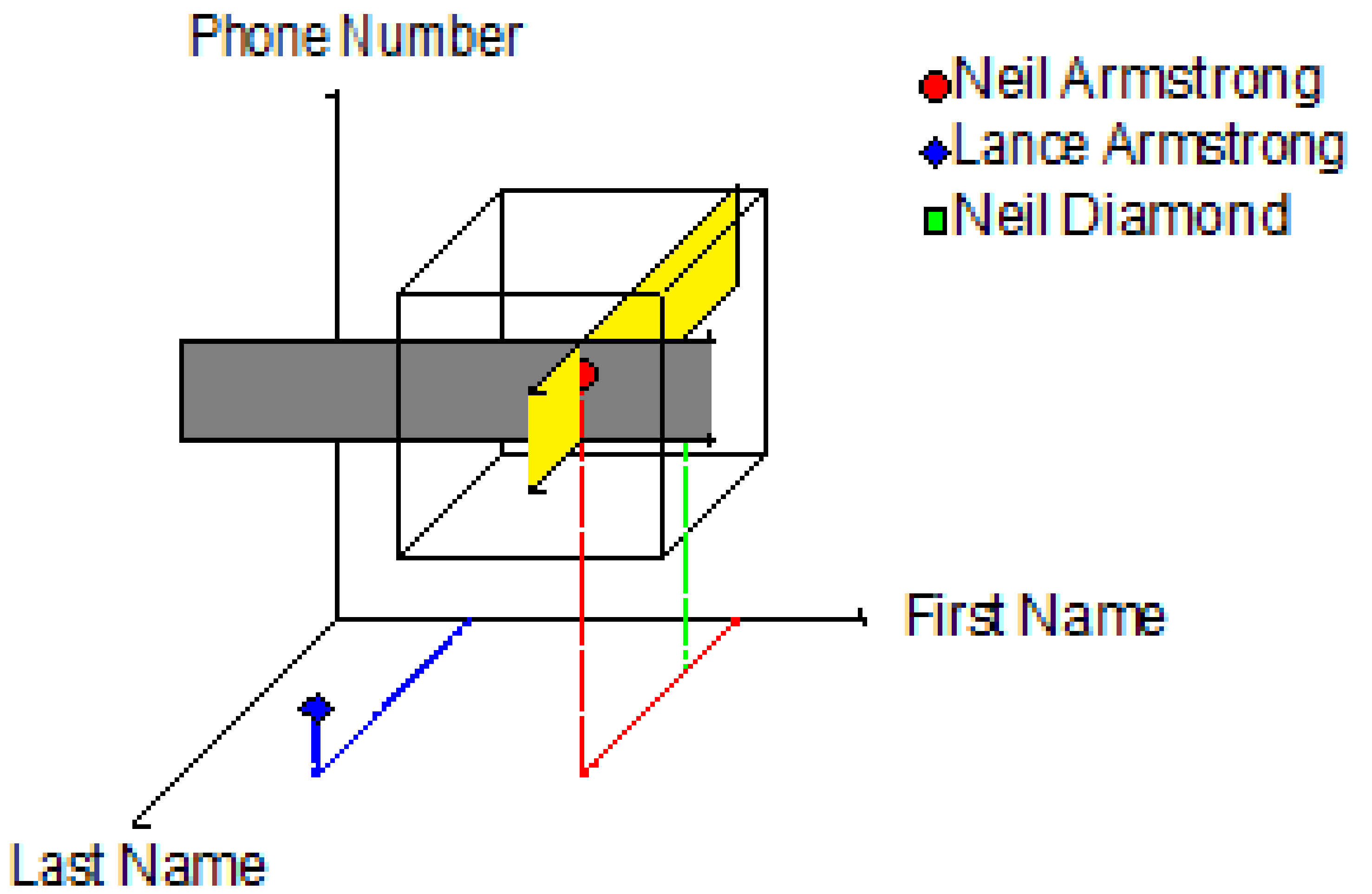
All people named Armstrong map to the grey plane



A more restrictive search for Neil Armstrong contacts fewer servers

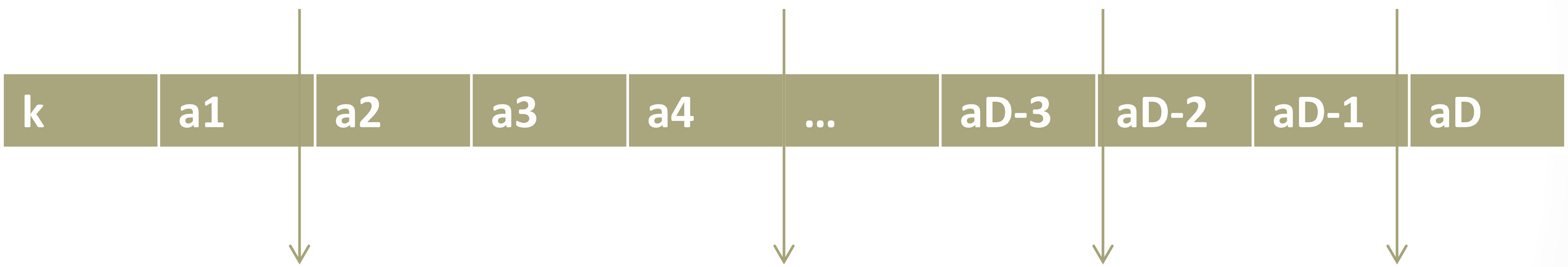


Range searches are natively supported



Space Partitioning: Subspaces

- The hyperspace would grow exponentially in the number of dimensions
- Space partitioning prevents exponential growth in the number of searchable attributes

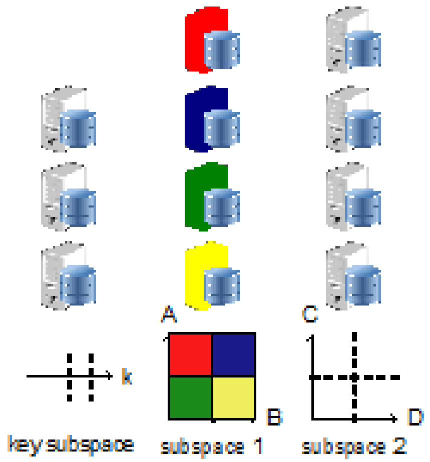


- A search is performed in the most restrictive subspace

Hyperspace Hashing Index

- Searches are efficient
- Hyperspace hashing is a mapping not an index
  - No per-object updates to a shared data structure
  - No overhead for building and maintaining B-trees
  - Functionality gained solely through careful placement

Update a record: Value dependent chaining



Since the values of the fields determine where in the hypercube the record is stored

Must update the record in an ordered method – since it may involve a move of the data

Commits start at the tail and move to the head

- A put involves one node from each subspace
- Servers are replicated in each region to hold replicas of the data providing fault tolerance
- Updates propagate from a 'point leader' that contains the most recent update of a record – proceed to the tail (last dimension)

Hyperdex features

- Consistent: linearizable; GET requests will always return the latest PUT.
- High Availability: the system will stay up in the presence of $\leq f$ failures.
- Partition-Tolerant: for partitions with $\leq f$ nodes, you can be certain your system is still operational.
- Horizontally Scalable: you can grow your system by adding additional servers.
- Performance: high throughput and low variance.
- Searchable: it provides an expressive API for searching your data.

Summary: HyperDex

- Hyperspace hashing
- Value-dependent chaining
- 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 data structures and search