

# Couchbase Developer Tutorial with Java

Raghavan N. Srinivas  
Developer Advocate



**COUCHBASE**

# Objective

- A lecture and Hands-on based program
  - To get started on programming with Couchbase using Java
  - To learn about the basic operations, asynchronous and advanced operations
  - To learn about secondary indexing with views and queries (a Couchbase Server 2.0 feature)
  - To enhance applications with views and queries

# Speaker Introduction

- Architect and Evangelist working with developers
- Speaker at JavaOne, RSA conferences, Sun Tech Days, JUGs and other developer conferences
- Taught undergrad and grad courses
- Technology Evangelist at Sun Microsystems for 10+ years
- Still trying to understand and work more effectively on Java and distributed systems
- Couchbase Developer Advocate working with Java and Ruby developers
- Philosophy: *“Better to have an unanswered question than a unquestioned answer”*

# AGENDA



# Software Requirements

- You will need to install the following software on your laptops:
  - Web Browser
  - Java 6
  - SSH Client (if planning to use Amazon EC2 or other cloud provider)
  - Eclipse (Java EE version with JSP support) or your favorite IDE
  - Couchbase Java SDK (  
<http://www.couchbase.com/develop/java/next>)
  - PDF Viewer

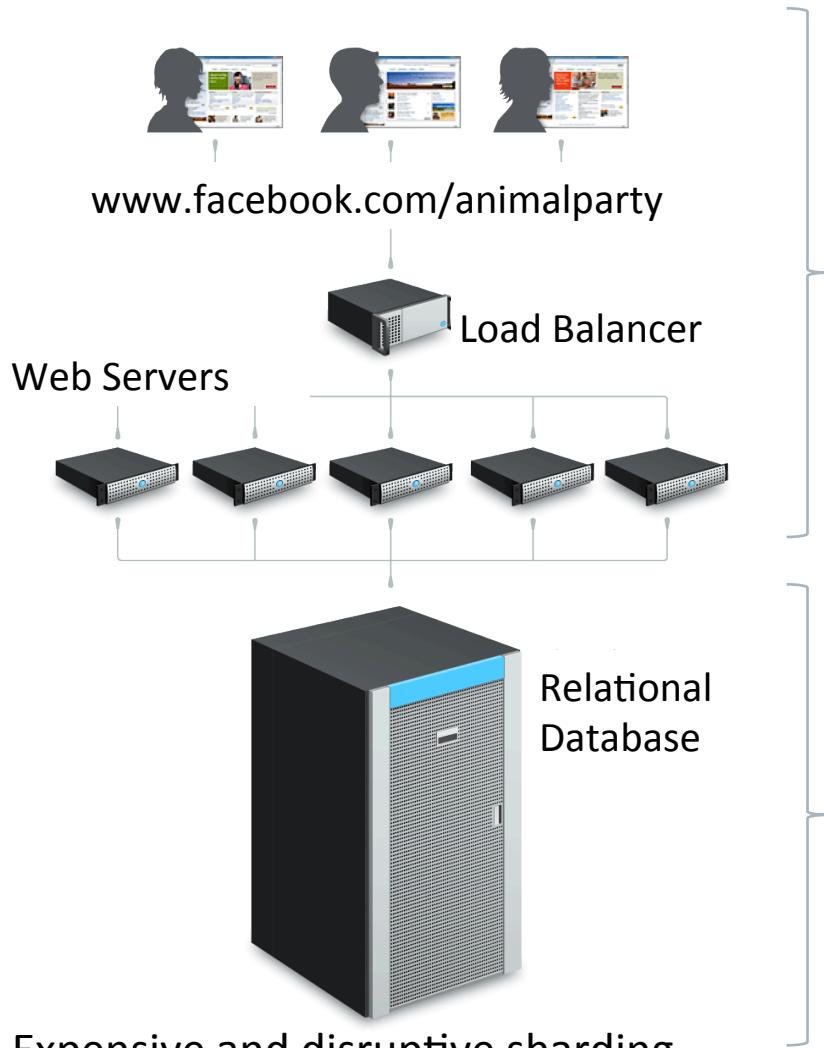
# Overview of Training Material/Exercises

- After installing the prerequisite software
  - Unzip [CouchbaseJavaTraining\\_v5.zip](#) (appropriate version will be provided) into HOL\_ROOT directory
  - Couchbase client JARs and dependencies are provided in the [JARs subdirectory](#) (for convenience)
  - Outline is in [CouchbaseJavaOutline.pdf](#)
  - Presentation and Exercise statements are in [CouchbaseJavaPresentation.pdf](#) or [CouchbaseJavaPresentationAbbreviated.pdf](#)
  - Lab manual (with sketchy instructions) are in [CouchbaseJavaManual.pdf](#)
  - Solutions to exercises (sometimes partial) are in the [Exercises sub directory](#)
  - Exercises are [sequential in nature](#) (i.e. exercises depend on successful execution of earlier exercises). Some are marked as optional

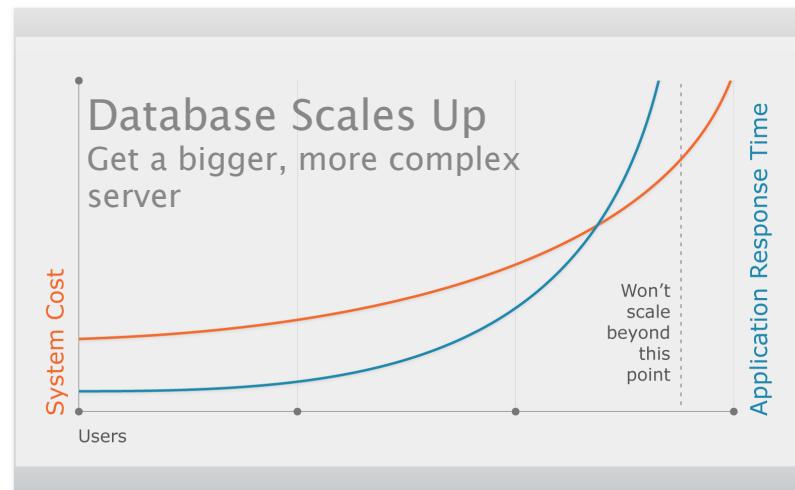
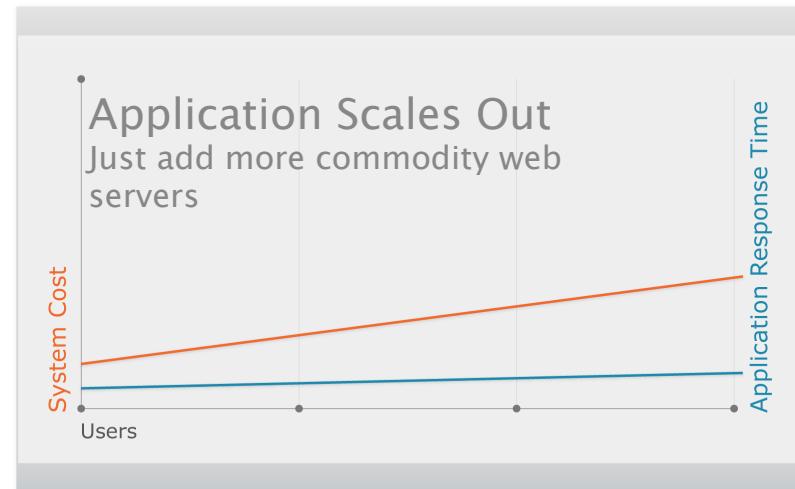
# HIGH LEVEL ARCHITECTURE



# Web Application Architecture and Performance

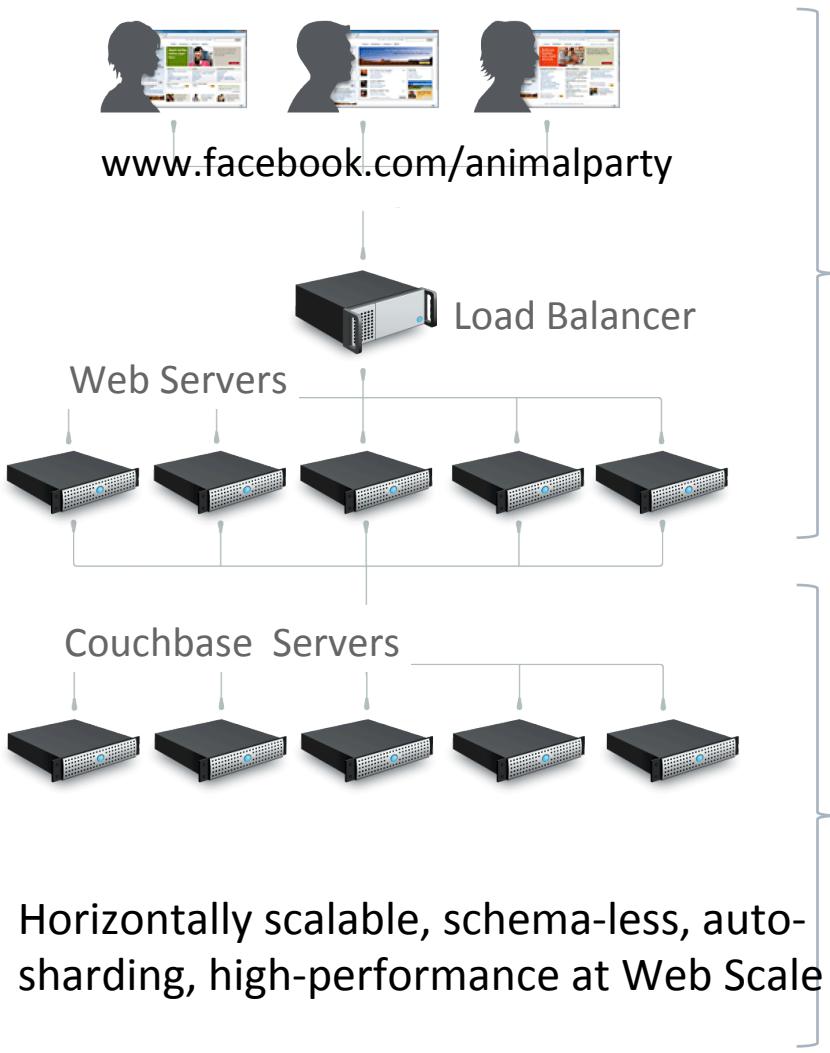


- Expensive and disruptive sharding
- Doesn't perform at Web Scale

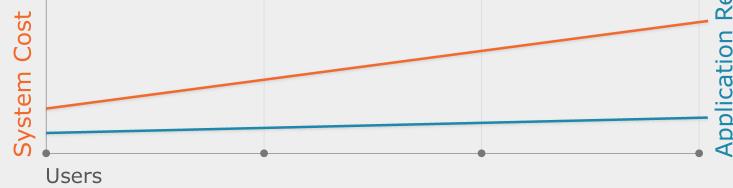


# Couchbase data layer scales like application logic tier

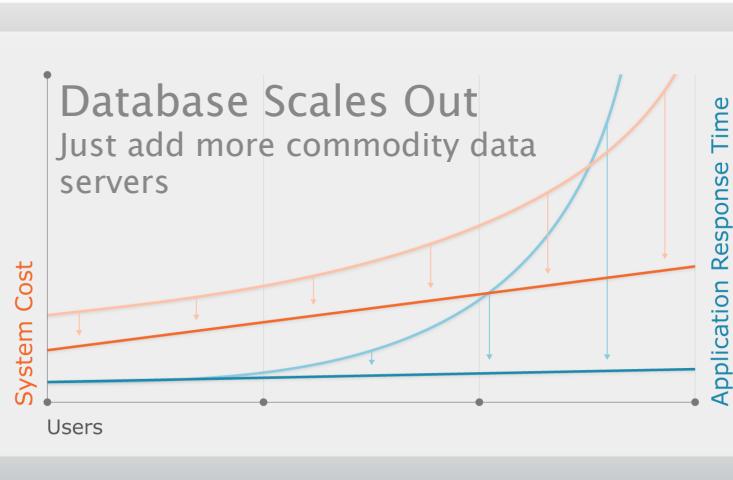
Data layer now scales with linear cost and constant performance.



Application Scales Out  
Just add more commodity web servers



Database Scales Out  
Just add more commodity data servers



Scaling out flattens the cost and performance curves.

# WHAT IS COUCHBASE



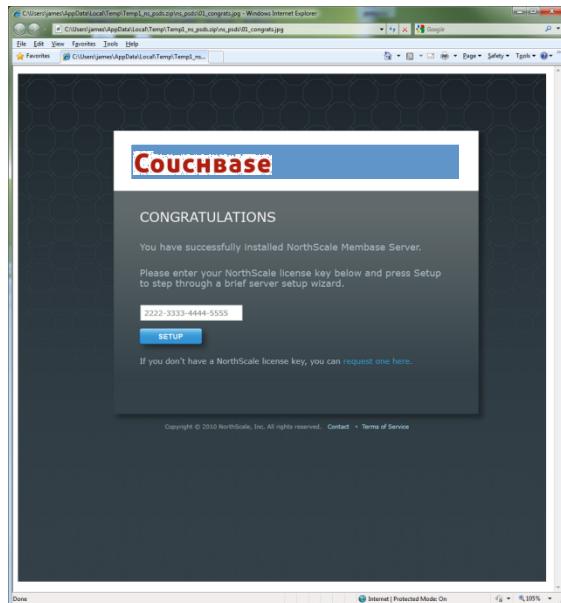
# What is Couchbase?

- Multitenant Document (JSON-based) Store
- Distributed
- Fault Tolerant
  - Persistent
  - Replicated
- Schema-Free

# Qualities

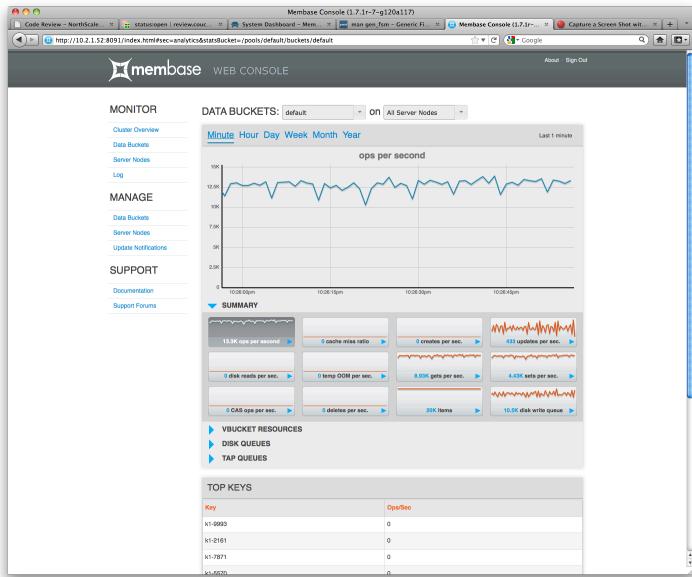
- Simple
  - memcached API
- Uniformly FAST
  - Sub-millisecond response times
  - Predictable scalability
- Elastic
  - Easy scalable

# Couchbase Server is Simple, Fast, Elastic



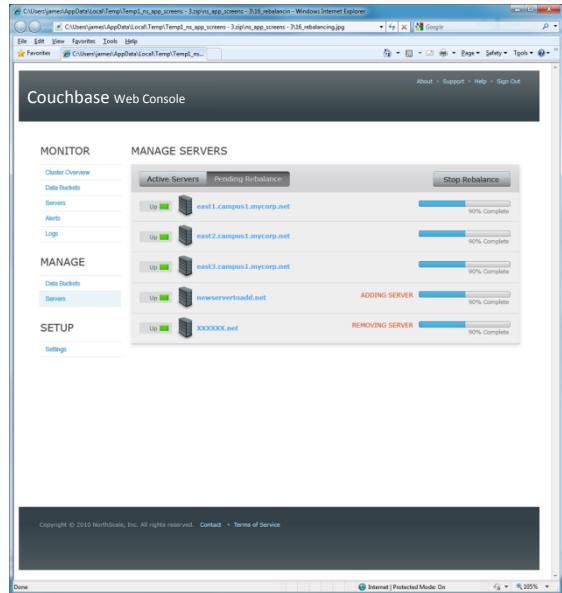
- Five minutes or less to a working cluster
  - Downloads for Windows, Linux and OSX
  - Start with a single node
  - One button press joins nodes to a cluster
- Easy to develop against
  - Just SET and GET – no schema required
  - Drop it in. 10,000+ existing applications already “speak Couchbase” (via memcached)
  - Practically every language and application framework is supported, out of the box
- Easy to manage
  - One-click failover and cluster rebalancing
  - Graphical and programmatic interfaces
  - Configurable alerting

# Couchbase Server is Simple, Fast, Elastic



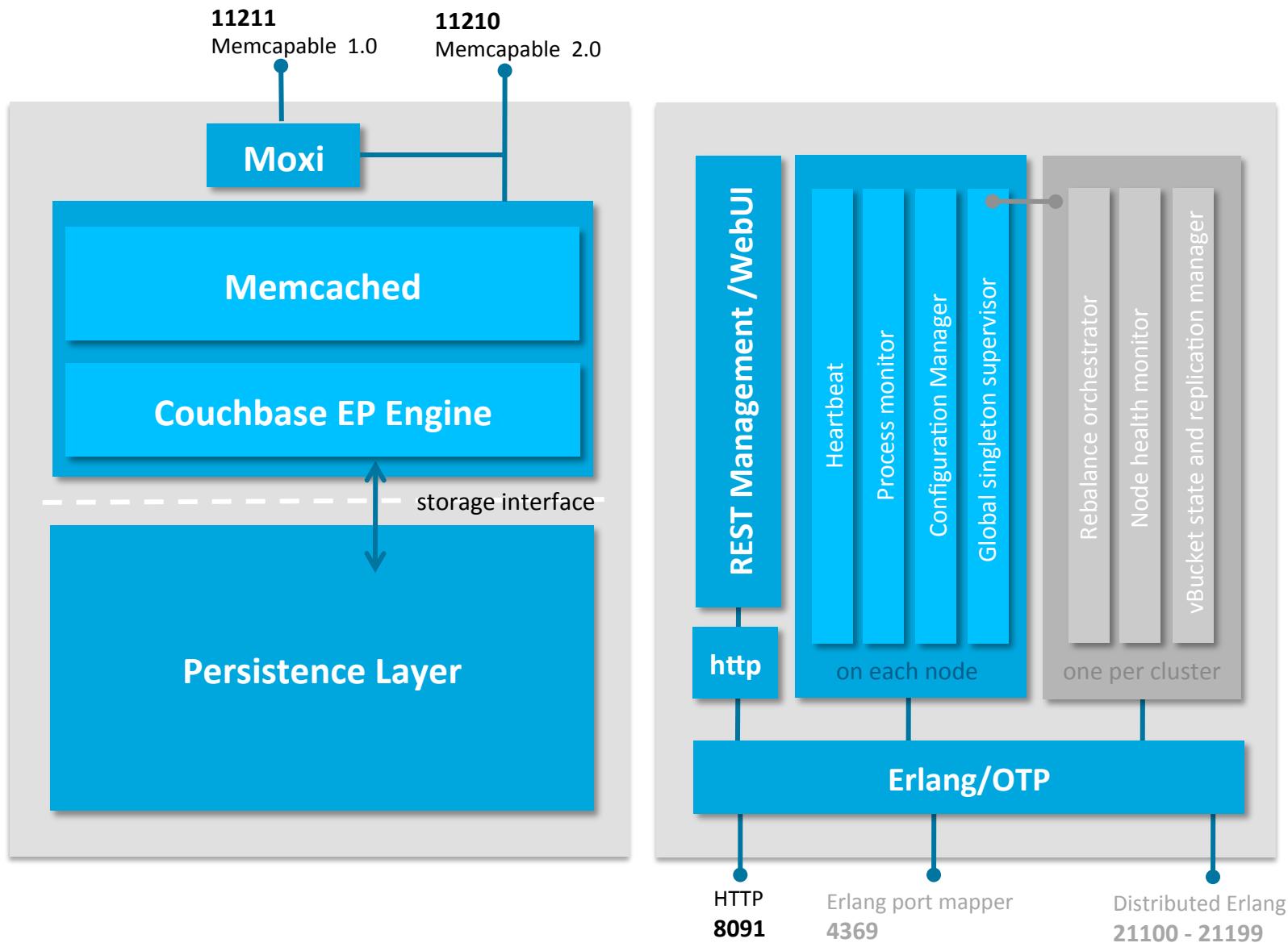
- Predictable
  - “Never keep an application waiting”
  - Quasi-deterministic latency and throughput
- Low latency
  - Built-in Memcached technology
  - Auto-migration of hot data to lowest latency storage technology (RAM, SSD, Disk)
  - Selectable write behavior – asynchronous, synchronous (on replication, persistence)
- High throughput
  - Multi-threaded
  - Low lock contention
  - Asynchronous wherever possible
  - Automatic write de-duplication

# Couchbase Server is Simple, Fast, Elastic

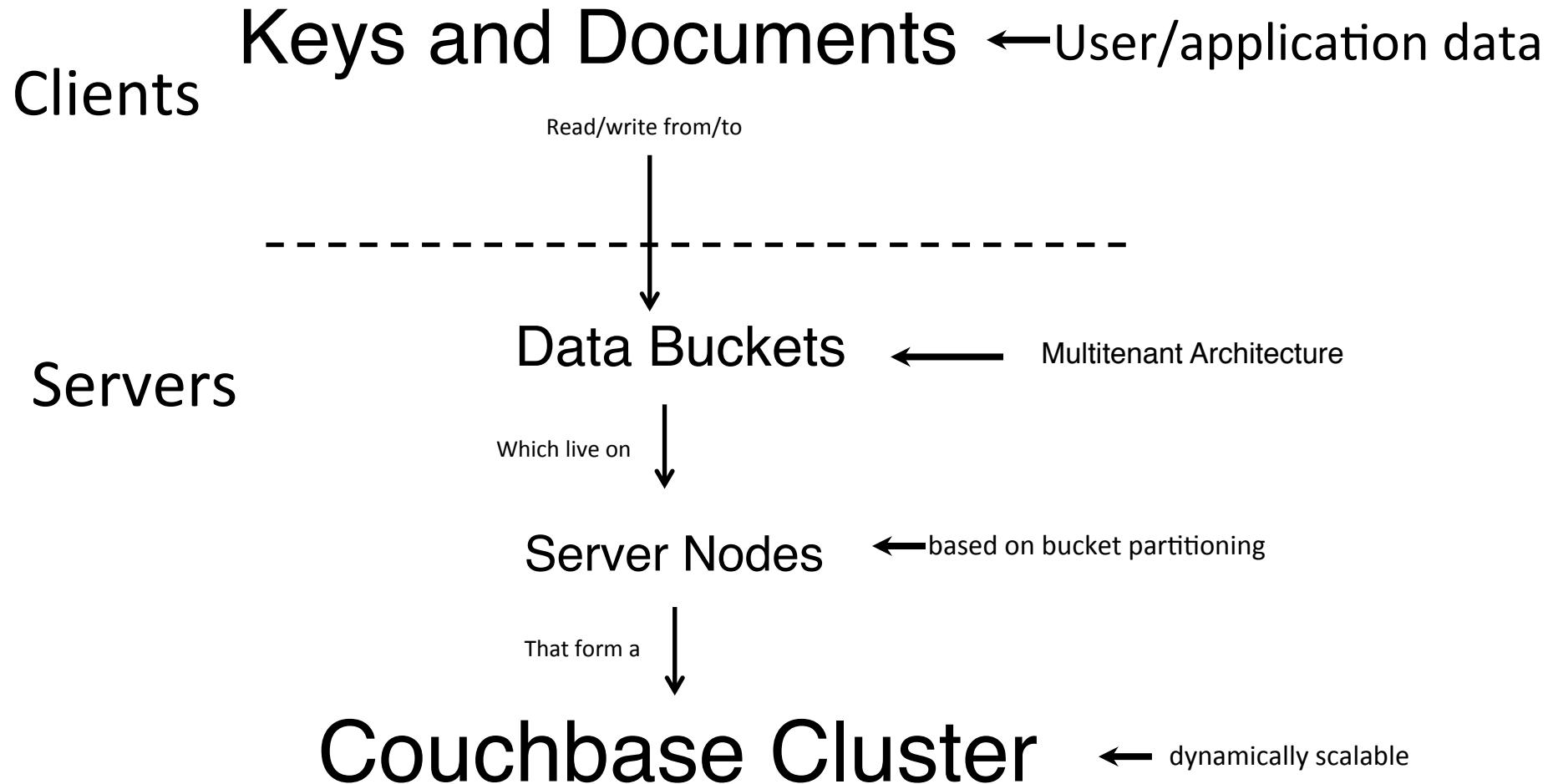


- Zero-downtime elasticity
  - Spread I/O and data across commodity servers (or VMs)
  - Consistent performance with linear cost
  - Dynamic rebalancing of a live cluster
- All nodes are created equal
  - No special case nodes
  - Clone to grow
- Extensible
  - Change feeds
  - Real-time map-reduce
  - RESTful interface for management

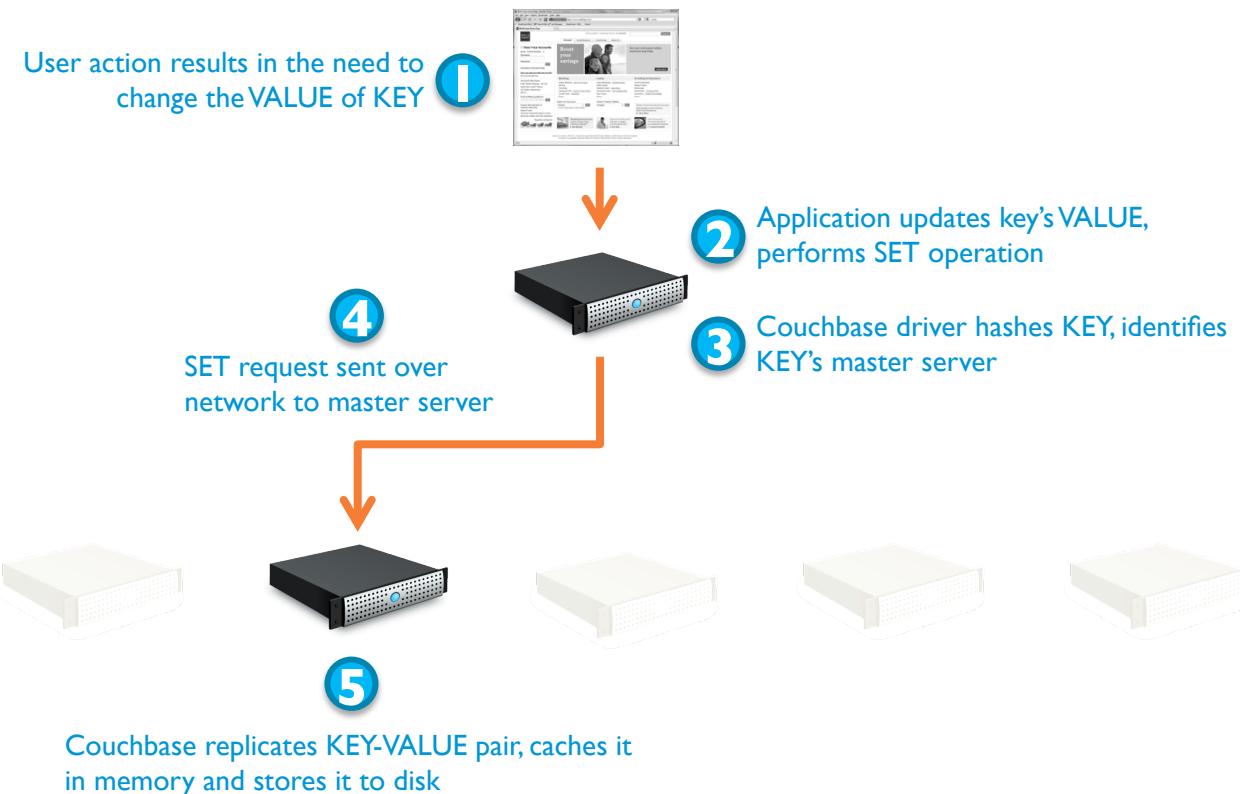
# Component Architecture



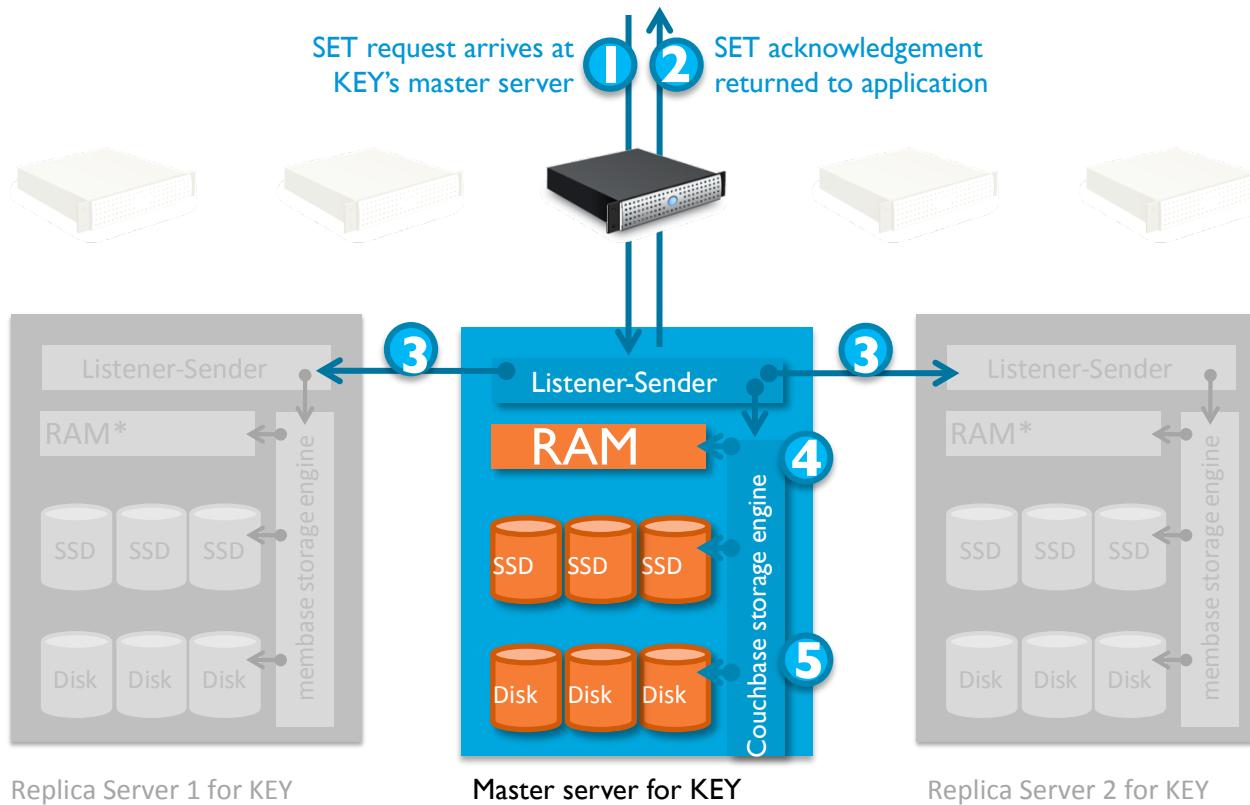
# Key Concepts



# Architecture

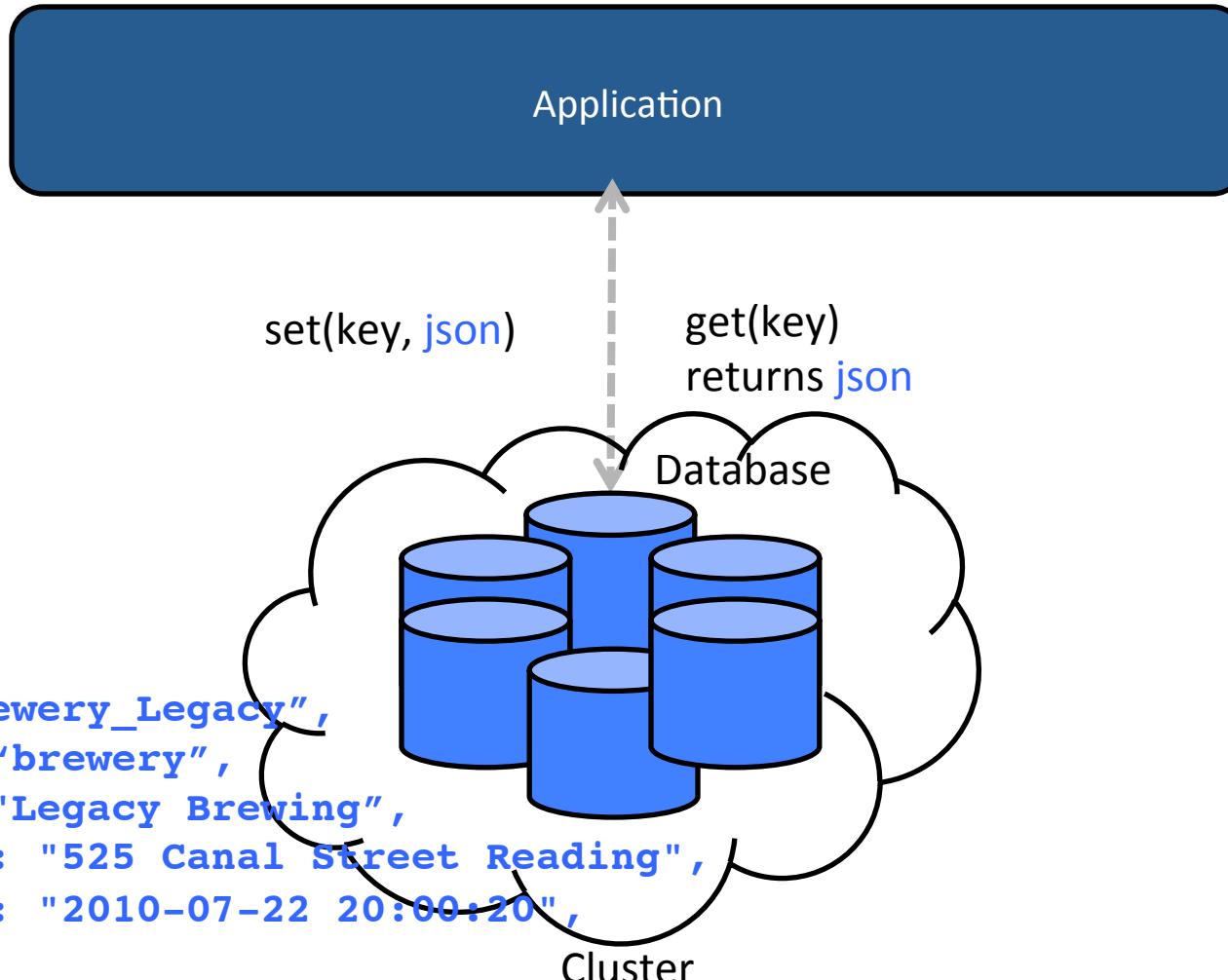


# Operations workflow



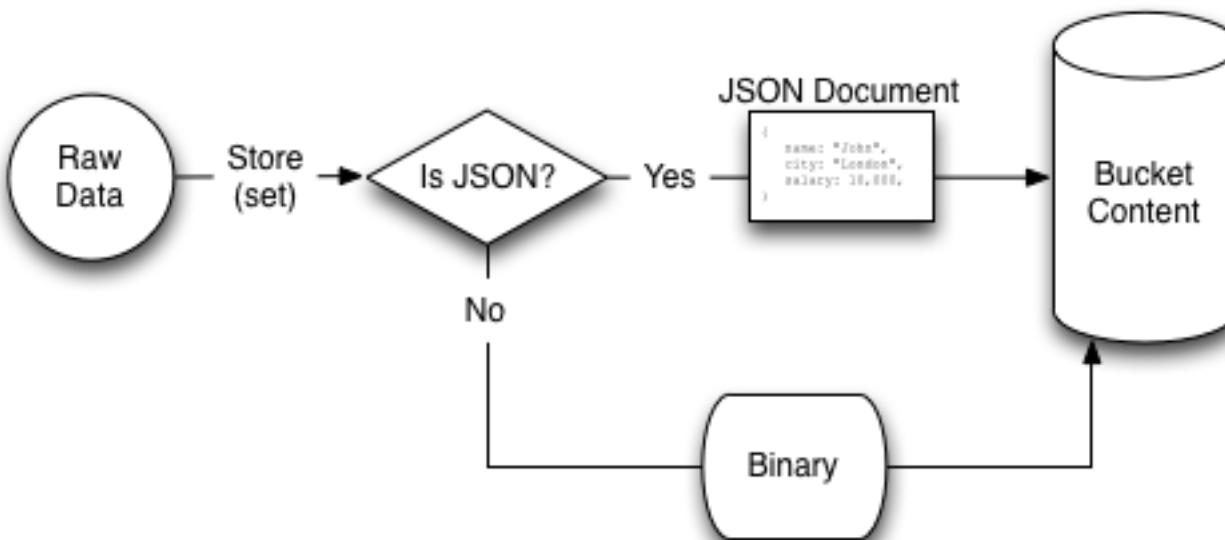
Durability/HA comes first from off-node replication, fallback to disk persistence

# A Distributed Hash Table Document Store



# A JSON-based document store

- JSON is processed specially on the server



# GETTING STARTED



# Installation

- RedHat – RPM
- Ubuntu – dpkg
- Windows
- Mac OS
- GUI Installer
- Unattended installation
- Quickstart once installed
- <http://<hostname>:8091>

# Management Tools

## Web Console

- `http://<IP>:8091`
- JQUERY-based
- Simple/easy management/monitoring
- “pretty graphs”

## CLI

- Python script: `/opt/couchbase/bin/couchbase-cli`
- Useful for scripting, automated management

## REST API

- RESTful API (JSON)
- “Authoritative” API
- Programmatic access for management/monitoring
- Management channel for “smart” clients
- No “data” manipulation
- Documented at: <http://couchbase.org/wiki/display/membase/Membase+Management+REST+API>

Be careful, no “validation/verification”  
on possibly destructive operations

# Exercise 1a – Node Up and Running

- Objective
  - to setup the Couchbase server on localhost
- Steps
  - ssh (if required) into machine
  - Install Couchbase (example):
    - rpm -i couchbase-server-enterprise\_<version>.rpm
  - Launch the Web Console at
    - <http://<host>:8091>
  - Run the Setup Wizard and accept the defaults for each step (except the per node RAM quota for the **default** bucket in step 2, which should be lower since you will create another bucket in the next step)

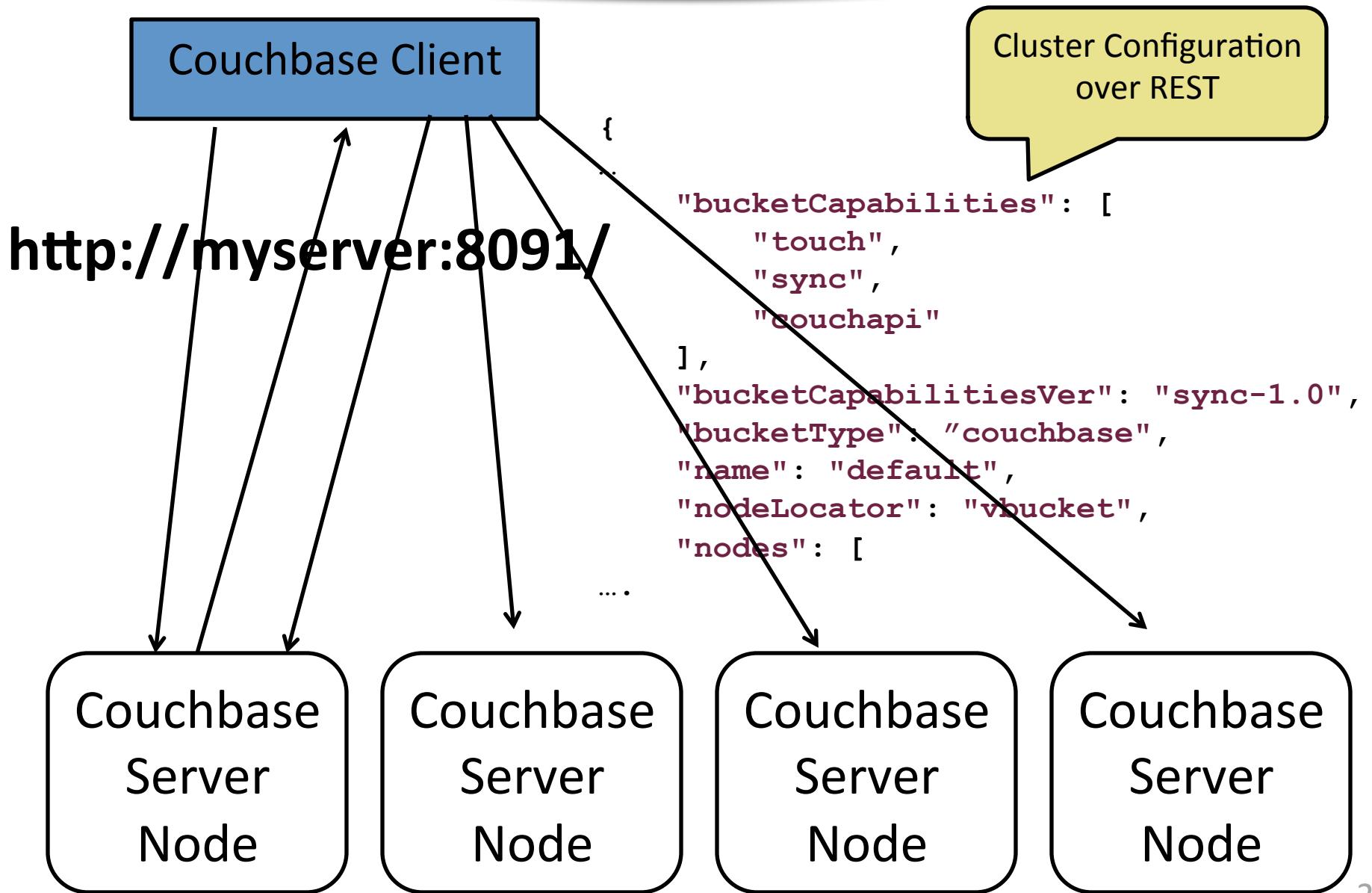
# COUCHBASE CLIENT INTERFACE



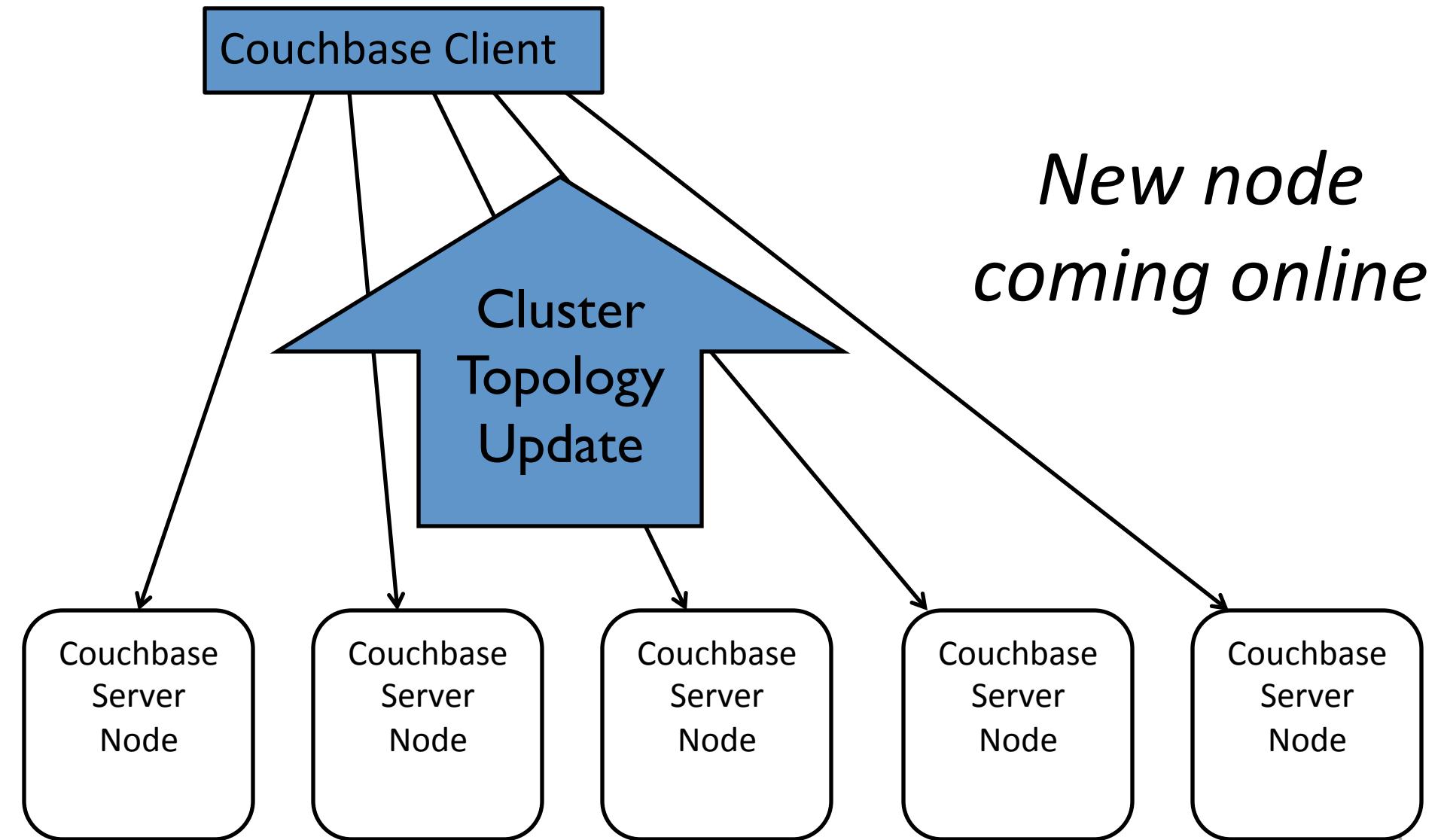
# Smart Clients

- Communicate directly with cluster
- Understand cluster topology
- Automatically distribute requests to cluster nodes
- Automatically direct requests on topology changes
- Automatically direct requests during failover

# Client Setup: Getting Cluster Configuration



# Client at Runtime: Adding a node



# Opening a Connection

- Connect to the Cluster URI of any cluster node

```
List<URI> uris = new LinkedList<URI>();  
  
uris.add(URI.create("http://127.0.0.1:8091/pools"));  
try {  
    client = new CouchbaseClient(uris, "default", "");  
} catch (Exception e) {  
    System.err.println("Error connecting to Couchbase: " +  
e.getMessage());  
    System.exit(0);  
}
```

# Authentication

- Buckets authenticated using SASL
- Create a ConnectionFactory, then client:

```
List<URI> baseURIs = new ArrayList<URI>();  
baseURIs.add(base);  
CouchbaseConnectionFactory cf = new  
    CouchbaseConnectionFactory(baseURIs,  
        "userbucket", "password");  
  
client = new CouchbaseClient(cf);
```

# Java Client Library Dependencies

- common-codecs
  - Base 64 encoding. Encoders and decoders
- jettison
  - JSON Processing
- netty
  - HTTP comet streaming
- httpcore-nio
  - High performance NIO sockets (for views)
- httpcore
  - Needed by httpcore-nio

# Exercise 2 – Set up Eclipse Project and JARs

- Objective
  - to setup the Couchbase SDK and its dependencies
- Steps
  - Import HelloCouchbase into Eclipse as a Maven project
  - Modify the pom.xml (if required) and run the project
- If you do not want to use Maven and Eclipse OR want to use your own IDE, copy the JAR files and include them in your classpath as described in

[http://www.couchbase.com/docs/couchbase-sdk-java-1.1/  
hello-world.html](http://www.couchbase.com/docs/couchbase-sdk-java-1.1/hello-world.html)

# PROTOCOL OVERVIEW



# Data is stored as key/document pairs

- Key is the identifier
  - A string without spaces
  - May use separators/identifiers; for example person\_93847
  - Must be unique within a bucket
- Document is pure bytestring (JSON adds more meaning)
  - No implied value or understanding on server-side (generally)
  - Integers are valid for certain operations (increment, decrement)
  - Can store strings, images, or serialized objects
  - Server will try to discern JSON from non-JSON

# Metadata

- Server stores metadata with each key/value pair
  - Expiry (TTL)
  - CAS (checksum) value
  - Flags
- Expiry is used by server to delete values after the specified time
- Flags can be used to identify the data type of stored information
  - In Java, this information is used to serialize the data type of the object stored

# Time To Live (TTL)

- TTL
  - Property to set expiration on the document
  - the actual value sent may either be
  - Unix time (number of seconds since January 1, 1970, as a 32-bit value)
  - OR number of seconds starting from current time. number of seconds may not exceed  $60*60*24*30$  (number of seconds in 30 days)
  - if the number sent by a client is larger than
  - that, the server will consider it to be real Unix time value rather than an offset from current time.

# Store Operations

Operation	Description
add()	Adds new value if key does not exist or returns error.
replace()	Replaces existing value if specified key already exists.
set()	Sets new value for specified key.

All sets allow for ‘expiry’ time specified in seconds:

Expiry Value	Description
Values < 30*24*60*60	Time in seconds to expiry
Values > 30*24*60*60	Absolute time from epoch for expiry

# Retrieve Operations

Operation	Description
get()	Get a value.
getAndTouch()	Get a value and update the expiry time.
getBulk()	Get multiple values simultaneously, more efficient.
gets()	Get a value and the CAS value.

# Update Operations

Operation	Description
append()	Appends data to an existing key.
prepend()	Prepends data to an existing key.
incr()	Increments an integer value by specified amount, default 1.
decr()	Decrements an integer value by specified amount, default 1.
replace()	Replaces an existing key/value pair.
touch()	Updates the expiry time for a given item.

# Exercise 4: Connect, Store, Retrieve

- Objective
  - to learn about some of the operations supported on the Couchbase client
- Steps
  - Set up a Couchbase connection
  - Modify the program provided to use the following operations
    - set(), get(), add(), replace(), touch(), append(), prepend(), etc.
  - Try different parameters available

# ASYNCHRONOUS OPERATIONS



# Couchbase Java Client Asynchronous Functions

- Synchronous commands
  - Force wait on application until return from server
- Asynchronous commands
  - Allow background operation until response available
  - Operations return **Future** object
  - Useful when data on server not in-memory, or during sets/updates

# Asynchronous functions

- A Typical Asynchronous call
  - An asynchronous call that returns a Future object (Non blocking)
  - Process something (in the meantime) that does not depend on the result and the result is still being computed on the server
  - Either
    - Call the **get()** method on the Future object (blocking call) OR
    - Call the **cancel()** method to cancel the operation

# It's Asynchronous

- Set Operation is asynchronous

```
// Do an asynchronous set
OperationFuture<Boolean> setOp =
    client.set(KEY, EXP_TIME, VALUE);

// Do something

// Check to see if our set succeeded
// We block when we call the get()
if (setOp.get().booleanValue()) {
    System.out.println("Set Succeeded");
} else {
    System.err.println("Set failed: " +
        setOp.getStatus().getMessage());
}
```

# It's Asynchronous

- Even a get can be async

```
GetFuture getOp = client.asyncGet(KEY);

// do some work

if ((getObject = getOp.get()) != null) {
    System.out.println("Asynchronous Get Succeeded: " + getObject);
} else {
    System.err.println("Asynchronous Get failed: " +
        getOp.getStatus().getMessage());
}
```

# Exercise 5: Asynchronous Operations

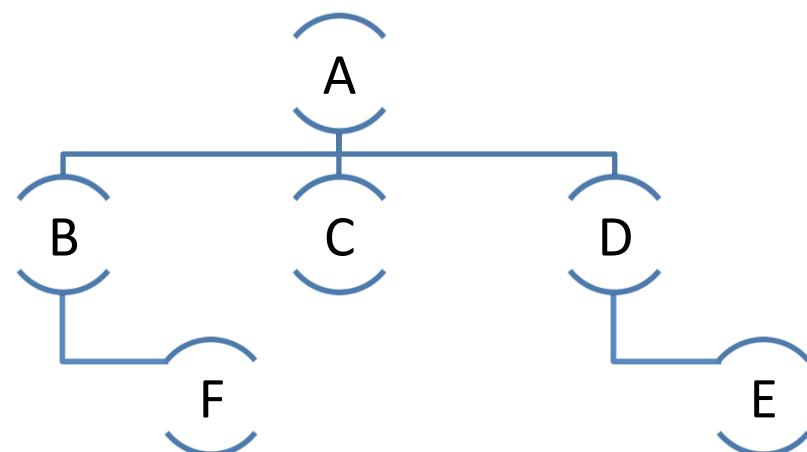
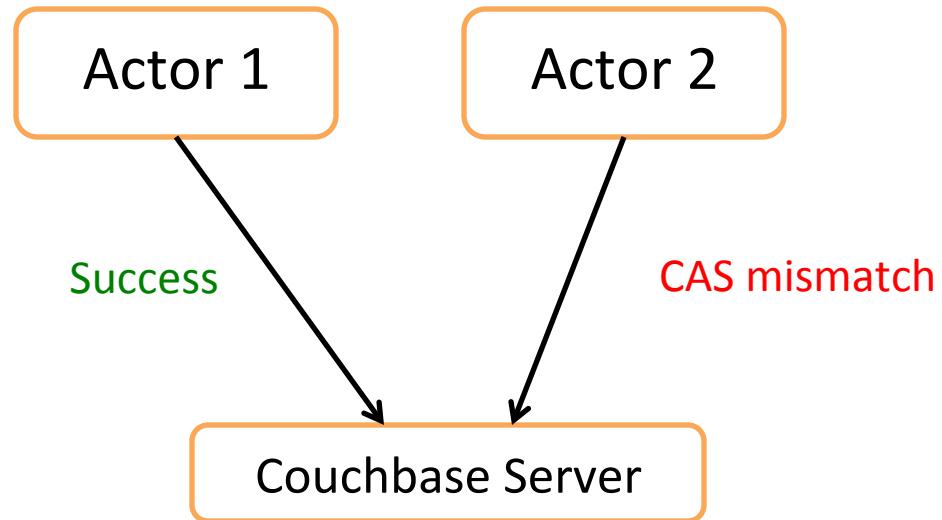
- Objective
  - to learn about some of the asynchronous operations supported on the Couchbase client
- Steps
  - Set up a Couchbase connection
  - Modify the program provided to use the following operations
    - `asyncGet()`, `set()`, `delete()`, etc.
    - Use the return Future value
    - Try different parameters available

# CONCURRENCY



# Distributed System Design: Concurrency Controls

- Compare and Swap Operations
  - Often referred to as “CAS”
  - Optimistic concurrency control
  - Available with many mutation operations, depending on client.
- Get with Lock
  - Often referred to as “GETL”
  - Pessimistic concurrency control
  - Locks have a short TTL
  - Locks released with CAS operations
  - Useful when working with object graphs



# Check and Set/Compare and Swap (CAS)

- Uses checksum to validate a change to a value:
  - Client gets key and checksum (cas\_token)
  - Client updates using key and checksum
  - If checksum doesn't match, update fails
- Client can only update if the key + checksum match
- Used when multiple clients access the same data
- First client updates with checksum
- Subsequent client updates fail without the right checksum
- Use CASMutation and CASMutator for higher level of abstraction

# CAS Operation

- CAS Example

```
CASValue<Object> casv = client.gets(KEY);
Thread.sleep(random.nextInt(1000)); // a random workload

// Wake up and do a set based on the previous CAS value
Future<CASResponse> setOp =
    client.asyncCAS(KEY, casv.getCas(), random.nextLong());

// Wait for the CAS Response
try {
    if (setOp.get().equals(CASResponse.OK)) {
        System.out.println("Set Succeeded");
    } else {
        System.err.println("Set failed: ");
    }
}
...
```

# Exercise 7a: Concurrency Operations

- **Objective**
  - to learn about Concurrency operations supported on the Couchbase client
- **Steps**
  - Set up a Couchbase connection
  - Use the following operations
    - `gets()` and `asyncCAS()` to demonstrate contention between two separate clients

# INTRODUCING DOCUMENTS



# A JSON Document

{

```
"id": "beer_Hoptimus_Prime",  
"type": "beer",  
"abv": 10.0,  
"brewery": "The Bruery Brewing Co.",  
"category": "A float American Ale",  
"name": "Hoptimus Prime",  
"style": "Imperial or Double India Pale Ale",  
}
```

The primary key

The type information

A float



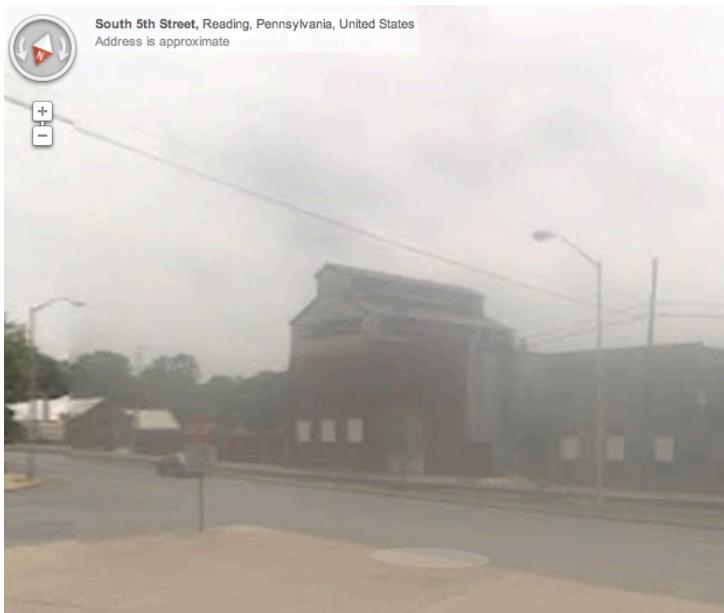
beeraday.net

# Other Documents and Document Relationships

{

```
"id": "beer_Hoptimus_Prime",
"type" : "beer",
"abv": 10.0,
"brewery": "brewery_Legacy_Brewing_Co",
"category": "North American Ale",
"name": "Hoptimus Prime",
"style": "Double India Pale Ale"
```

}



{

```
"id": "brewery_Legacy_Brewing_Co",
"type" : "brewery",
"name" : "Legacy Brewing Co.",
"address": "525 Canal Street
Reading, Pennsylvania, 19601 United
States",
"updated": "2010-07-14T13:45:28Z",
"latitude": -75.92833333333333,
"longitude": 40.325725}
```

Afterthought

# Simplicity of Document Oriented Datastore

- Schema is optional (schema evolves with the app.)
  - Technically, each document has an implicit schema
  - Extend the schema at any time!
    - Need a new field? Add it. Define a default for similar objects which may not have this field yet.
    - These changes to the schema (as the application evolves) can be tracked by a version number or other fields (as needed)
- Data is self-contained
  - Documents more naturally support the world around you, the data structures around you
- Model data for your App/Code instead for the Database

# Adding a Document: Observations and Considerations

- Observations
  - Conversion to document was very simple, many JSON options
  - Flexible schema: Did not need to add the latitude and longitude to every record
  - Flexible schema: Can add the brewery detail later
- Considerations
  - Use a “type” field for high level filtering on object types
  - Why use a particular key/\_id : **“beer\_My\_Brew”**
  - Should I have a TTL?

# Common Questions when Adopting Couchbase

*Q: What if I need to fetch referenced documents?*

A: Simply get them one after another or use another View.

*Q: How can I update just a small portion of a document?*

A: The best approach is to keep the document model live in your application, then use CAS operations to store modified documents. The Ruby sample application has a good example.

*Q: I currently use serialized objects with memcached or Membase, can I do this still with Couchbase Server?*

A: Absolutely! Everything previously supported and used is still there. JSON offers advantages with heterogenous platform support and preparing for Couchbase 2.0 views.

# Exercise 10: Loading Beer and Brewery Data

- Objective
  - Loading the Beer Data into Couchbase so that it can be used in the sample App.
- Steps
  - Set up a Couchbase connection
  - Use Gson libraries to load the Beers (beers.json will be provided) and Breweries (breweries.json) into Couchbase
  - View the Key and Value using the admin console on the server
  - Modify the schema to add a field(s) and view the Key and Value on the server

# THE VIEW AND QUERY API



# Couchbase Server 2.0: Views

- Views can cover a few different use cases
  - Simple secondary indexes (the most common)
  - Aggregation functions
    - Example: count the number of North American Ales
  - Organizing related data

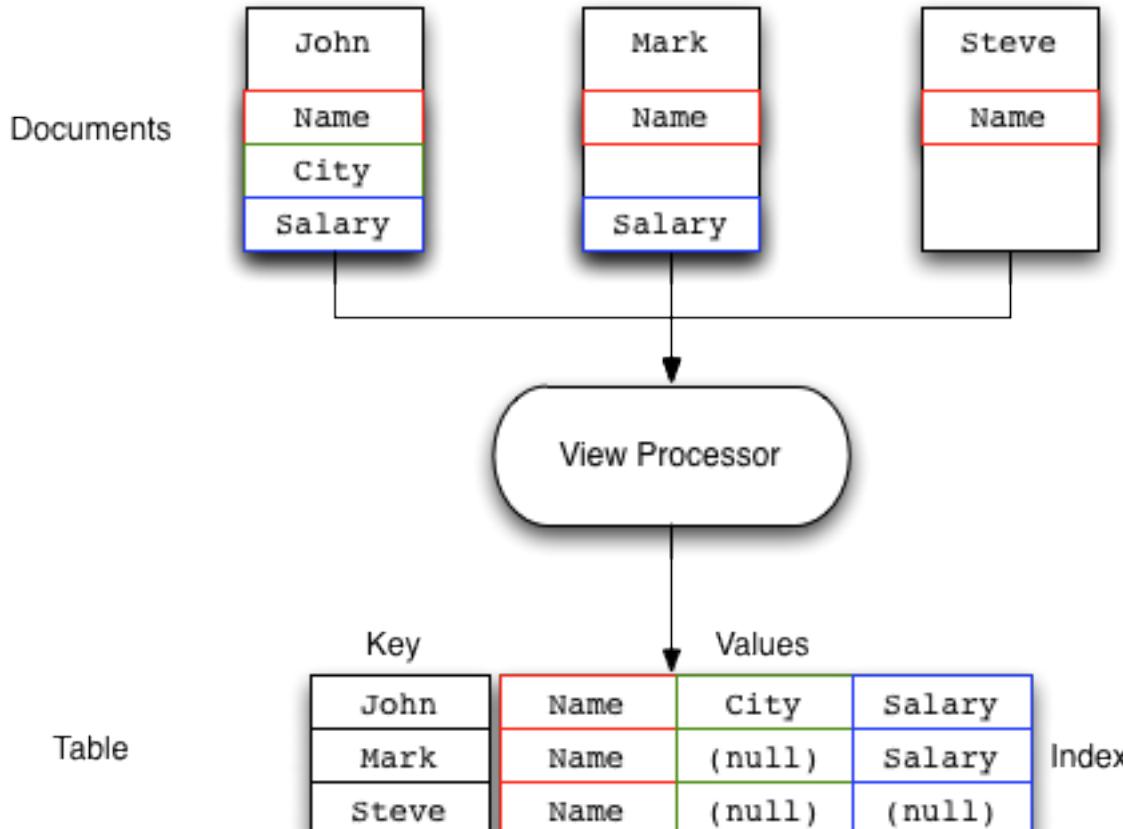
# Secondary Indexing and Views

UserId	Message	Type	date
<User-1>	Hello London	Message	Mon Mar-19 09:02:02
<User-2>	Hello Berlin	Message	Mon Mar-19 10:03:05
<User-3>	Hello Rome	Message	Mon Mar-19 10:05:06
<User-3>	Hello Zurich	Message	Mon Mar-19 10:06:08
<User-3>	Hello Paris	Message	Mon Mar-19 10:06:09
<User-4>	Hello Amsterdam	Message	Mon Mar-19 10:07:03
<User-5>	Hello Madrid	Message	Mon Mar-19 10:11:01

All JSON documents of Type  
Message since Mon Mar-19  
10:00:00

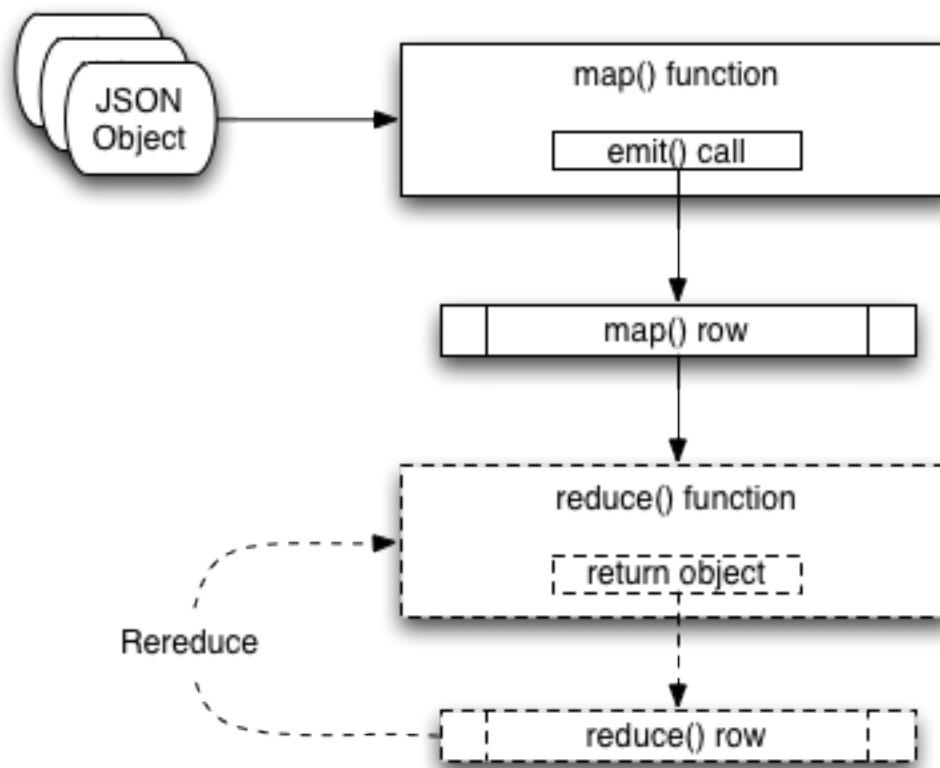
# What are Views?

- Extract fields from JSON documents and produce an index of the selected information



# View Creation using incremental Map/Reduce

- map function creates a mapping between input data and output
- reduce function provides a summary (such as count, sum, etc.)



# Views – adapting to changing schemas

- Original JSON document

```
{ "email" : "rags@acm.org", "name" : "Rags Srinivas" }
```

- View

```
function(doc) {  
    emit([doc.name, doc.email], null);  
}
```

- New JSON document

```
{ "email" : "rags@acm.org", "first" : "Rags", "last" :  
"Srinivas" }
```

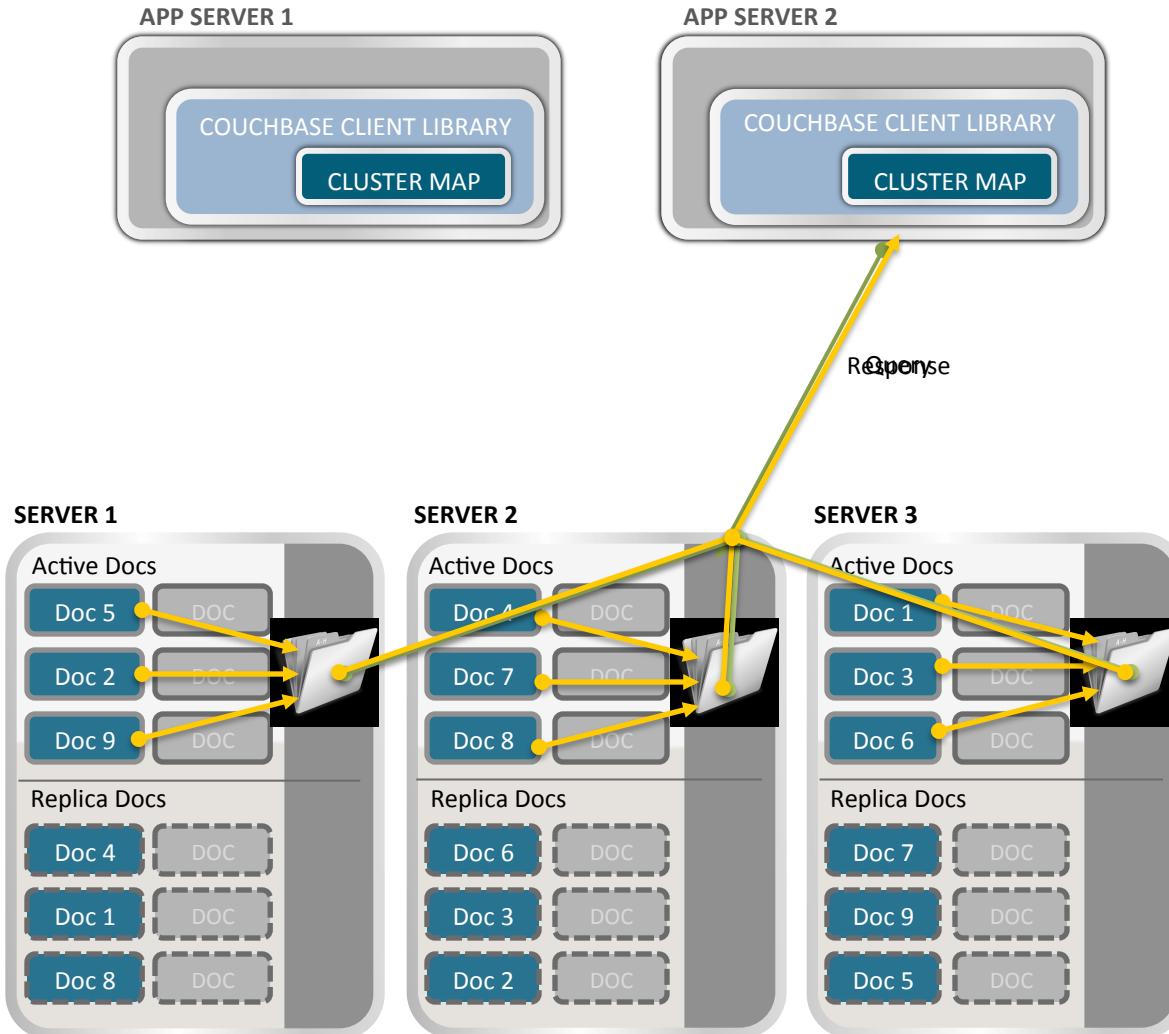
- New View

```
function(doc) {  
    if (doc.name != null) {  
        emit([doc.name, doc.email], null);  
    } else {  
        emit([doc.last + "," + doc.first, doc.email], null);  
    }  
}
```

# Views – Details

- Views
  - All the views within a design document is incrementally updated when the view is accessed (lazily by default)
  - The entire view is recreated if the view definition has changed
  - Views can be conditionally updated by specifying the stale argument to the view query
  - The index information stored on disk consists of the combination of both the key and value information defined within your view.
  - During a view query, the index information for the given view query on *all* the vBuckets within the cluster is collated and returned to the client.

# Indexing and Querying



- Indexing work is distributed amongst nodes
  - Large data set possible
  - Parallelize the effort
- Each node has index for data stored on it
- Queries combine the results from required nodes

# THE QUERY API



# Query APIs

```
// map function
function (doc) {
  if (doc.type == "beer") {
    emit(doc._id, null);
  }
}

// Java code
Query query = new Query();

query.setReduce(false);
query.setIncludeDocs(true);
query.setStale(Stale.FALSE);
```

# THE VIEW API



# View APIs with Java

```
// map function
function (doc) {
  if (doc.type == "beer") {
    emit(doc._id, null);
  }
}

// Java code
View view = client.getView("beers", "beers");

ViewResponse result = client.query(view, query);

Iterator<ViewRow> itr = result.iterator();

while (itr.hasNext()) {
  row = itr.next();
  doc = (String) row.getDocument();
  // do something
}
```

# Querying in Development/Production modes

- Querying modes
  - Development/Production modes
  - Production environment involves processes
- How to set it
  - In Java, with the properties file, command line or `System.setProperty()`

# USING THE APIs



# Querying with Java – Descending Order

```
// map function
function (doc) {
  if (doc.type == "beer") {
    emit(doc.abv, null);
  }
}

// Java code
View view = client.getView("beers", "beers_by_abv");

Query query = new Query();

query.setReduce(false);
query.setIncludeDocs(true);
query.setStale(Stale.FALSE);
Query.setDescending(true);
```

# Implementing a Inner Left Join

- An example
  - “Join” the beers and breweries data based on brewery Id.
- Steps
  - Define a view for one of the data documents (or table)
  - Access the view on the first document
  - Iterate over the rows
  - For each row
    - Access another view using the “foreign key” i.e. brewery ID on the second document
    - Iterate over the rows

# Implementing a Join

```
View view2 = client.getView("breweries", "breweries");

Query query2 = new Query();
query2.setIncludeDocs(true);
query2.setStale(Stale.FALSE);

// Set the brewery Id and get the brewery details
query2.setKey(beer.breweryId);

ViewResponse result2 = client.query(view2, query2);

Iterator<ViewRow> itr2 = result2.iterator();

while (itr2.hasNext()) {
    row2 = itr2.next();
    doc = (String) row2.getDocument();
    // do something
}
```

# Querying with Java – Custom View Key Range

```
View view2 = client.getView("breweries", "breweries");

Query query2 = new Query();
query2.setIncludeDocs(true);
query2.setStale(Stale.FALSE);

// Set the brewery Id and get the brewery details
query2.setRange(beer.breweryId, beer.breweryId);

ViewResponse result2 = client.query(view2, query2);

Iterator<ViewRow> itr2 = result2.iterator();

while (itr2.hasNext()) {
    row2 = itr2.next();
    doc = (String) row2.getDocument();
    // do something
}
```

# Exercise 14a: Listing Beer and Brewery Data

- Objective
  - Listing the Beer and Brewery Data using Views
- Steps
  - Setup Views on the server (in development) for listing Beer and Brewery Data
  - Write a program to
    - Set up a Couchbase connection
    - Use the views to display the data and try different options
    - Shutdown the connection

# Exercise 14b: Implementing a Join

- Objective
  - Implementing a left inner join
- Steps
  - Use the views setup earlier
  - Write a program to
    - Set up a Couchbase connection
    - Use the views to implement an inner left join using the brewery Id
    - Shutdown the connection

---

# RESOURCES AND SUMMARY

---

# Resources, Summary and Call For Action

- Couchbase Server Downloads
  - <http://www.couchbase.com/downloads-all>
  - <http://www.couchbase.com/couchbase-server/overview>
- Views
  - <http://www.couchbase.com/docs/couchbase-manual-2.0/couchbase-views.html>
- Developing with Client libraries
  - <http://www.couchbase.com/develop/java/current>
- Couchbase Java Client Library wiki – tips and tricks
  - <http://www.couchbase.com/wiki/display/couchbase/Couchbase+Java+Client+Library>

**THANKS - Q&A**