CMPE 282 Cloud Services
NoSQL

Instructor: Kong Li

Content

- CAP
- Eventual Consistency
- NoSQL
- Cassandra
- HBase
- MongoDB
- CouchDB
- Lab: Cassandra, MongoDB

Big Data

- Characteristics
  - Volume: MB, GB, TB, PB, etc
  - Variety: different forms or types
  - Velocity: batch, near realtime, realtime
- Search for actionable insights
  - Regardless of structured, semi-structured, or unstructured data
  - Q: How to analyze structured, semi-structured, and unstructured data?
- Evolution: Batch ➔ real time ➔ prediction
- Tools
  - Generic: NoSQL, SQL, search
  - Batch: MapReduce, Hive, Pig, etc.
  - Real time / streaming: Spark (streaming), Storm, etc
  - Machine learning: Mahout, Spark ML, etc
- Q: how to use the right tool for the job?
  - http://www.slideshare.net/AmazonWebServices/aws-november-webinar-series-architectural-patterns-best-practices-for-big-data-on-aws

Brewer's CAP Theorem

- Three desirable properties when designing distributed sys
  - **C**onsistency: All copies have the same value
  - **A**vailability: System can run even if parts have failed
  - **P**artition-tolerant: Survive network partitioning
- It is *impossible* to achieve all three in
  - Async networks: no clock; node makes decision based on msg received and local computation
  - Partially sync networks: each node has local clock; all clocks increase at the same rate; clocks are not synchronized
- Any two of these three can be achieved
- Examples
- Large systems will partition at some point ➔ availability or consistency?
  - Traditional DB chooses consistency
  - Most web apps choose availability (exception: order processing, etc.)
- CAP theorem only matters when there is a partition

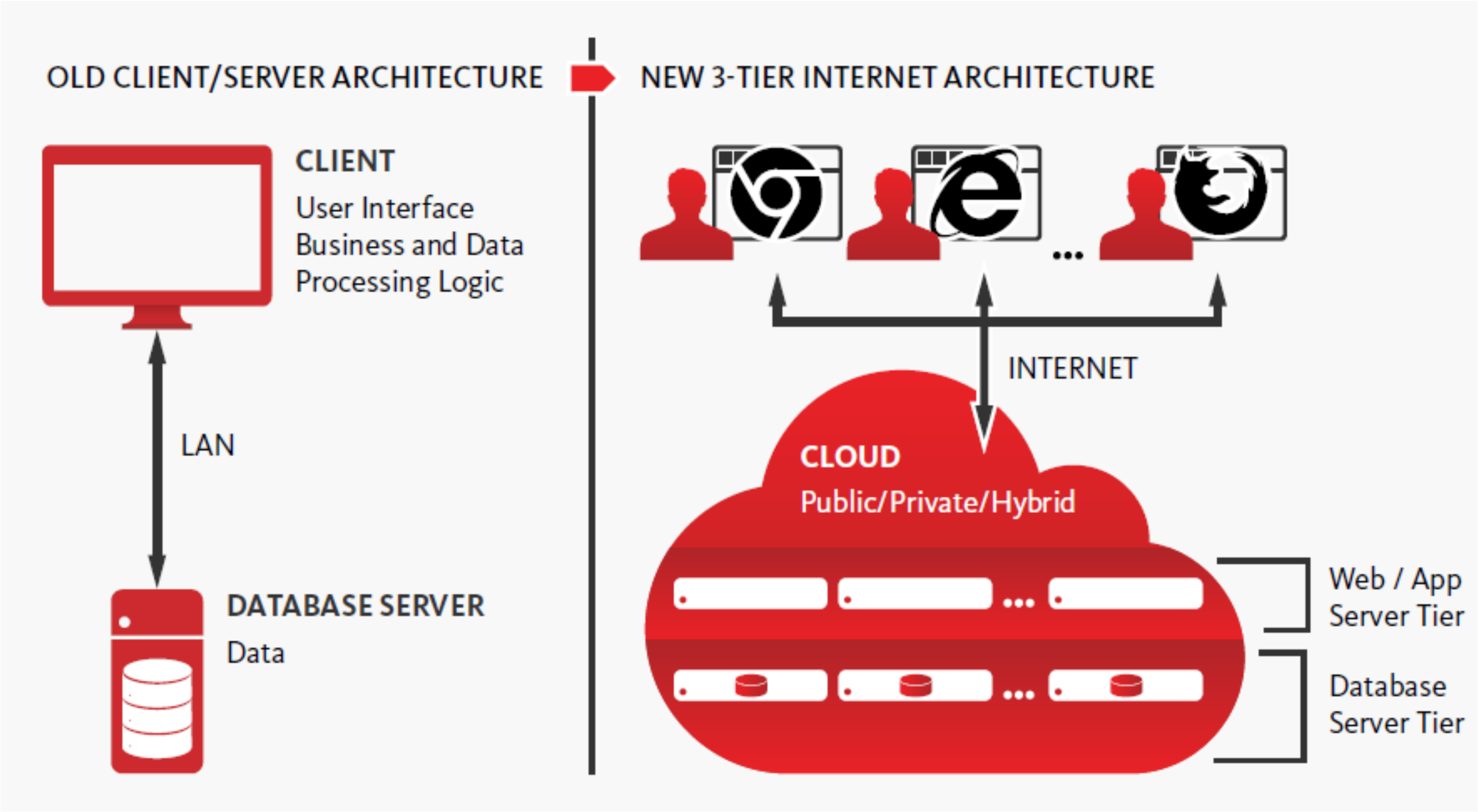
Eventual Consistency

- Eventual consistency: when no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
  - For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
  - You may not know how long it may take
- Known as **BASE** (**B**asically **A**vailable, **S**oft state, **E**ventual consistency), as opposed to ACID
  - **Soft state**: copies of a data item may be inconsistent
  - **Eventually Consistent** – copies becomes consistent at some later time if there are no more updates to that data item
- Used by most NoSQL
- Tradeoffs: consistency, availability, latency/performance

NoSQL

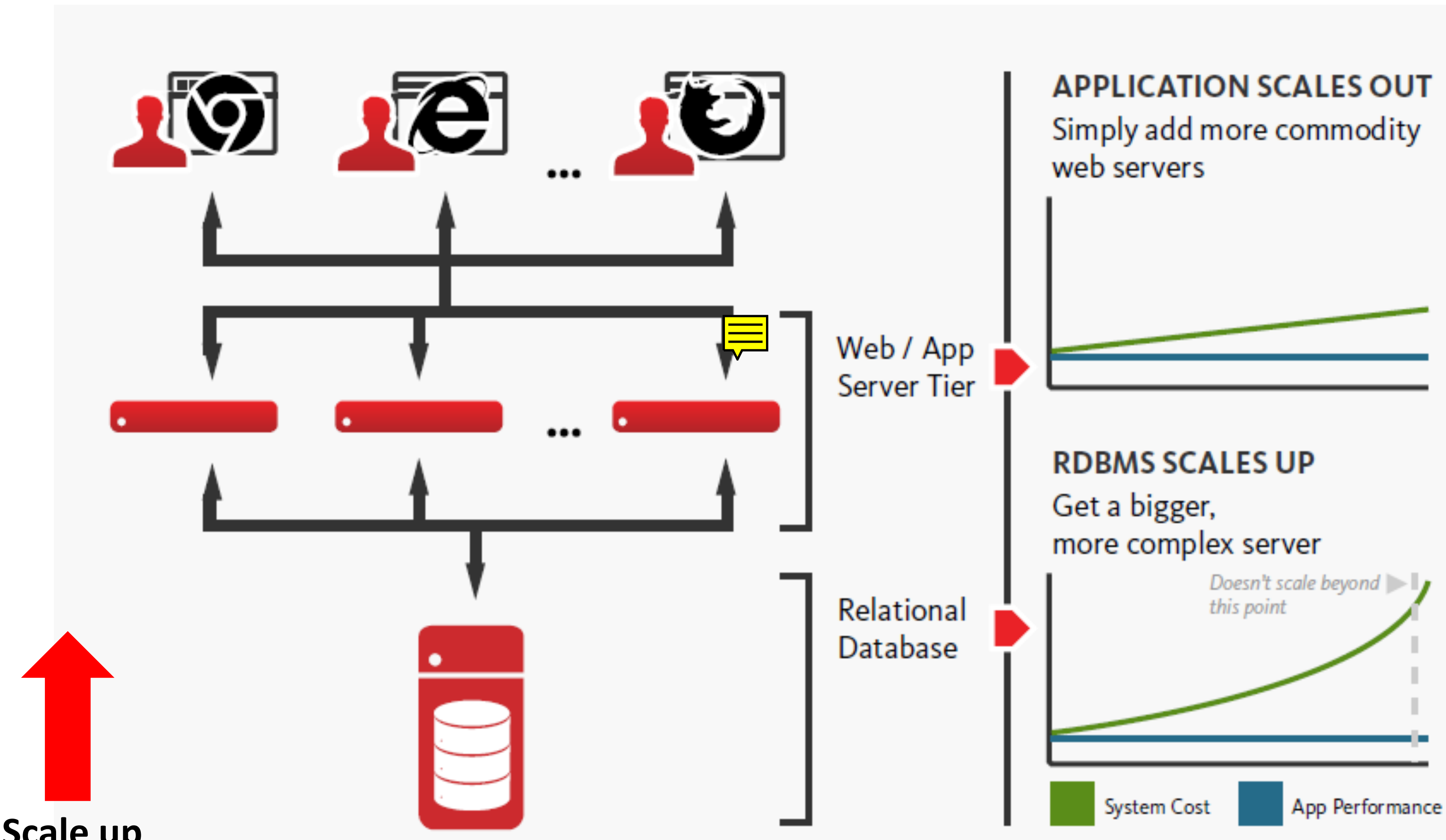
- Schema-less data stored as some form of key-value pair DB
  - APIs: get(key), set(key, value)
- Simpler functionality
- Scalability: scale out
- Eventual consistency (BASE)
- Each addresses a specific set of issues that RDBMS do not
  - none provides a panacea for all issues
- Some gradually brings back functionalities from RDBMS
- “No to SQL” ➔ “Not only SQL”
- Categories
  - Key-value: AWS DynamoDB, Azure Tables, Google Cloud Datastore, Riak, Redis
  - Column-family/BigTable: Cassandra, HBase, Hypertable
  - Document: MongoDB, CouchDB, AWS DynamoDB
  - Graph: Neo4j

The Cloud



OLD CLIENT/SERVER ARCHITECTURE
NEW 3-TIER INTERNET ARCHITECTURE
CLIENT
User Interface
Business and Data
Processing Logic
LAN
DATABASE SERVER
Data
INTERNET
CLOUD
Public/Private/Hybrid
Web / App
Server Tier
Database
Server Tier

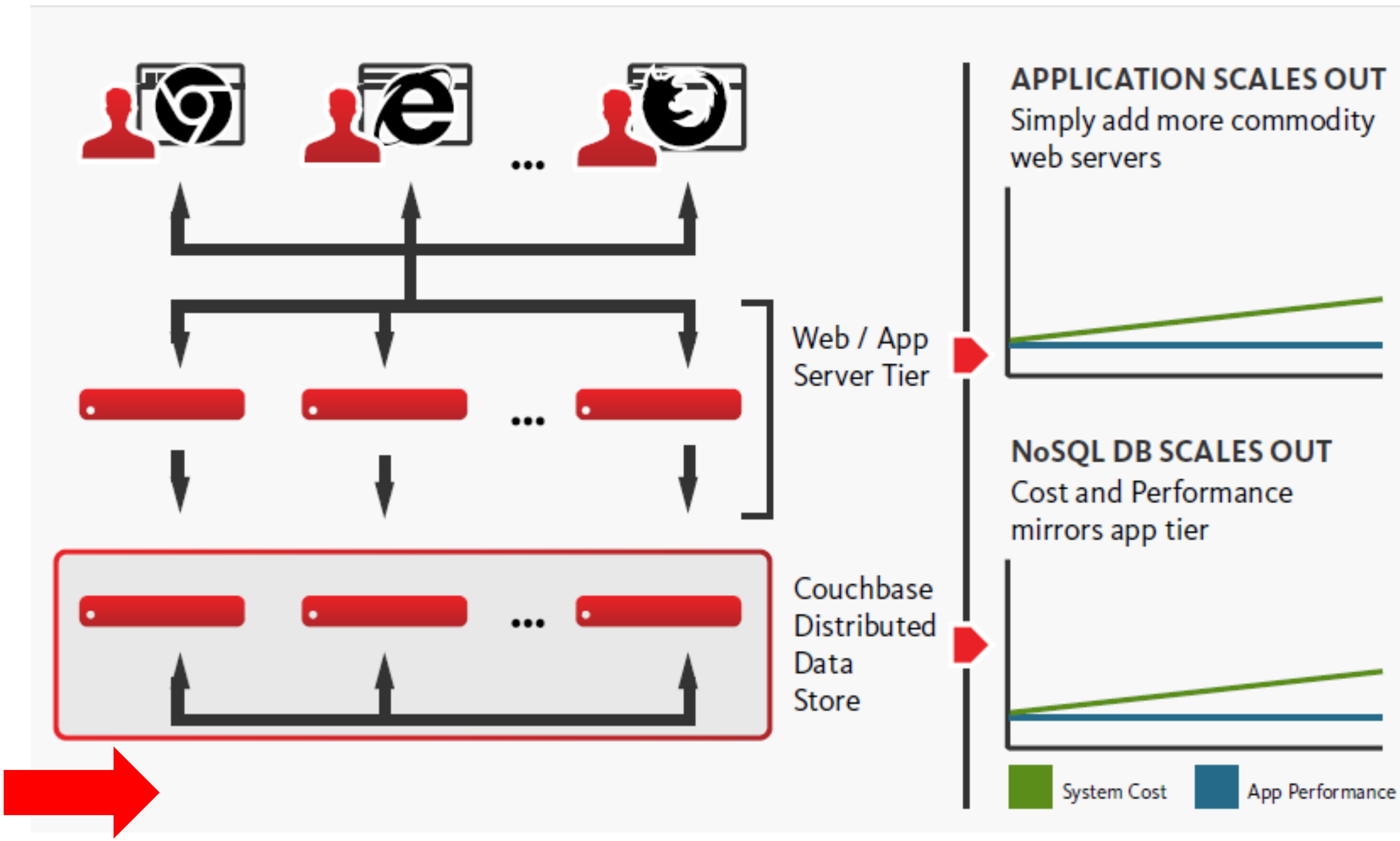
Scalability of RDBMS



Scale up

Scale out can be done but more difficult

Scalability of NoSQL



Scale out

RDBMS vs NoSQL

	RDBMS	NoSQL
	Schema: tables/columns	Schema-less: key-value pairs
Data	Structured data	Semi- and un-structured data
Process	Requirements ➔ Data model (ERD, etc) ➔ tables	Store data first ➔ (Transform data) ➔ get/put
Data association	Referential integrity, foreign keys	You are on your own
Functionality	Complex	Simple (➔ Better performance?)
Properties	ACID	BASE
Pros	(strong) consistency Transactions Joins	Schema-less Scalability, Availability Performance
Cons	Scalability Availability	Eventual consistency Tricky join No transactions

NoSQL – Key-value Pair

- Simple architecture
  - a unique identifier (key) maps to an obj (value)
  - DB itself does not care about obj
- Good for
  - Simple data model
  - Scalability
- Bad for
  - “Complex” datasets
- Examples:
  - AWS DynamoDB, Azure Tables, Google Cloud Datastore, Riak, Redis, Azure CosmosDB

NoSQL – Column-family/BigTable

- Key-value pair + grouping
  - Key – {a set of values}
  - E.g., time series data from sensors, languages of web page
- Good for
  - More complex data set (than simple key-value pair)
  - Scalability
- Examples
  - Cassandra
  - HBase
  - Hypertable
  - Azure CosmosDB

NoSQL – Document

- DB knows the nature of the data
  - Document – JSON
  - Need index to improve performance
- Scalability – clustering, replication, some w/ partition-tolerant
- Good for
  - Systems already using document-style data
  - Scalability
- Examples: MongoDB, CouchDB, AWS DynamoDB, Azure CosmosDB

NoSQL – Graph

- How data is related and what calculation is to be performed
- Usually also has “transaction”
- Good for
  - Interconnected Data and non-tabular
    - Geospatial problems, recommendation engine, network analysis, bioinformatics
  - “Closeness” of relationship
    - How vulnerable a company is to “bad news” for another company
- Bad for
  - Simple tabular data
- Examples: Neo4j, Azure CosmosDB

Cassandra

- Scalable HA NoSQL, key-value pair + column family (≈ table in RDBMS)
- Cluster spanning multiple data centers (DCs): linear scalability
- Data model: a partitioned row store
  - Rows are organized into tables
  - Partition: consistent hashing - decides storage node (≈ AWS DynamoDB)
    - adding/removing nodes in cluster only reshuffles keys among *neighboring* nodes
  - Tunable consistency: decided by client apps, per operation
    - Consistency level: 1, 2, 3, quorum, all, any, etc.
    - W: sent to all replica, R: decided by consistency level
- Replication and multi-DC replication
  - asynchronous masterless (peer-to-peer) replication
  - Rack aware, rack unaware, datacenter aware: performance vs availability
- Cassandra Query Language (CQL): SQL like
  - No joins and subqueries
  - materialized views (i.e. pre-computed results of queries)
- Hadoop integration, w/ MapReduce support

controls how many responses the coordinator waits for before responding to the client

quorum ≈ AWS DynamoDB

NRW: Consistency vs Availability

- R and W: read quorum Q_R and write quorum Q_W

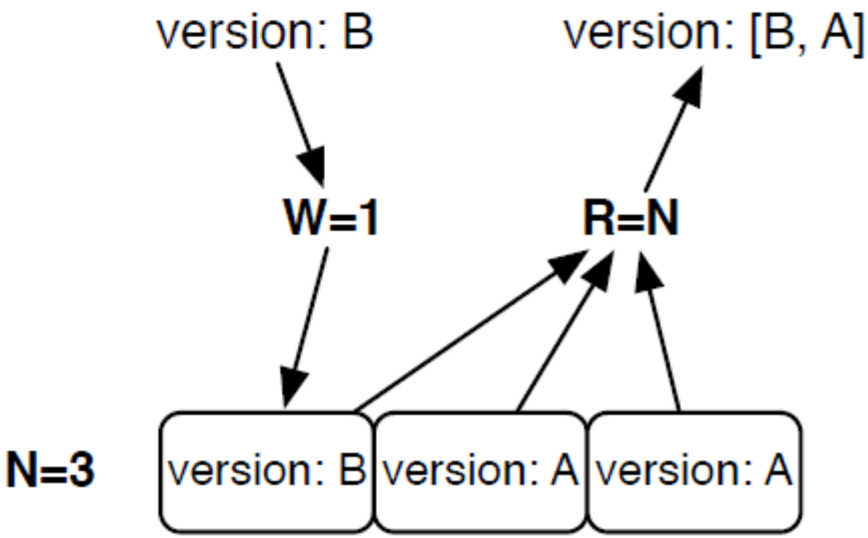


Figure 11—Consistency by reads: W=1, R=N

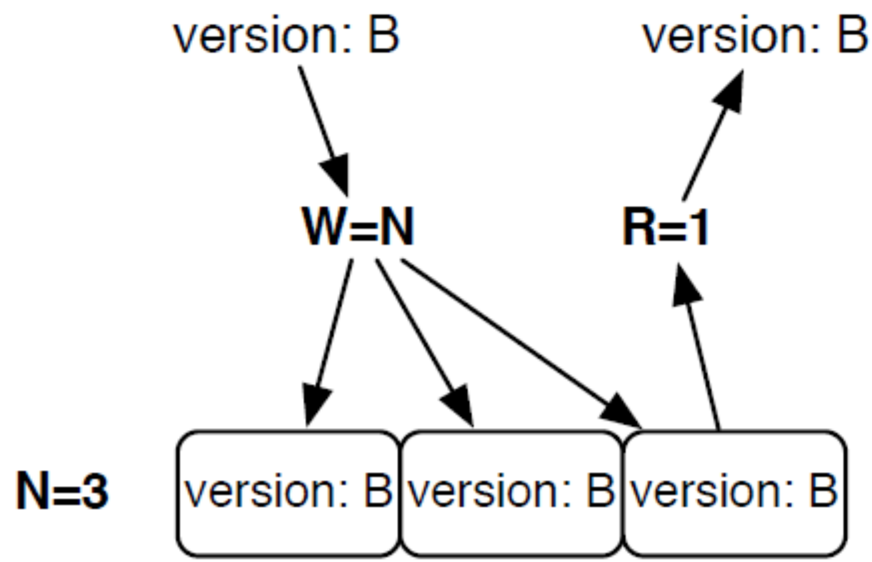


Figure 10—Consistency by writes: W=N, R=1

- Both are special case of consistency by quorum

NRW: Consistency vs Availability (cont'd)

- R and W: read quorum Q_R and write quorum Q_W

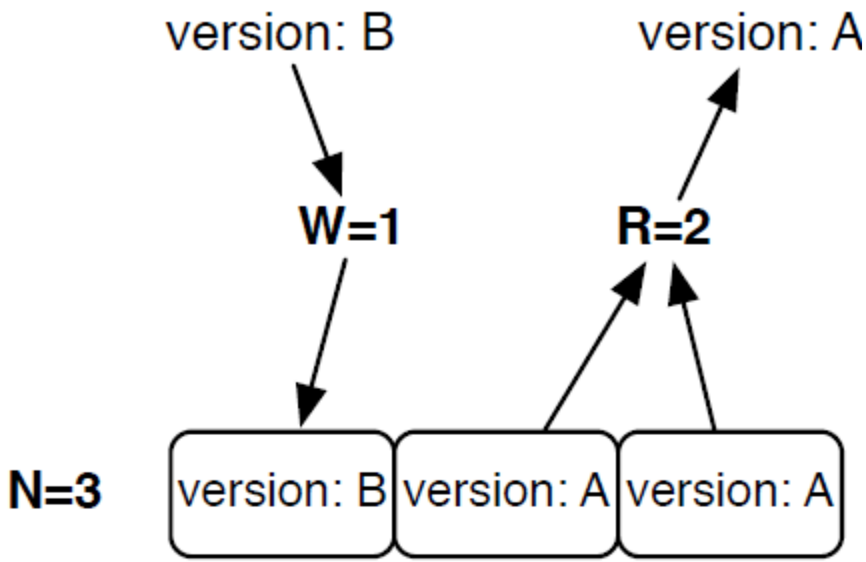


Figure 9—Eventual consistency: W+R <= N

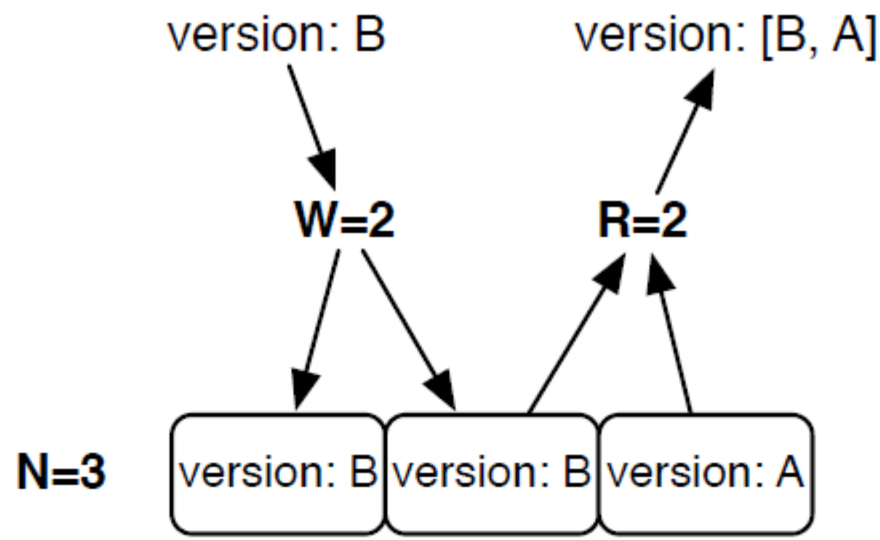
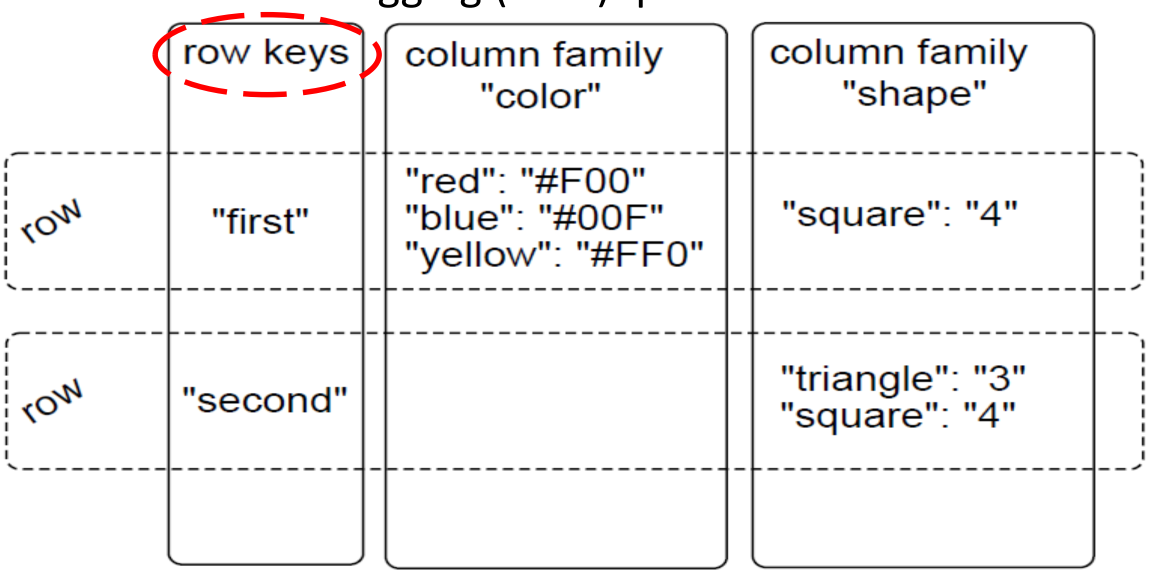


Figure 12—Consistency by quorum: W+R > N

AWS DynamoDB: R – strong/eventual consistent; W – unconditional/conditional
Cassandra: tunable consistency; per operation
Riak: tunable on a per-bucket, or per-request, basis

HBase

- Column-oriented DB based on Google BigTable
- “table”, “row”, and “column”: different from RDBMS
- Distributed, (timestamp) versioned, NoSQL DB on top of Hadoop and HDFS
- Cluster: scale out, failover
- CRUD: shell
- Strong consistent reads and writes: *not* eventual consistency
  - All operations are atomic at the *row level*, regardless of # of columns
  - Write ahead logging (WAL): protect from server failure



first/color:red = #F00

HBase (cont'd)

- Region-based storage architecture:
  - Region: a set of rows, [starting-row-key, ending-row-key), w/ a region server
    - Regions never overlap
  - *Auto sharding* of tables:
    - tables distributed on cluster via regions, regions automatically split & re-distributed
- HBaseMaster: master node
  - Assign regions to region servers
  - Automatic failover between *region servers*
  - Upon HBaseMaster failure, one region server takes over as master
- HDFS and HBase: rack-aware, data replication within & between DC racks
- Strength
  - Scale out architecture, rack-aware replication
  - Built-in versioning
- weakness
  - Complexity: + Hadoop + etc
  - Designed for Big, not for simple/small problem: minimum 5 nodes

MongoDB

- Scalable distributed document DB
  - Nested JSON document (binary form of JSON – BSON)
  - A document: a JSON obj w/ key-value pairs ≈ row in RDBMS
- CRUD: JavaScript, ACID at document level
- Emulate SP in RDBMS: executing a server-side function (.js)
- No joins: still allow to retrieve data via relationship
- B-tree based Indexing: performance
- map(), reduce(), and finalize()

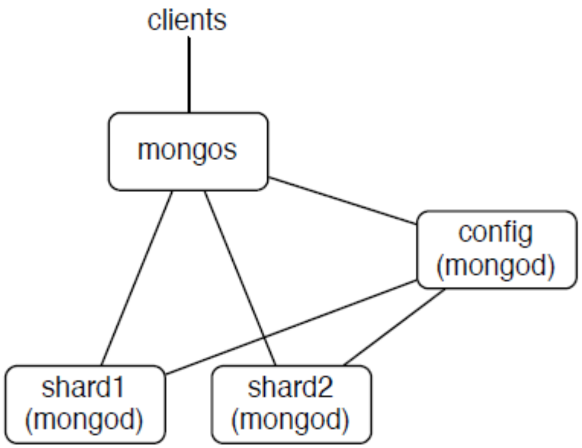
$lookup: left outer join for pipelined data aggregation

12 bytes
•globally unique
•Can be override

JSON syntax:
- Key-value: {“key”: “value”}
- , separated: {…}, {…}
- {} for obj: {“key”: {“k2”: “v2”}}
- [] for array: {“key”: [{…}, {…}]}

MongoDB (cont'd)

- *Replica Sets*
  - One server as primary (master), others as secondary (w/ replicated data)
  - Client only reads from or writes to master
  - When master is down, one of secondary server becomes primary (master)
  - *majority of nodes that can still communicate make up the network*
    - Quorum - require odd number of nodes in replica set, for consistency
- *Auto sharding* – range-based, hash-based, user-defined
  - mongod (config server): track how sharding is done
  - mongos: query router - entry point for clients
  - Data distribution: splitting, balancing
  - App transparent



- GridFS: distributed file system
- Strength
  - Scale out, sharding, replication, easy to use, similar query capability as RDBMS (w/o joins)
- Weakness: Too flexible? simple typo could cause debugging headache

CouchDB

- JSON- and REST-based document-oriented DB
- Each document
  - A set of key-val, val is any JSON structure nested to any depth (like MongoDB)
  - Unique immutable _id: auto-gen or explicit assigned per doc
  - _rev: revision per doc, starting from 1, auto-modified after doc changes
  - Each update/delete must specify *both* _id and _rev (why?)
  - Can be nested (like MongoDB)
- CRUD: Futon web interface
  - RESTful (GET: read, POST: create, PUT: update or create, DELETE: delete)
  - Update: overwrite the *entire* doc (unlike MongoDB - modify *in place*)
- No transaction, no locking
- View: ordered list of key-value pairs, indexed, key and val can be any JSON
  - Node.js (server-side JavaScript); can be persisted as a design document
  - Content generated by mapreduce functions incrementally (~ materialized view)

CouchDB (cont'd)

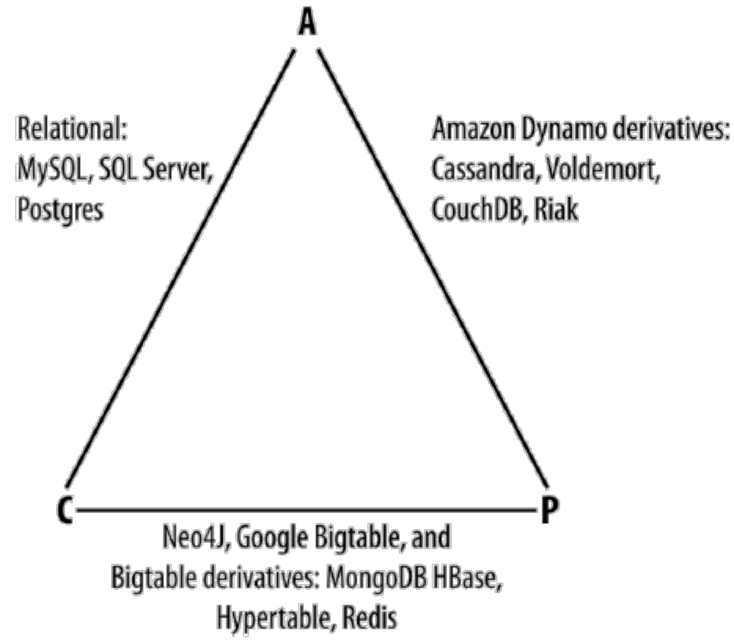
- Changes API – monitor *any* changes to documents
  - polling, long-polling, and continuous
  - Can develop non-blocking event-driven client apps
- Replication
  - All or nothing; no sharding
  - Multi-master replication: HA
  - Conflict resolution: same _id + _rev – all nodes choose the same winner
- Strength
  - Variety of deployment: smart phone to DC
  - Indexed view built by mapreduce incremntally
  - Multi-master replication, HA
- Weakness
  - Mapreduce-based view: limited when compared with views in RDBMS
  - All or nothing replication; no sharding

NoSQL Pros and Cons

- Pros
  - schema-less
  - scalability (scale out), availability
  - performance
  - simpler change mgmt (“agile”, faster dev)
- Cons
  - schema-less
  - eventual consistency
  - tricky (server-side) join
  - No transaction
- Trade-offs (w/ RDBMS): functionality, availability, consistency, durability, scalability, performance
- To SQL or NoSQL?
  - It depends; likely co-exist
  - both provide features the other doesn't
    - RDBMS: valuable where data consistency is required

CAP Again

- MongoDB and HBase
  - Consistent and partition-tolerant (CP)
  - During network partition, unable to respond is possible (availability)
- Cassandra
  - Availability and partition-tolerant (AP)
  - Tunable consistency; per operation
- CouchDB
  - Availability and partition-tolerant (AP)
- Some DB can be configured to modify the level/degree
  - MongoDB can be CA
  - CouchDB can be CP
- Values of N, R, W
  - AWS Dynamo: R - strong/eventual consistent; W – unconditional/conditional
  - Cassandra: tunable consistency
  - Riak: tunable on a per-bucket, or per-request, basis



Lab: Cassandra

- Server v 3.x or 2.2.4
  - cassandra.apache.org
  - Mac, Windows: www.planetcassandra.org/cassandra
    - set “cdc_raw_directory” in conf/cassandra.yaml
  - Install: cassandra.apache.org/doc/latest/getting_started/installing.html
- CQL: cassandra.apache.org/doc/latest/cql/index.html
  - Shell: cqlsh
- Java client driver: downloads.datastax.com/java-driver
- Java client
  - Tutorial: academy.datastax.com/demos/getting-started-apache-cassandra-and-java-part-i
  - GetStarted.java: gist.github.com/beccam/06c3283e5ee4a480a555

Lab: Cassandra (cont'd)

- Start server (console mode): cassandra -f
- Start CQL shell: cqlsh

Why does it look like RDBMS/SQL?

```
$ cqlsh
> desc keyspaces;
> CREATE KEYSPACE mydb WITH
    replication={'class': 'SimpleStrategy',
                 'replication_factor': '1'};
> desc mydb ;
> use mydb ;
> CREATE TABLE users (
firstname text,
lastname text,
age int,
email text,
city text,
PRIMARY KEY (lastname));
```

Tunable consistency

Supported data types include set, tuple, list, map, etc.

```
> desc table users;
> INSERT INTO users (firstname, lastname, age,
    email, city) VALUES ('John', 'Smith', 46,
    'johnsmith@email.com', 'Sacramento');
> INSERT INTO users (firstname, lastname, age,
    email, city) VALUES ('Jane', 'Doe', 36,
    'janedoe@email.com', 'Beverly Hills');
> SELECT * FROM users;
> SELECT * FROM users WHERE lastname= 'Doe';
> UPDATE users SET city= 'San Jose' WHERE
    lastname= 'Doe';
> DELETE from users WHERE lastname = 'Doe';
```

Lab: Cassandra (cont'd)

$ nodetool flush mydb

Convert memtables to sstables

$ sstabledump data/mydb/users-14c79820638311e7bf210b60b8eb8b8b/mc-1-big-Data.db

```
[
  {
    "partition" : {
      "key" : [ "Doe" ],
      "position" : 0
    },
    "rows" : [
      {
        "type" : "row",
        "position" : 17,
        "liveness_info" : { "tstamp" : "2017-07-08T02:14:40.267Z" },
        "cells" : [
          { "name" : "age", "value" : "36" },
          { "name" : "city", "value" : "San Jose", "tstamp" : "2017-07-08T02:15:
45.805Z" },
          { "name" : "email", "value" : "janedoe@email.com" },
          { "name" : "firstname", "value" : "Jane" }
        ]
      }
    ]
  },
  {
```

Lab: MongoDB

- Server: community edition
  - www.mongodb.org/downloads
  - Install: docs.mongodb.com/manual/installation/
  - Case sensitive
- Shell: docs.mongodb.org/getting-started/shell/client/
- Java client driver
  - oss.sonatype.org/content/repositories/releases/org/mongodb/mongo-java-driver/3.4.2/mongo-java-driver-3.4.2.jar
  - mongodb.github.io/mongo-java-driver
- Java client:
  - mongodb.github.io/mongo-java-driver/3.4/driver/getting-started/quick-start/

Lab: MongoDB (cont'd)

- Start server on Ubuntu: sudo service mongod start
- Start server on windows: Windows: mkdir /data/db/; mongod.exe
- Start client: mongo

```
$ mongo
> show dbs
> use mydb
> db
> db.inventory.insert(
  {item: "ABC1",
    details: {model: "14Q3", madeby: "XYZ
     Inc"},
    stock: [ { size: "S", qty: 25 }, { size: "M",
     qty: 50 } ],
    category: "clothing"
  })
> show collections
> db.inventory.find()
```

```
> db.inventory.insert({ item: "ABC2", …})
> db.inventory.find( {item: "ABC2"} )
> db.inventory.count()
> db.inventory.update({ item: "ABC2" }, {
    item: "ABC2", category: "cookware" })
> db.inventory.remove({item : "ABC"})
> db.help()
> db.inventory.help()
> db.inventory.remove({})
> show collections
> db.inventory.drop()
> show collections
> db.dropDatabase()
```

References

- Seven Databases in Seven Days
- Cassandra: http://cassandra.apache.org/
- Cassandra then and now
  - http://docs.datastax.com/en/articles/cassandra/cassandrathenandnow.html
- MongoDB: https://www.mongodb.org/
- CouchDB: http://couchdb.apache.org/
- Couchbase: http://www.couchbase.com/
- Riak: http://basho.com/products/#riak
- HBase: http://hbase.apache.org/
- 7 hard truths about the NoSQL revolution
  - http://www.infoworld.com/article/2617405/nosql/7-hard-truths-about-the-nosql-revolution.html
- AWS DynamoDB: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html#WorkingWithItems.ReadingData
- https://d0.awsstatic.com/whitepapers/AWS_Comparing_the_Use_of_DynamoDB_and_HBase_for_NoSQL.pdf