VINCITORE PRINCIPI ATTIVI 2012

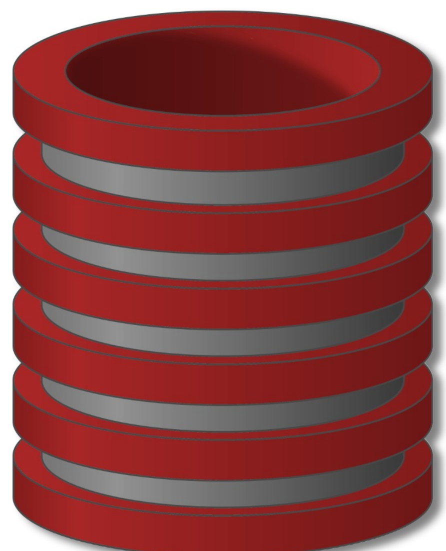
Regione Puglia
Assessorato Politiche Giovanili
e Cittadinanza Sociale

Presidenza del Consiglio dei Ministri
Dipartimento della Gioventù
e del Servizio Civile Nazionale

DTK

Database NoSQL

Quando, vantaggi e svantaggi



Dr. Fabio Fumarola

Scaling Up Databases

A question I'm often asked about Heroku is: "How do you scale the SQL database?" There's a lot of things I can say about using caching, sharding, and other techniques to take load off the database. But the actual answer is: we don't. SQL databases are fundamentally non-scalable, and there is no magical pixie dust that we, or anyone, can sprinkle on them to suddenly make them scale.

Adam Wiggins Heroku

Adam Wiggins, Heroku Patterson, David; Fox, Armando (2012-07-11). **Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing**, Alpha Edition (Kindle Locations 1285-1288). Strawberry Canyon LLC. Kindle Edition.

Data Management Systems: History

- In the last decades RDBMS have been successful in solving problems related to storing, serving and processing data.
- RDBMS are adopted for:
  - Online transaction processing (OLTP),
  - Online analytical processing (OLAP).
- Vendors such as Oracle, Vertica, Teradata, Microsoft and IBM proposed their solution based on Relational Math and SQL.

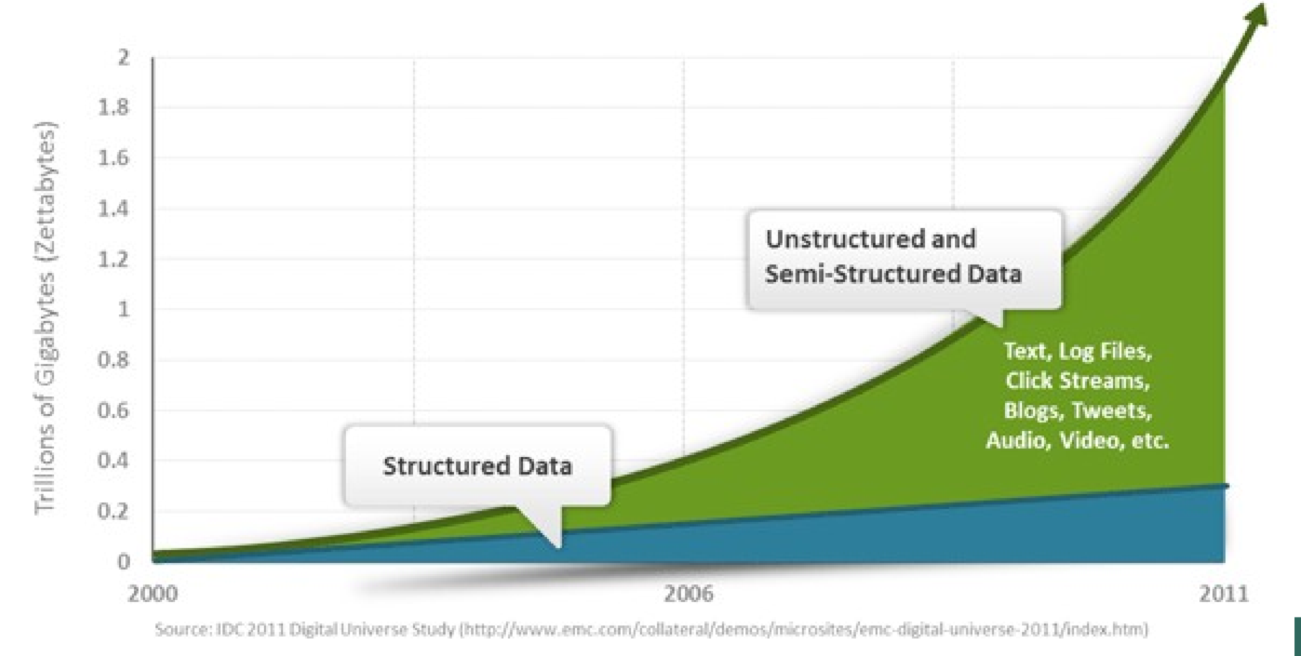
But….

Something Changed!

- Traditionally there were transaction recording (OLTP) and analytics (OLAP) of the recorded data.
- Not much was done to understand:
  - the reasons behind transactions,
  - what factor contributed to business, and
  - what factor could drive the customer's behavior.
- Pursuing such initiatives requires working with a large amount of varied data.

Something Changed!

- This approach was pioneered by Google, Amazon, Yahoo, Facebook and LinkedIn.
- They work with different type of data, often semi or un-structured.
- And they have to store, serve and process huge amount of data.



Something Changed!

- RDBMS can somehow deal with this aspects, but they have issues related to:
  - expensive licensing,
  - requiring complex application logic,
  - Dealing with evolving data models
- There were a need for systems that could:
  - work with different kind of data format,
  - Do not require strict schema,
  - and are easily scalable.

Evolutions in Data Management

- As part of innovation in data management system, several new technologies where built:
  - 2003 - Google File System,
  - 2004 - MapReduce,
  - 2006 - BigTable,
  - 2007 - Amazon DynamoDB
  - 2012 Google Cloud Engine
- Each solved different use cases and had a different set of assumptions.
- All these mark the beginning of a different way of thinking about data management.

Go to hell RDBMS!

Hello, Big Data!

Big Data: Try { Definition }

Big Data means the data is large enough that you have to think about it in order to gain insights from it

Or

Big Data when it stops fitting on a single machine

“Big Data, is a fundamentally different way of thinking about data and how it’s used to drive business value.”

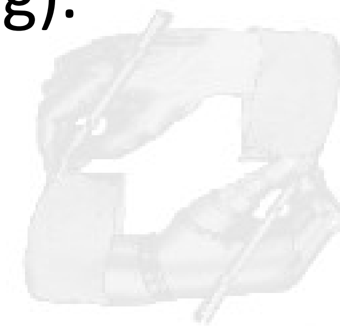
NoSQL

NoSQL

- In 2006 Google published BigTable paper.
- In 2007 Amazon presented DynamoDB.
- It didn’t take long for all these ideas to used in:
  - Several open source projects (Hbase, Cassandra) and
  - Other companies (Facebook, Twitter, …)
- And now? Now, nosql-database.org lists more than 150 NoSQL databases.

NoSQL related facts

- Explosion of social media sites (Facebook, Twitter) with large data needs.
- Rise of cloud-based solutions such as Amazon S3 (simple storage solution).
- Moving to dynamically-typed languages (Ruby/Groovy), a shift to dynamically-typed data with frequent schema changes.
- Functional Programming (Scala, Clojure, Erlang).



NoSQL Categorization

- Key Value Store / Tuple Store
- Column-Oriented Store
- Document Store
- Graph Databases
- Multimodel Databases
- Object Databases
- Unresolved and Uncategorized

https://github.com/hbaseinaction/twitbase

TwitBase Case Study

TwitBase: Data Model

Entities

- User(user: String, name: String, email: String, password: String, twitsCount: Int)
- Twit(user: String, datetime: DateTime, text: String)
- Relation(from: String, relation: String, to: String)

Design Steps:

1. Primary key definition
2. Data shape and access patterns definition
3. Logical model definition (Physical Model)

TwitBase: Actions

- Users:
  - add a new user,
  - retrieve a specific user,
  - list all the users
- Twit:
  - post a new twit on user’s behalf,
  - list all the twits for the specified user

TwitBase: Actions

- Relation
  - Add a new relationship where from follows to,
  - list everyone user-Id follows,
  - list everyone who follows user-Id
  - count users' followers
- The considered relations are follows and followedBy

Key-Value Store

Key Value Store

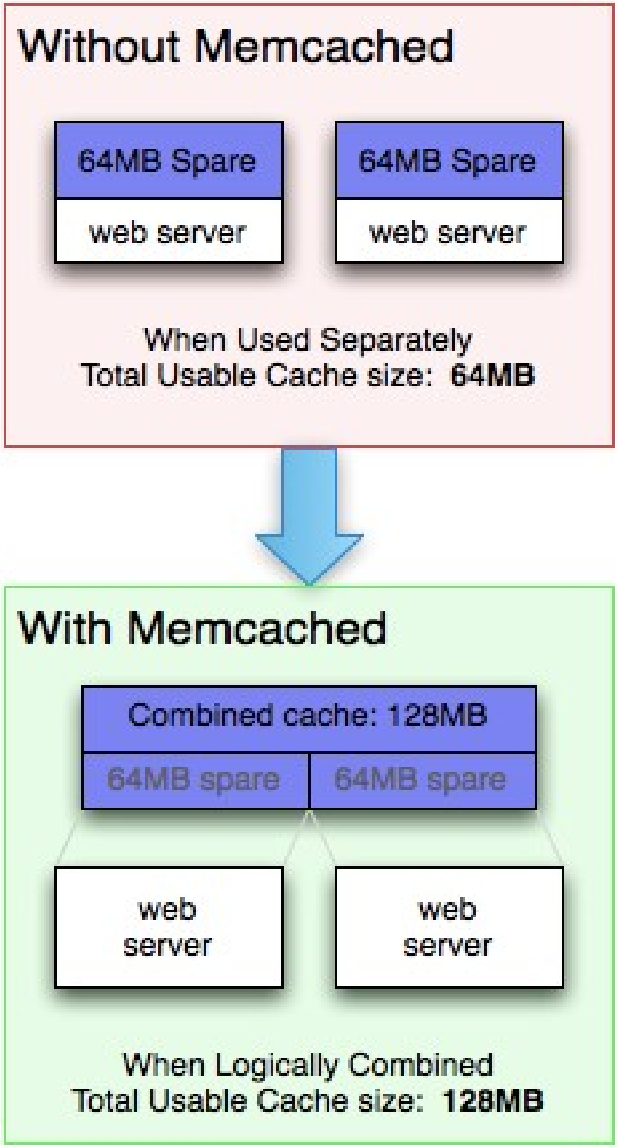
- Extremely simple interface:
  - Data model: (key, value) pairs
  - Basic Operations: : Insert(key, value), Fetch(key),Update(key), Delete(key)
- Values are store as a "blob":
  - Without caring or knowing what is inside
  - The application layer has to understand the data
- Advantages: efficiency, scalability, fault-tolerance

Key Value Store: Examples

- Memcached – Key value stores,
- Membase – Memcached with persistence and improved consistent hashing,
- Aerospike – fast key value for ssd disks,
- Redis – Data structure server,
- Riak – Based on Amazon's Dynamo (Erlang),
- Leveldb - A fast and lightweight key/value database library by Google,
- DynamoDB – Amazon Key Value Database.

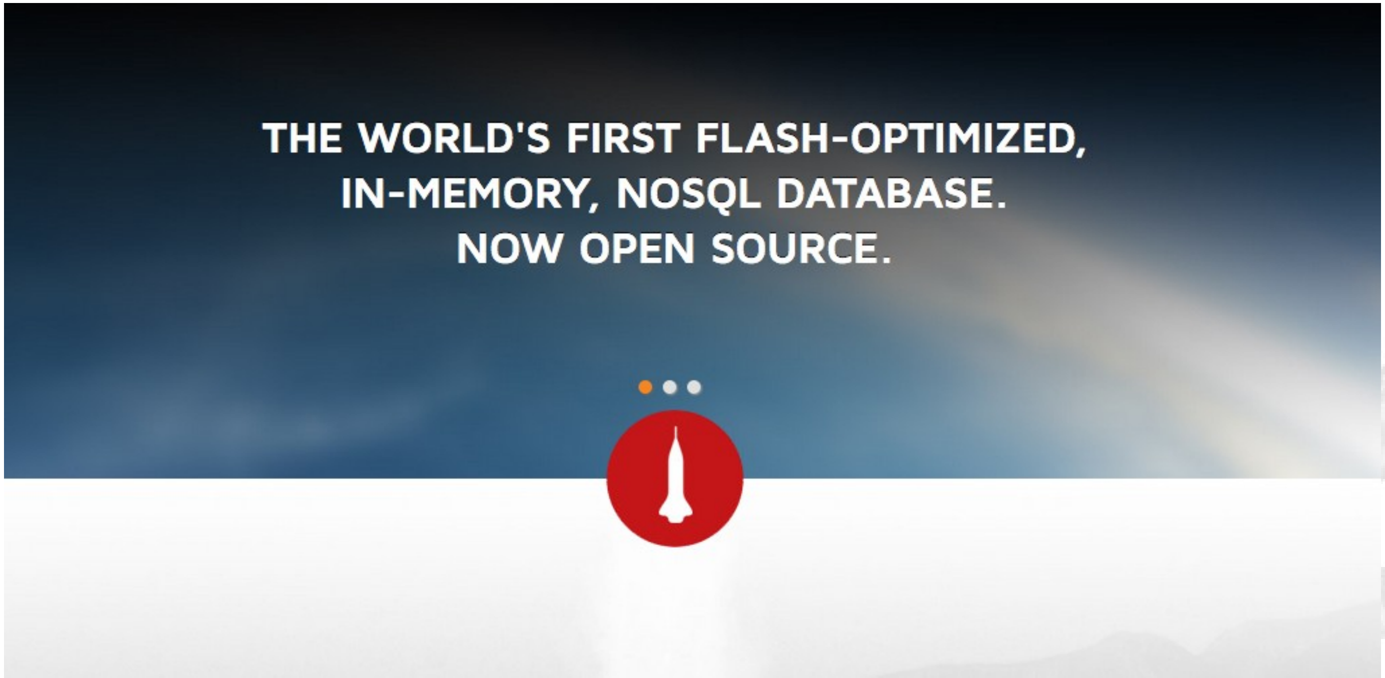
Memcached & MemBase

- Atomic operations set/get/delete.
- O(1) to set/get/delete.
- Consistent hashing.
- In memory caching, no persistence.
- LRU eviction policy.
- No iterators.



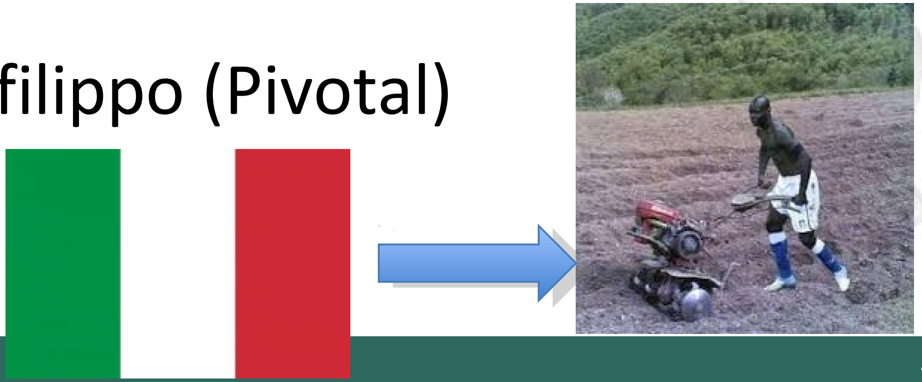
Aerospike

- Key-Value database optimized for hybrid (DRAM + Flash) approach
- First published in the Proceedings of VLDB (Very Large Databases) in 2011, “Citrusleaf: A Real-Time NoSQL DB which Preserves ACID”



Redis

- Written C++ with BSD License
- It is an advanced key-value store.
- It can contain strings, hashes, lists, sets and sorted sets.
- It works with an in-memory.
- data can be persisted either by dumping the dataset to disk every once in a while, or by appending each command to a log.
- Created by Salvatore Sanfilippo (Pivotal)



Riak

- Distributed Database written in: Erlang & C, some JavaScript
- Operations
  - GET /buckets/BUCKET/keys/KEY
  - PUT|POST /buckets/BUCKET/keys/KEY
  - DELETE /buckets/BUCKET/keys/KEY
- Integrated with Solr and MapReduce
- Data Types: basic, Sets and Maps

```
curl -XPUT 'http://localhost:8098/riak/food/favorite' \
  -H 'Content-Type:text/plain' \
  -d 'pizza'
```

LevelDB

LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values.

- Keys and values are arbitrary byte arrays.
- Data is stored sorted by key.
- The basic operations are Put(key ,value), Get(key), Delete(key).
- Multiple changes can be made in one atomic batch.

Limitation

- There is no client-server support built in to the library.

DynamoDB

- Fully managed NoSQL cloud database service
- Characteristics:
  - Low latency (< 5ms read, < 10ms to write) (SSD backend)
  - Massive scale (No table size limit. Unlimited storage)
- It run over ssd disk
- Cons: 64KB limit on row size, Limited Data Types: It doesn’t accept binary data, 1MB limit on Querying and Scanning

Redis: TwitBase

- Supported Data Types: strings, hashes, lists, sets and sorted sets
- TwitBase Domain Model:
  - User(user: String, name: String, email: String, password: String, twitsCount: Int)
  - Twit(user: String, datetime: DateTime, text: String)
  - Relation(from: String, relation: String, to: String)
- Design Steps:
  - Primary Key definition
  - Data shape and access patterns definition
  - Logical model definition (Physical Model)

Redis TwitBase: User

User(user: String, name: String, email: String, password: String)

- we can use the SET, HSET or HMSET operators

  SET users:1 {user: ‘pippo', name: ‘pippo basile’, email: ‘prova@mail.com’, password: ‘xxx’, count: 0}

  HSET users:1 user ‘pippo’

  HSET users:1 name ‘pippo basile’

  HMSET users:1 user ‘pippo’ name ‘pippo basile’

- Primary Key -> users:userId
- Operations

–add a new user -> SET, HSET or HMSET

–retrieve a specific user -> HGET or HKEYS/HGETALL

–list all the users -> KEYS users:* (What is the cost?)

http://redis.io/commands

Redis TwitBase: Twit

Twit(user: String, datetime: DateTime, text: String)

- we can use the SET or HSET operators

  SET twit:pippo:1405547914879 {user: ‘pippo', datetime: 1405547914879, email: ‘hello’}

  HSET twit:pippo:1405547914879 user ‘pippo’

  HSET twit:pippo:1405547914879 datetime 1405547914879

  HMSET twit:pippo:1405547914879 user ‘pippo’ datetime 1405547914879 …

- Primary Key-> twit:userId:timestamp (???)
- Operations

–post a new twit on user’s behalf -> SET, HSET or HMSET

–list all the twits for the specified user -> KEYS

http://redis.io/commands

Redis TwitBase: Relation

Relation(from: String, relation: String, to: String)

- we can use the SET, HSET, HMSET operators

  SET follows:1:2 {from: ‘pippo', relation: ‘follows’, to: ‘martin’}

  HMSET followed:2:1 from ‘martin’ relation ‘followedBy’, to ‘pippo’

- Primary Key:
- Operations

–add a new relation-> SET/HSET/HMSET

–list everyone user-Id follows -> KEYS

–list everyone who follows user-Id -> KEYS

–count users' followers -> any suggestion? What happen if we use LIST

LPUSH follows:1 {from: ‘pippo', relation: ‘follows’, to: ‘martin’} …

http://redis.io/commands

Key Value Store

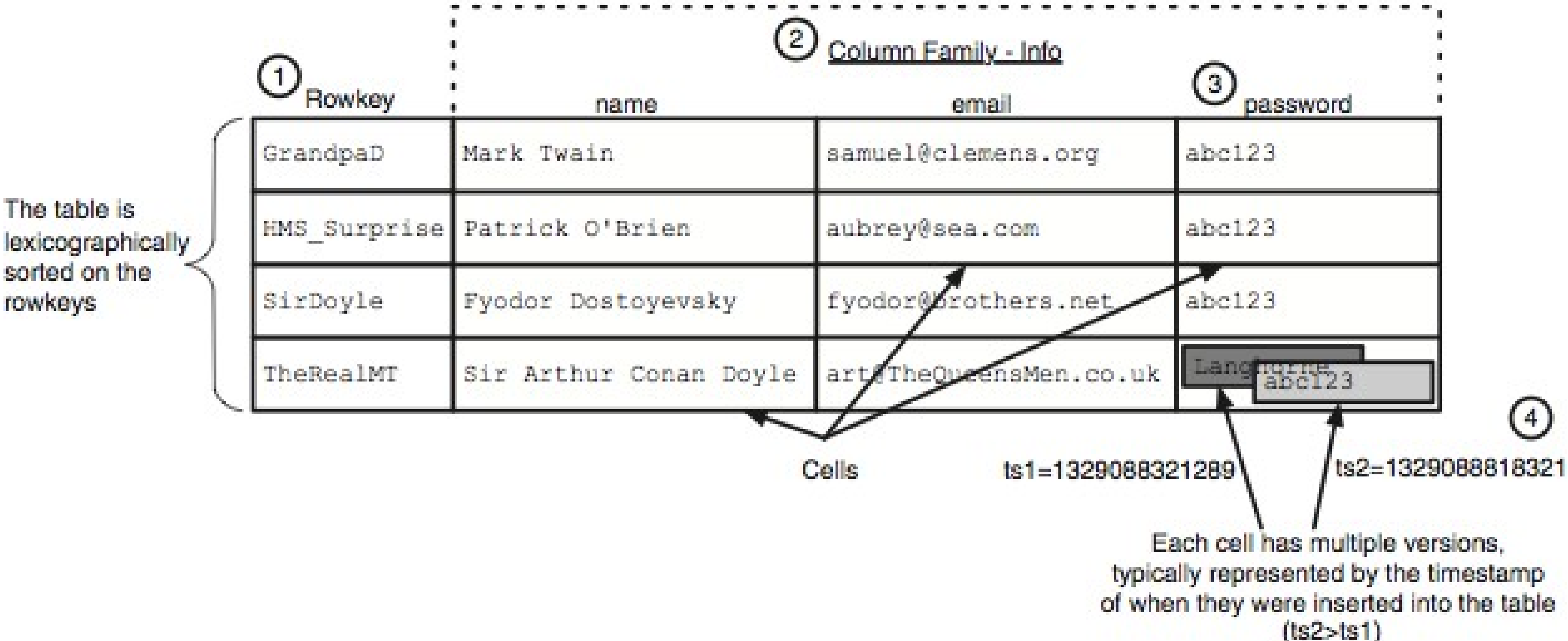
- Pros:
  - very fast
  - very scalable
  - simple model
  - able to distribute horizontally
- Cons:
  - many data structures (objects) can't be easily modeled as key value pairs

Column Oriented Store

Column-oriented

- Store data in columnar format
- Each storage block contains data from only one column
- Allow key-value pairs to be stored (and retrieved on key) in a massively parallel system
  - data model: families of attributes defined in a schema, new attributes can be added online
  - storing principle: big hashed distributed tables
  - properties: partitioning (horizontally and/or vertically), high availability etc. completely transparent to application

Column-oriented



Logical Model

Map<RowKey, Map<ColumnFamily, Map<ColumnQualifier, Map<Version, Data>>>>

Column Oriented Store

- BigTable
- Hbase
- Hypertable
- Cassandra

BigTable

- Project started at Google in 2005.
- Written in C and C++.
- Used by Gmail and all the other service at Google.
- It can be used as service (Google Cloud Platform) and it can be integrated with Google Big Query.

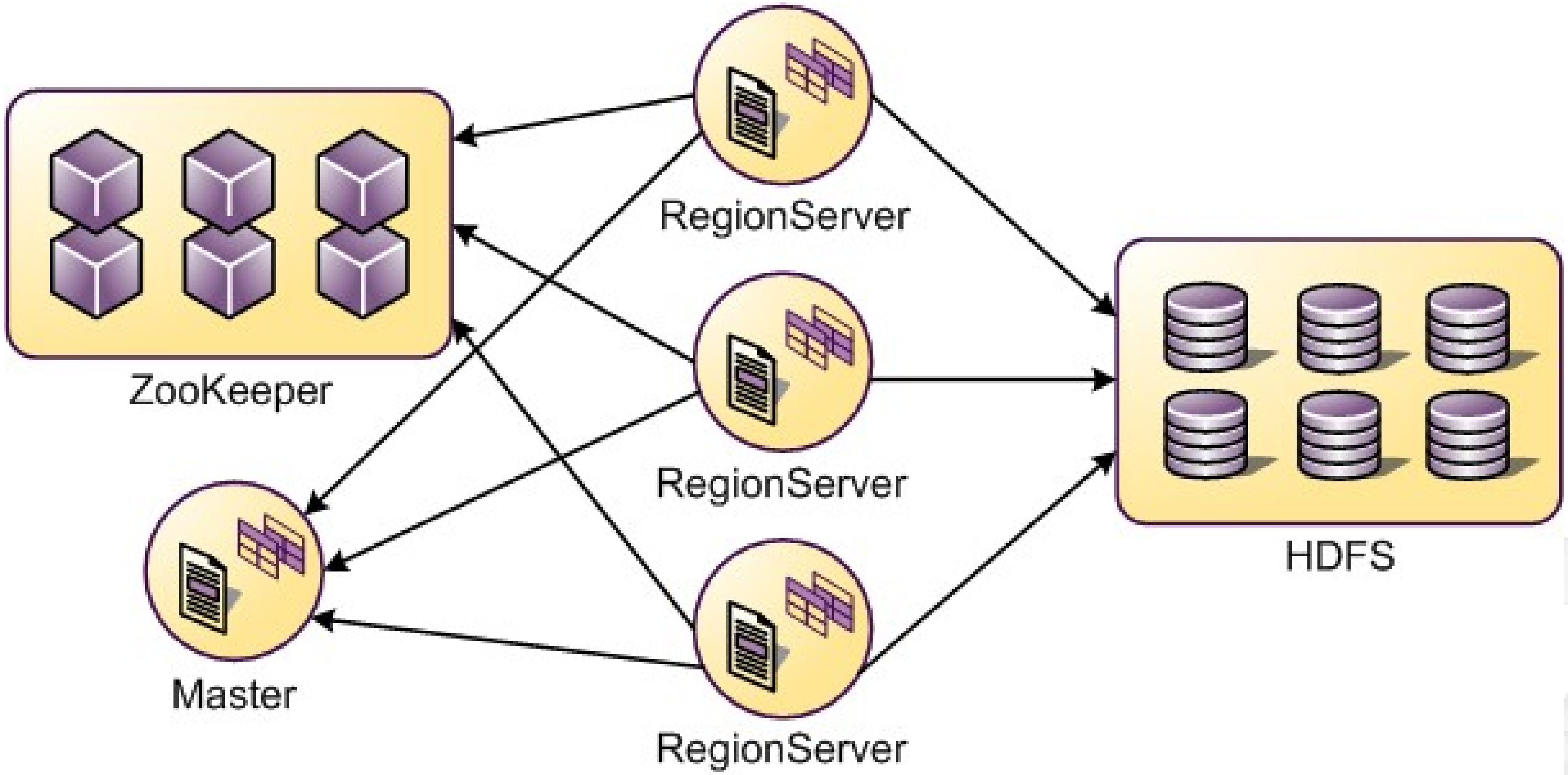
HBase

Apache HBase™ is the Hadoop database to use when you need you need random, realtime read/write access to your Data.

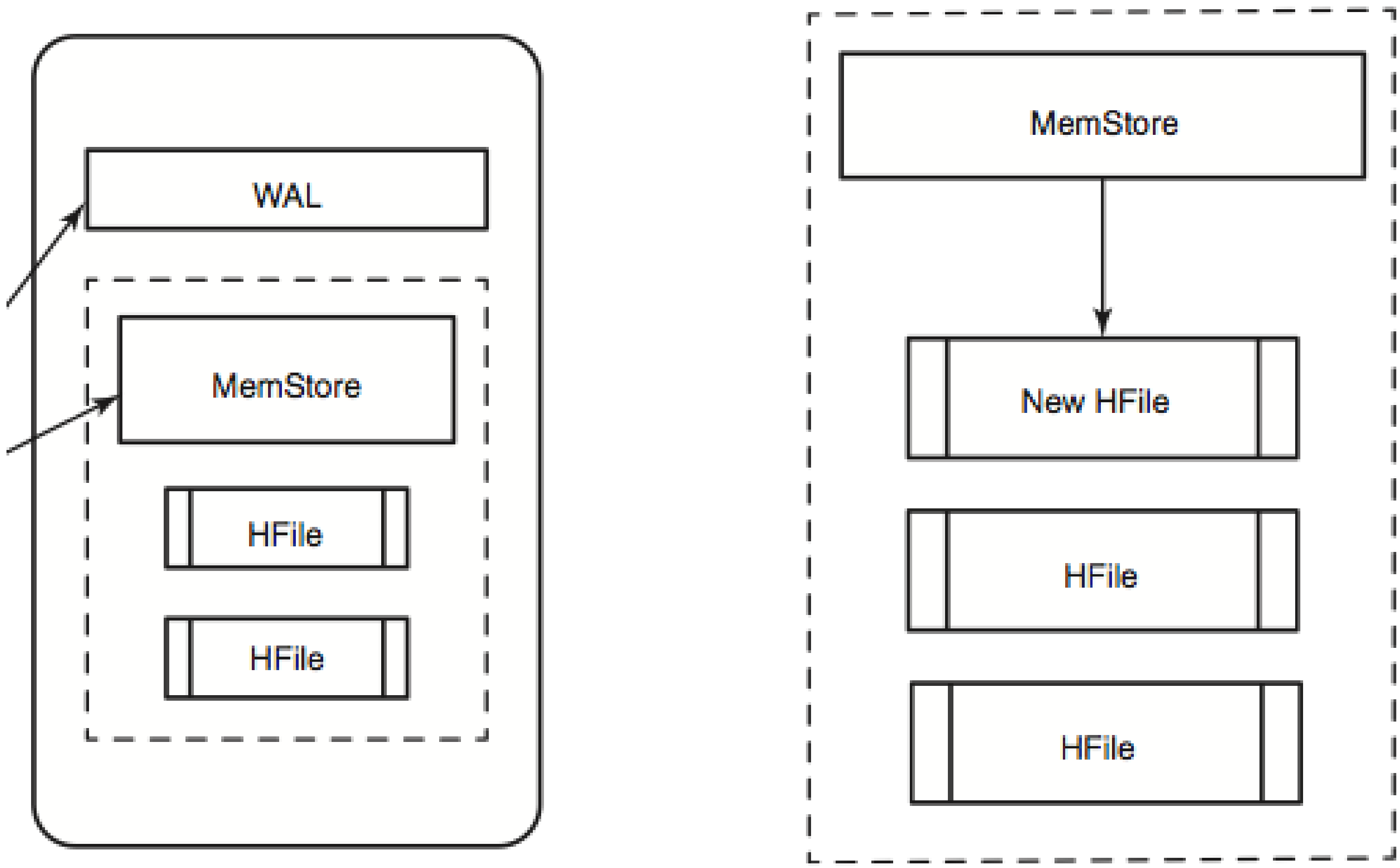
- Automatic and configurable sharding of tables
- Automatic failover support between RegionServers.
- Convenient base classes for backing Hadoop MapReduce jobs with Apache HBase tables.
- Easy to use Java API for client access.
- To be distributed, it has to run on top of hdfs
- Integrated with MapReduce

HBase

- Two roles: Master and Region Server



HBase: Data Manipulation



Hypertable

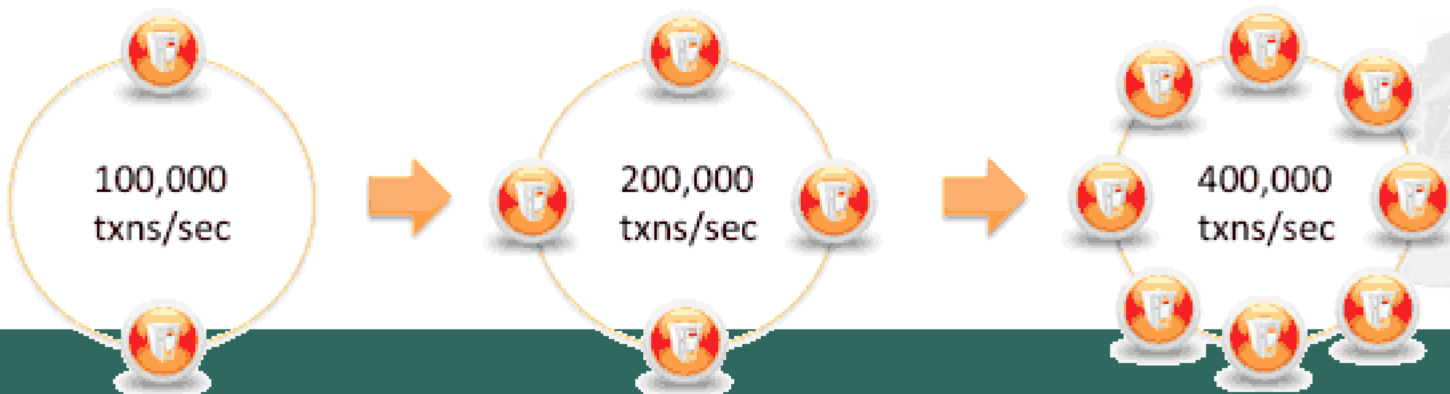
- Hypertable is an open source database system inspired by publications on the design of Google's BigTable.
- Hypertable runs on top of a distributed file system such as the Apache Hadoop DFS, GlusterFS, or the Kosmos File System (KFS). It is written almost entirely in C++.

BigTable – Hbase - Hypertable

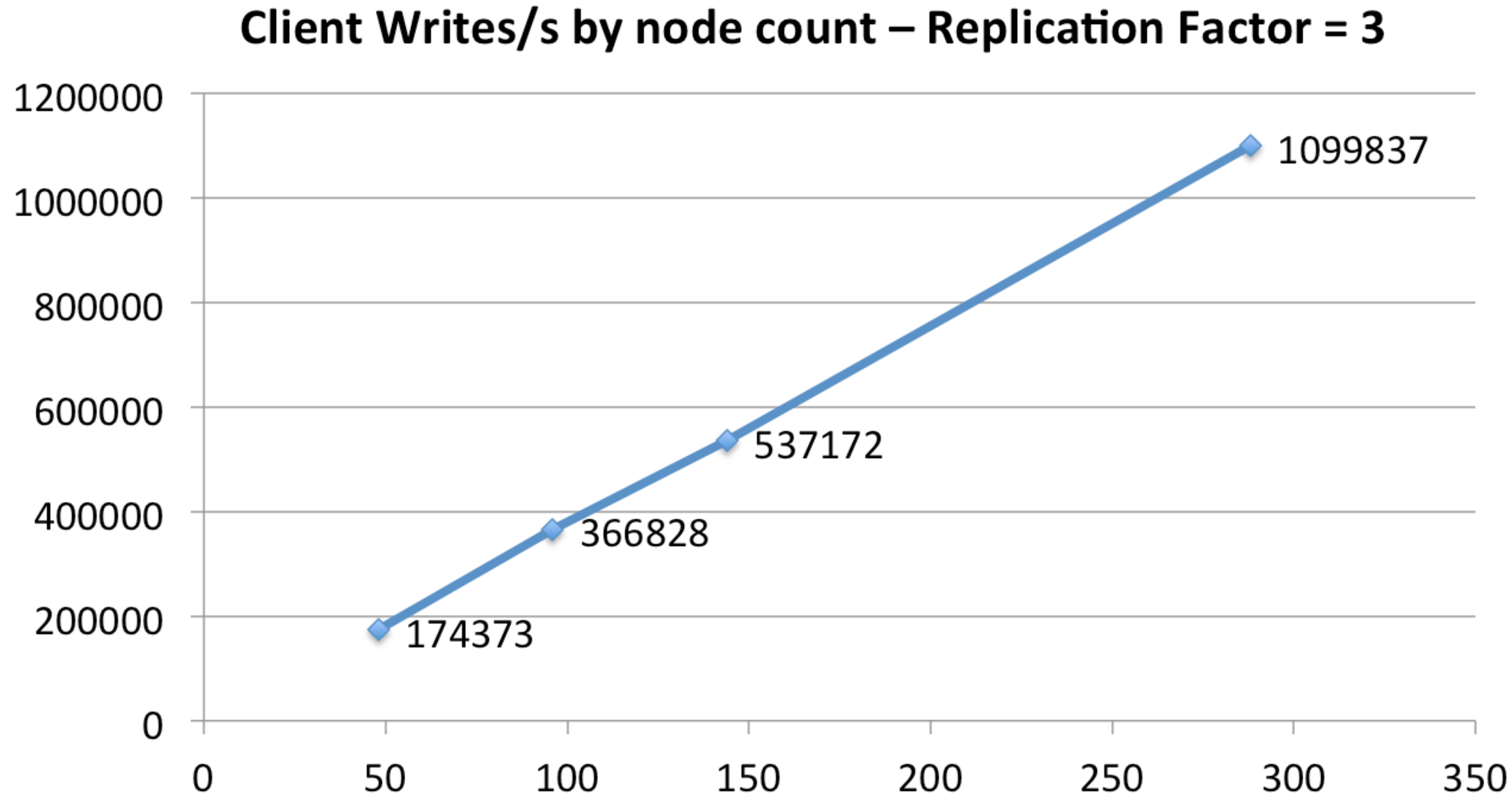
- Operator supported:
  - put(key, columnFamily, columnQualifier, value)
  - get(key)
  - Scan(startKey, endKey)
  - delete(key)
- Get and delete support optional column family and qualifier

Cassandra

- Big-Table extension:
  - All nodes are similar.
  - Can be used as a distributed hash-table, with an "SQL-like" language, CQL (but no JOIN!)
- Data can have expiration (set on INSERT)
- Map/reduce possible with Apache Hadoop
- Rich Data Model (columns, composites, counters, secondary indexes, map, set, list, counters)

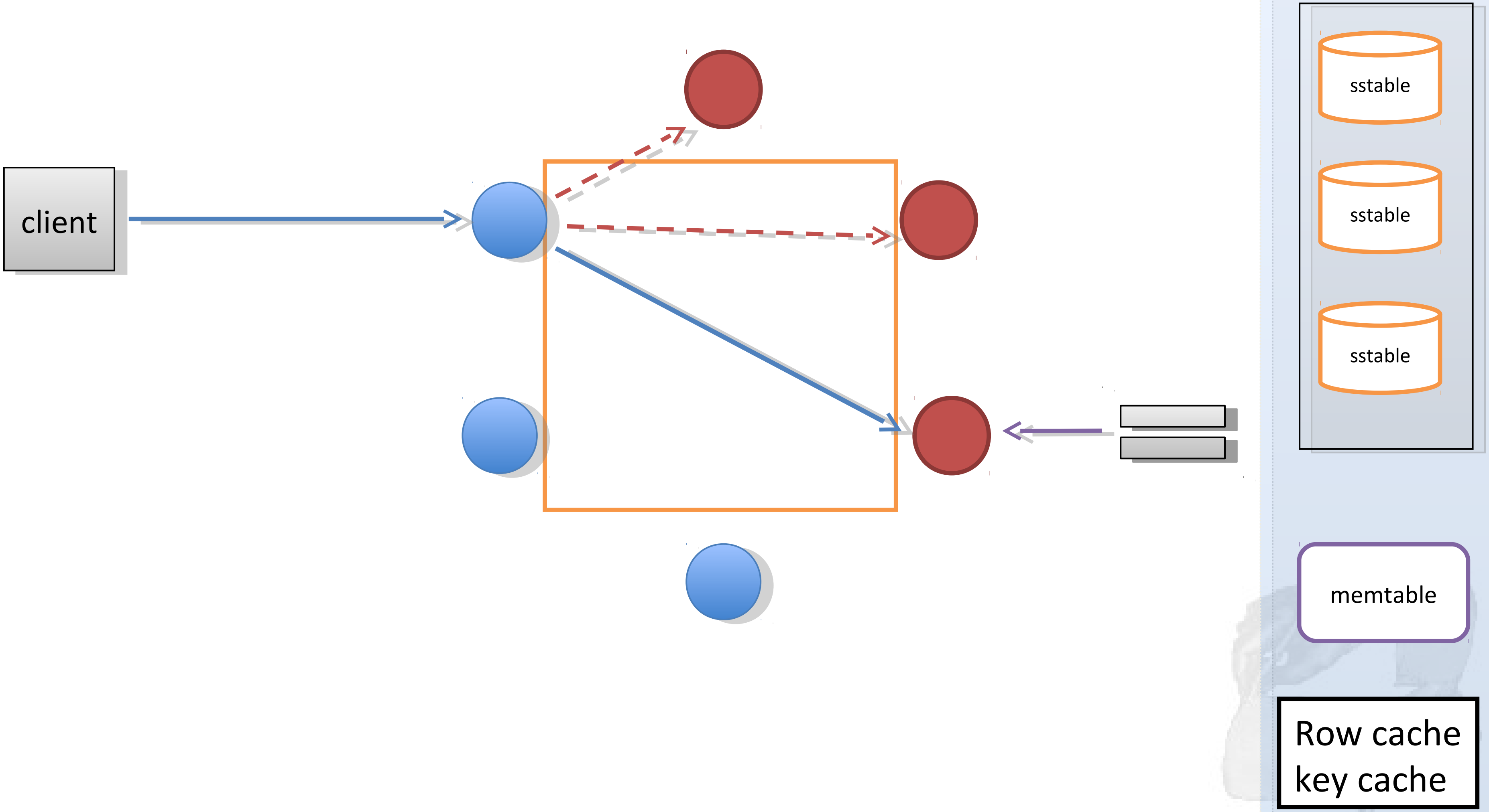


Proven Scalability and High Performances

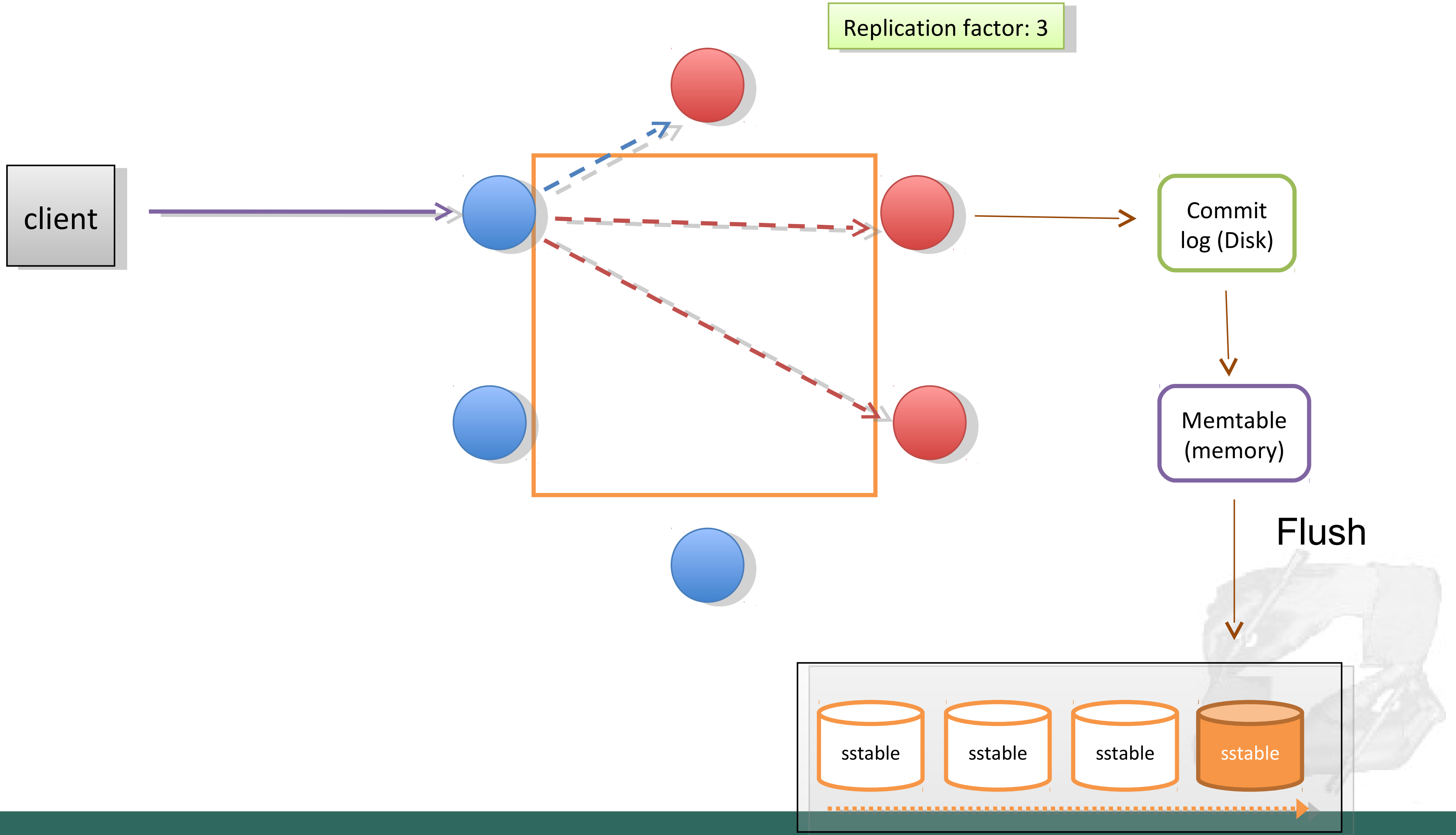


http://planetcassandra.org/nosql-performance-benchmarks/

Reading from Cassandra



Writing to Cassandra



CQL

```
CREATE TYPE address (
    street text,
    city text,
    zip int
);

CREATE TABLE user_profiles (
    login text PRIMARY KEY,
    first_name text,
    last_name text,
    email text,
    addresses map<text, address>
);

// Inserts a user with a home address
INSERT INTO user_profiles(login, first_name, last_name, email, addresses)
VALUES ('tsmith',
        'Tom',
        'Smith',
        'tsmith@gmail.com',
        { 'home': { street: '1021 West 4th St. #202',
                    city: 'San Fransisco',
                    zip: 94110 }});

// Adds a work address for our user
UPDATE user_profiles
   SET addresses = addresses
                + { 'work': { street: '3975 Freedom Circle Blvd',
                              city: 'Santa Clara',
                              zip: 95050 }}
 WHERE login = 'tsmith';
```

HBase: TwitBase

- TwitBase Domain Model:
  - User(user: String, name: String, email: String, password: String, twitsCount: Int)
  - Twit(user: String, datetime: DateTime, text: String)
  - Relation(from: String, relation: String, to: String)
- Design Steps:
  - Primary Key definition
  - Data shape and access patterns definition
  - Logical model definition (Physical Model)

HBase TwitBase: User

User(user: String, name: String, email: String, password: String)

- We can define the table users

  create ’users’, ‘info’
- Primary Key -> we need an unique identifier
- Operations

–add a new user -> put(key, columnFamily, columnQualifier, value)

–retrieve a specific user -> get(key)

–list all the users -> scan on the table users setting the family

HBase TwitBase: Twit

Twit(user: String, datetime: DateTime, text: String)

- We can define the table Twits

  create ‘twits’ ‘twits’

- Primary Key-> [md5(userId),Bytes.toByte(-1*timestamp)]
- Operations

–post a new twit on user’s behalf -> put

–list all the twits for the specified user -> scan on a partial key, that is from [md5(userId),8byte] to [md5(userId),8byte] +1 on the last byte

HBase TwitBase: Relation

Relation(from: String, relation: String, to: String)

- We can define the table Relation

  create ’follows’, ‘f’

  create ‘followedBy’, ‘f’

- Primary Key: [md5(fromId),md5(toId)]
- Operations

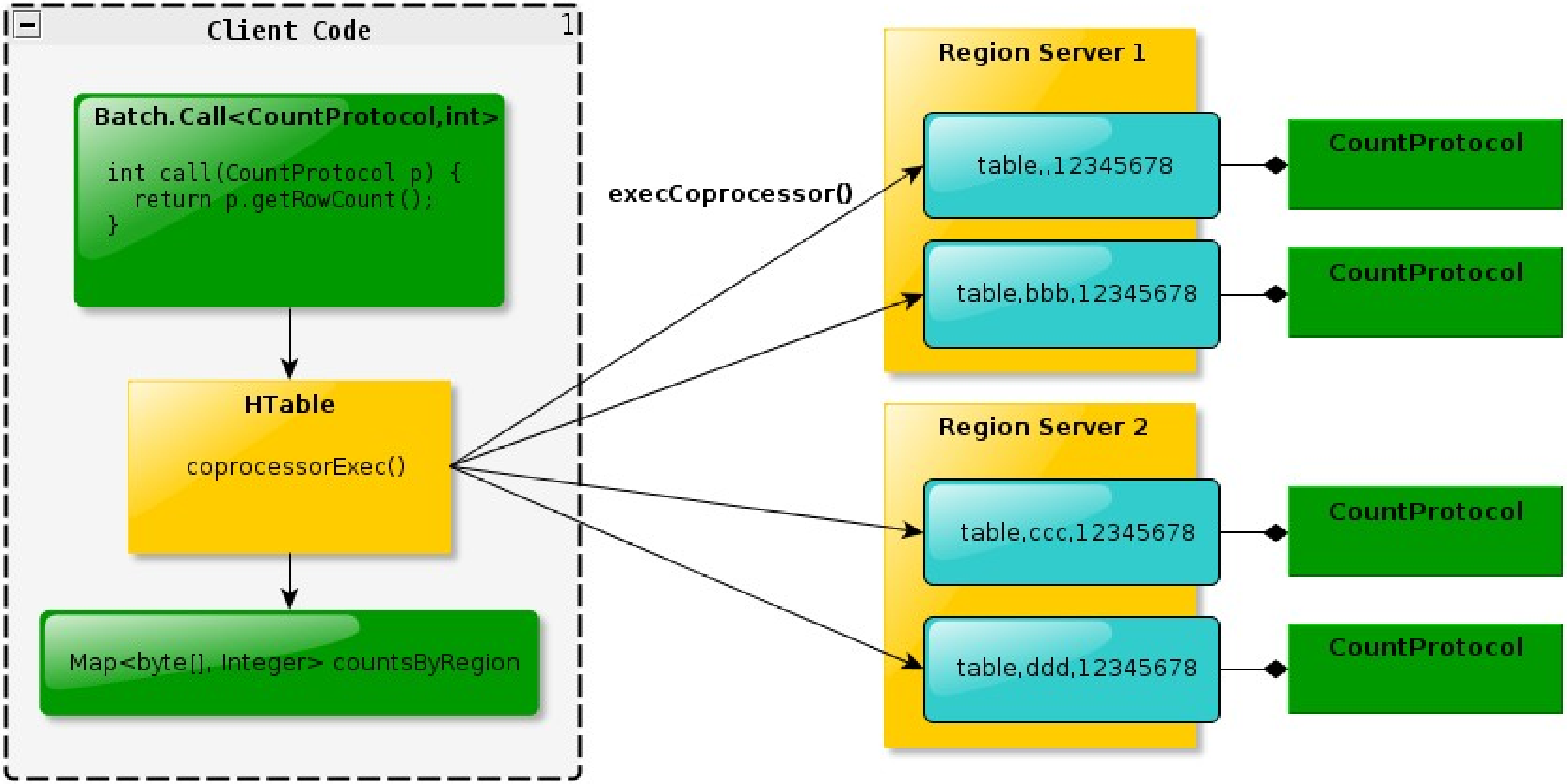
–add a new relation-> put

–list everyone user-Id follows -> Scan using the md5[userId]

–list everyone who follows user-Id -> Scan using the md5[userId]

–count users' followers -> any suggestion?

Coprocessors



https://blogs.apache.org/hbase/entry/coprocessor_introduction

Column Oriented Considerations

More efficient than row (or document) store if:

- Multiple row/record/documents are inserted at the same time so updates of column blocks can be aggregated
- Retrievals access only some of the columns in a row/record/document

Wait… online vs. offline operations

- We focused on online operations.
- Get and Put return result in milliseconds.
- The twits table’s rowkey is designed to maximize physical data locality and minimize the time spent scanning records.
- But not all the operations can be done online.
- What’s about offline operations (e.g site traffic summary report).
- This operations have performance concerns as well.

Scan vs Thread Scan vs MapReduce

- Scan is a serial operations
- We can used a thread pool to speedup the computation
- We can use MapReduce to split the work in Map and Reduce.

```
1. map(key: LongWritable ,value: Text ,context: Context)

2. reduce(key: Text, vals: Iterable[LongWritable],context:
   Context)
```

HBase MapReduce integration

```
TableMapReduceUtil.initTableMapperJob
( "twits", scan, Map.class,
ImmutableBytesWritable.class,
Result.class,
job);
```

Document Store

Document Store

- Schema Free.
- Usually JSON (BSON) like interchange model, which supports lists, maps, dates, Boolean with nesting
- Query Model: JavaScript or custom.
- Aggregations: Map/Reduce.
- Indexes are done via B-Trees.
- Example: Mongo
  - {Name:"Jaroslav",

    Address:"Malostranske nám. 25, 118 00 Praha 1“

    Grandchildren: [Claire: "7", Barbara: "6", "Magda: "3", "Kirsten: "1", "Otis: "3", Richard: "1"]

    }

Document Store: Advantages

- Documents are independent units
- Application logic is easier to write. (JSON).
- Schema Free:
  - Unstructured data can be stored easily, since a document contains whatever keys and values the application logic requires.
  - In addition, costly migrations are avoided since the database does not need to know its information schema in advance.

Document Store

- MongoDB
- CouchDB
- CouchBase
- RethinkDB

MongoDB

- Consistency and Partition Tolerance
- MongoDB's documents are encoded in a JSON-like format (BSON) which
  - makes storage easy, is a natural fit for modern object-oriented programming methodologies,
  - and is also lightweight, fast and traversable.
- It supports rich queries and full indexes.
  - Queries are javascript expressions.
  - Each object stored as an object Id.
- Has geospatial indexing.
- Supports Map-Reduce queries.

MongoDB: Features

- Replication Methods: replica set and master slave
- Read Performance - Mongo employs a custom binary protocol (and format) providing at least a magnitude times faster reads than CouchDB at the moment.
- Provides speed-oriented operations like upserts and update-in-place mechanics in the database.

CouchDB

- Written in Erlang.
- Documents are stored using JSON.
- The query language is in Javascript and supports Map-Reduce integration.
- One of its distinguishing features is multi-master replication.
- ACID: It implements a form of Multi-Version Concurrency Control (MVCC) in order to avoid the need to lock the database file during writes.
- CouchDB guarantees eventual consistency to be able to provide both availability and partition tolerance.

CouchDB: Features

- Master-Master Replication - Because of the append-only style of commits.
- Reliability of the actual data store backing the DB (Log Files)
- Mobile platform support. CouchDB actually has installs for iOS and Android.
- HTTP REST JSON interaction only. No binary protocol

CouchDB JSON Example

```
{
  "_id": "guid goes here",
  "_rev": "314159",

  "type": "abstract",

  "author": "Keith W. Hare"

  "title": "SQL Standard and NoSQL Databases",

  "body": "NoSQL databases (either no-SQL or Not Only SQL)
           are currently a hot topic in some parts of
           computing.",
  "creation_timestamp": "2011/05/10 13:30:00 +0004"
}
```

CouchBase

NoSQL FEATURES

SCALABILITY
engineered to scale, with ease.
DETAILS

+

PERFORMANCE
engineered to perform, consistently.
DETAILS

+

AVAILABILITY
engineered for high availability; always on.
DETAILS

NoSQL SOLUTIONS

CACHE
a distributed cache for high performance.
USE CASE

+

KEY / VALUE
a key / value store for performance & durability.
USE CASE

+

DOCUMENT
a document database for durability & processing.
USE CASE

Couchbase: Subscriptions

DTK

2.5.1 – latest

Operating System	Enterprise Edition Recommended for development and production [Why Enterprise?]	Community Edition Courtesy builds for enthusiasts [Why Community?]
64-bit Ubuntu 12.04	2.5.1 Release | [md5]	2.2.0 Release | [md5]
32-bit Ubuntu 12.04	2.5.1 Release | [md5]	2.2.0 Release | [md5]
Install instructions	Release Notes Manual	Release Notes Manual
64-bit Ubuntu 10.04	2.5.1 Release | [md5]	2.2.0 Release | [md5]
32-bit Ubuntu 10.04	2.5.1 Release | [md5]	2.2.0 Release | [md5]
Install instructions	Release Notes Manual	Release Notes Manual

RethinkDB

- Document Store based on JSON
- It extends MongoDB allowing for:
  - Aggregations using grouped map reduce
  - Joins and full sub queries
  - Full javascript v8 functions
- RethinkDB supports primary key, compound, secondary, and arbitrarily computed indexes stored as B-trees.

MongoDB: TwitBase

- TwitBase Domain Model:
  - User(user: String, name: String, email: String, password: String, twitsCount: Int)
  - Twit(user: String, datetime: DateTime, text: String)
  - Relation(from: String, relation: String, to: String)
- Design Steps:
  - Primary Key definition
  - Data shape and access patterns definition
  - Logical model definition (Physical Model)

MongoDB TwitBase: User

User(user: String, name: String, email: String, password: String)

- We can define the collection users

  db.createCollection(“users”)
- Primary Key ->can we use the default OBjectId?
- Operations

–add a new user -> db.users.insert({…})

–retrieve a specific user -> db.user.find({…})

–list all the users -> db.users.findAll({})

MongoDB TwitBase: Twit

Twit(user: String, datetime: DateTime, text: String)

- We can define the collection Twits

  db.createCollection("twits")
- Primary Key-> ???
- Operations

–post a new twit on user's behalf -> db.twits.insert({})

–list all the twits for the specified user -> db.find({userId : pippo})

Data Model Considerations

- Put as much in as it is possible (subdocuments, avoid joins)
- Separate data that can be referred to from multiple sources
- Document size consideration (16Mb)
- Complex data structures (search issues, no subdocument return)
- Data Consistency, there aren't locks

Twits Data Model

- Embed Twits into Users collections and assign an id to each twit.
- The Object id has a timestamp embedded so we can use that

MongoDB TwitBase: Relation

Relation(from: String, relation: String, to: String)

- We can embed the collection relation in the users collection
- Operations

–add a new relation

–list everyone user-Id follows

–list everyone who follows user-Id

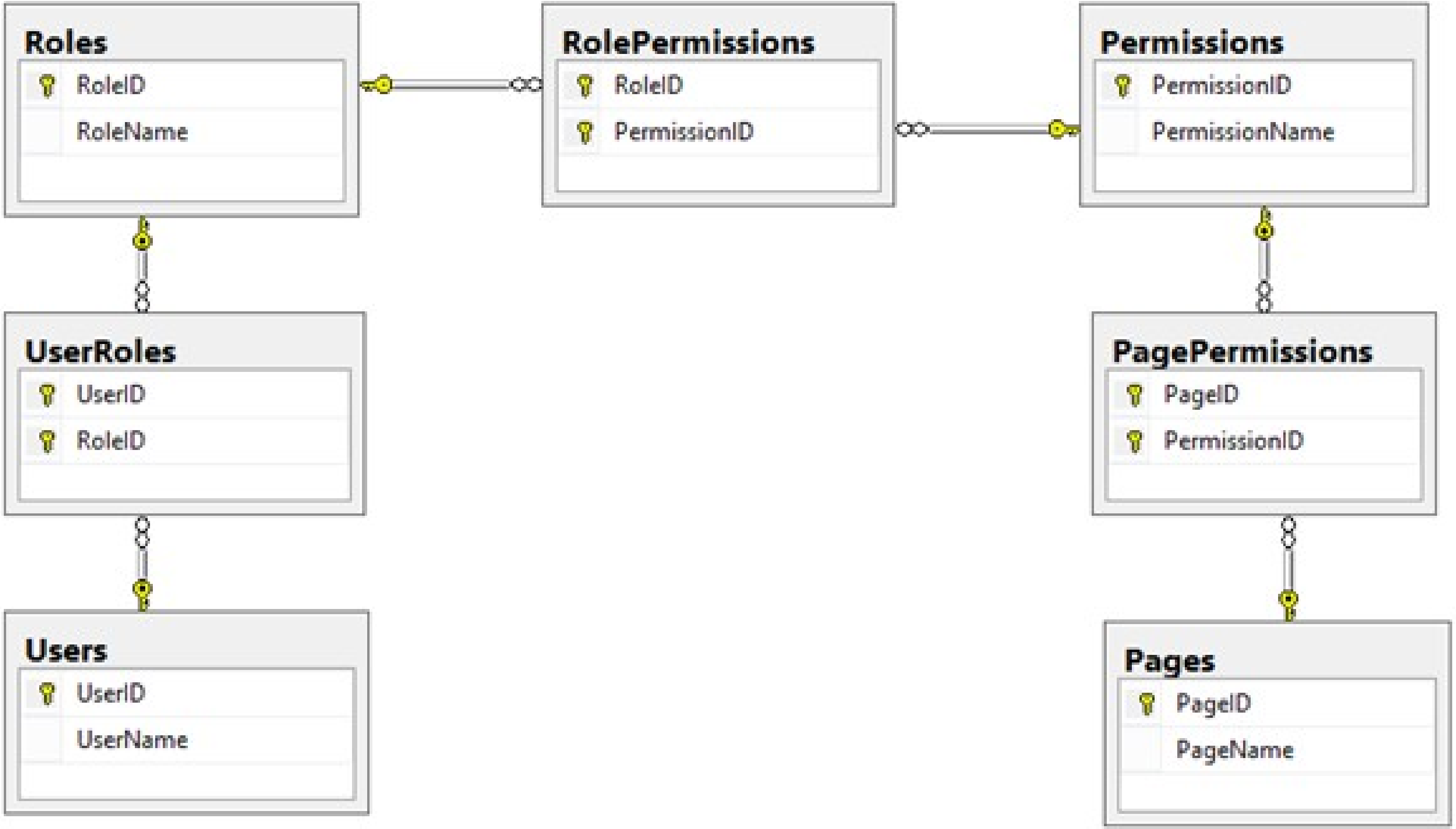
–count users' followers -> any suggestion? counter

Graph Databases

Graph Databases

- They are significantly different from the other three classes of NoSQL databases.
- Graph Databases are based on the mathematical concept of graph theory.
- They fit well in several real world applications (twits, permission models)
- Are based on the concepts of Vertex and Edges
- A Graph DB can be labeled, directed, attributed multi-graph
- Relational DBs can model graphs, but an edge does not require a join which is expensive.

Example: Role Permission



Example: Role Permission

```sql
SELECT  P.PageID, P.PageName
FROM    [PagePermissions] PP
        INNER JOIN [Pages] P ON PP.PageID = P.PageID
        INNER JOIN [Permissions] PE ON PE.PermissionID = PP.PermissionID
        INNER JOIN [RolePermissions] RP ON PE.PermissionID = RP.PermissionID
        INNER JOIN [Roles] R ON RP.RoleID = R.RoleID
        INNER JOIN [UserRoles] UR ON R.RoleID = UR.RoleID
        INNER JOIN [Users] U ON UR.UserID = U.UserID
WHERE   U.UserName = 'JohnDoe'
AND PE.PermissionName = 'VIEW'
```

- The real problem here is that we are trying to solve a Graph problem by using Sets (Relational Algebra)

Graph Store

- Neo4j
- Titan
- OrientDB

Neo4j

- A highly scalable open source graph database that supports ACID,
- has high-availability clustering for enterprise deployments,
- Neo4j provides a fully equipped, well designed and documented rest interface
- Neo4j is the most popular graph database in use today.
- License AGPLv3/Commercial

Neo4j: Features

- It includes extensive and extensible libraries for powerful graph operations such as traversals, shortest path determination, conversions, transformations.
- It includes triggers, which are referred to as transaction event handlers.
- Neo4j can be configured as a multi-node cluster.
- An embedded version of Neo4j is available, which runs directly in the application context.
- GIS Indexes (QuadTree, HierarchyTree, …)

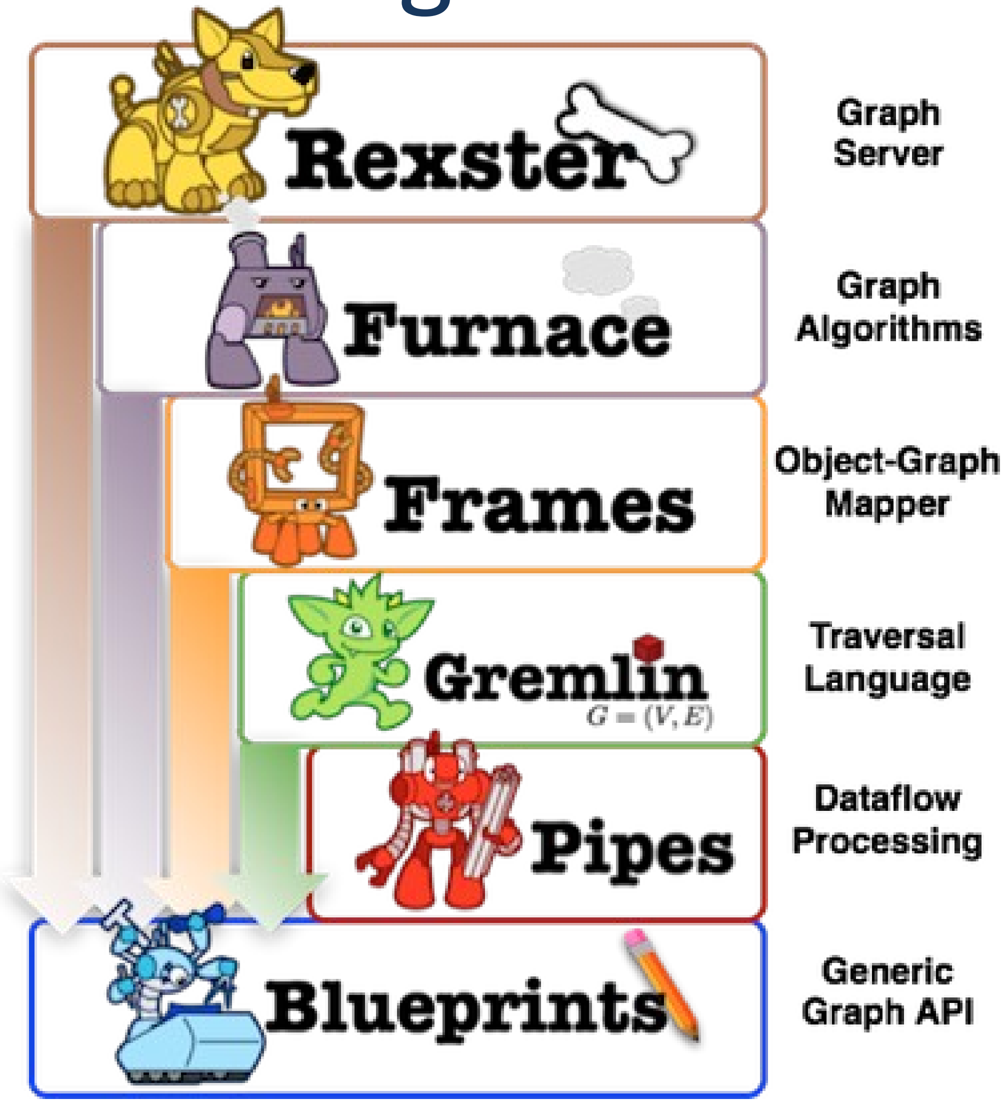
Titan

- Support for ACID and eventual consistency.
- Support for various storage backends: Apache Cassandra, Apache Hbase, Oracle BerkeleyDB, Akiban Persistit and Hazelcast.
- Support for geo, numeric range, and full-text search via: ElasticSearch, Apache Lucene
- Native integration with the TinkerPop graph stack
- Open source with the liberal Apache 2 license.
- Edge compression and vertex-centric indices

OrientDB

- Key-Value DB, Graph DB and Document DB
- SQL and Transactions
- Distributed: OrientDB supports Multi-Master Replication
- It supports different types of relations:
  - 1-1 and N-1 referenced relationships
  - 1-N and N-M referenced relationships
    - LINKLIST, as an ordered list of links
    - LINKSET, as an unordered set of links. It doesn't accepts duplicates
    - LINKMAP, as an ordered map of links with key a String. It doesn't accepts duplicated keys

All: Gremlin Integration



TitanDB: TwitBase

- TwitBase Domain Model:
  - User(user: String, name: String, email: String, password: String, twitsCount: Int)
  - Twit(user: String, datetime: DateTime, text: String)
  - Relation(from: String, relation: String, to: String)
- Design Steps:
  - Primary Key definition
  - Data shape and access patterns definition
  - Logical model definition (Physical Model)

TitanDB TwitBase: User

User(user: String, name: String, email: String, password: String)

- We can define a vertex for each user

```
g.createKeyIndex("name",Vertex.class);
Vertex pippo = g.addVertex(null);
juno.setProperty(”type", ”user");
juno.setProperty(”user", ”pippo");
```

- Primary key -> unique index on name property
- Operations

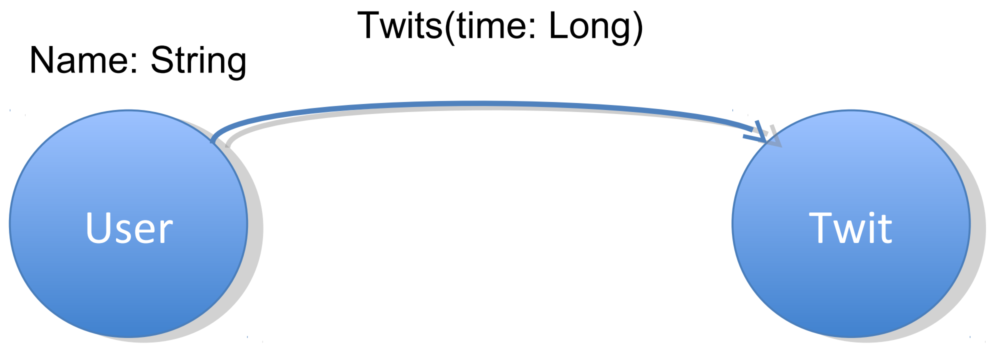
–add a new user -> g.addVertex()

–retrieve a specific user -> g.getVertices(“name”,”pippo”)

–list all the users -> g.getVertices(“type”,”user”)

TitanDB TwitBase: Twit

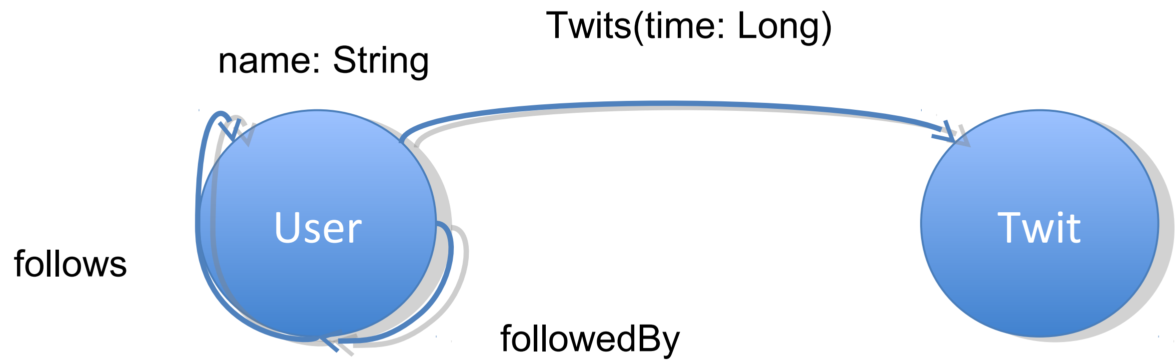
Twit(user: String, datetime: DateTime, text: String)



- Operations
  - post a new twit on user's behalf
    - Val twit = g.addVertex("twit"); val edge = g.addEdge(pippo,twit,"twit")
  - list all the twits for the specified user
    - Val results = pippo.query().labels("twit").vertices()

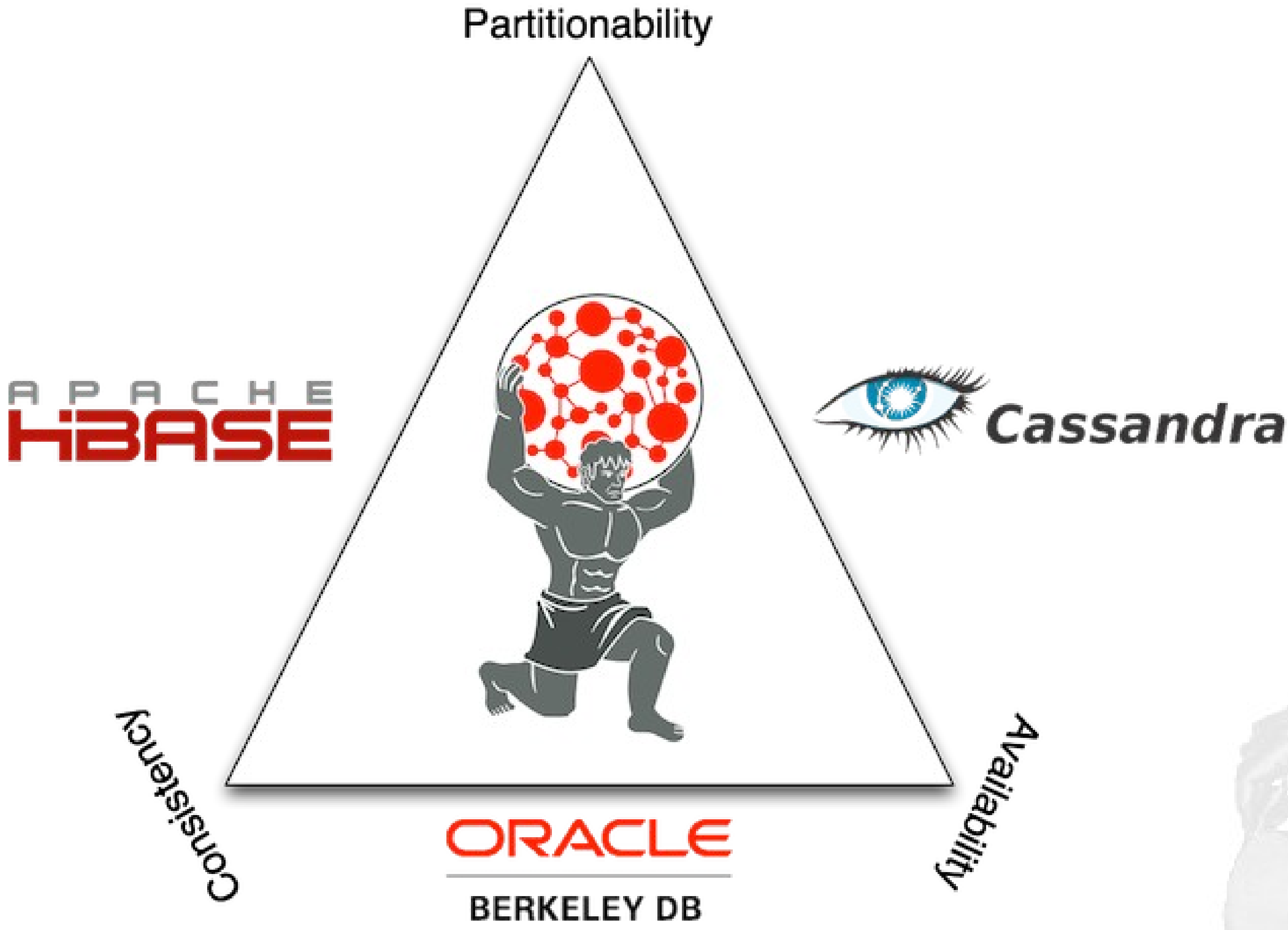
TitanDB TwitBase: Relation

Relation(from: String, relation: String, to: String)



- Operations
  - add a new relation -> val edge = g.addEdge(pippo,martin,”follows”)
  - list everyone user-Id follows -> pippo.query().labels(“follows”).vertices()
  - list everyone who follows user-Id -> pippo.query().labels(“followedBy”).vertices()
  - count users' followers -> pippo.query().labels(“followedBy”).count()

Storage Backend



Storage Model

- Adjacency list in one column family
- Row key = vertex id
- Each property and edge in one column
  - Denormalized, i.e. stored twice
- Direction and label/key as column prefix
- Index are maintained into a separate column family

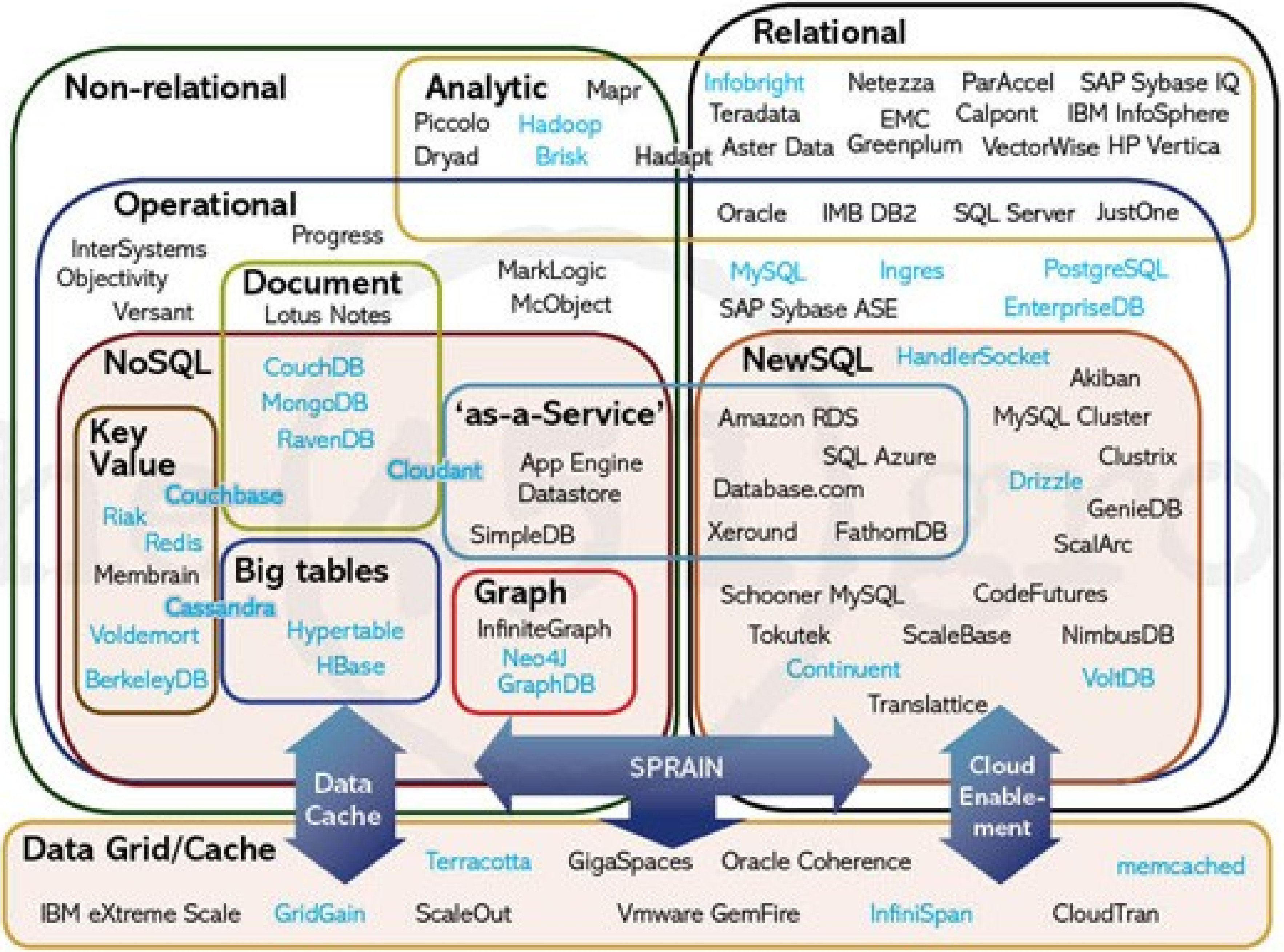
TwitBase: Stream and Recommendations

- Add Stream
  - pippo.in("follows").each{g.addEdge(it,tweet,"stream",['time':4])}
- Read Stream
  - Martin.outE(“stream”)[0..9].inV.map
- Followship Recommendation:

```
val follows = g.V('name',’Hercules’).out('follows').toList()!
val follows20 = follows[(0..19).collect{random.nextInt(follows.size)}]!
val m = [:]!
follows20.each
  { it.outE('follows’[0..29].inV.except(follows).groupCount(m).iterate() }
m.sort{a,b -> b.value <=> a.value}[0..4]
```

More!

Additional Informations

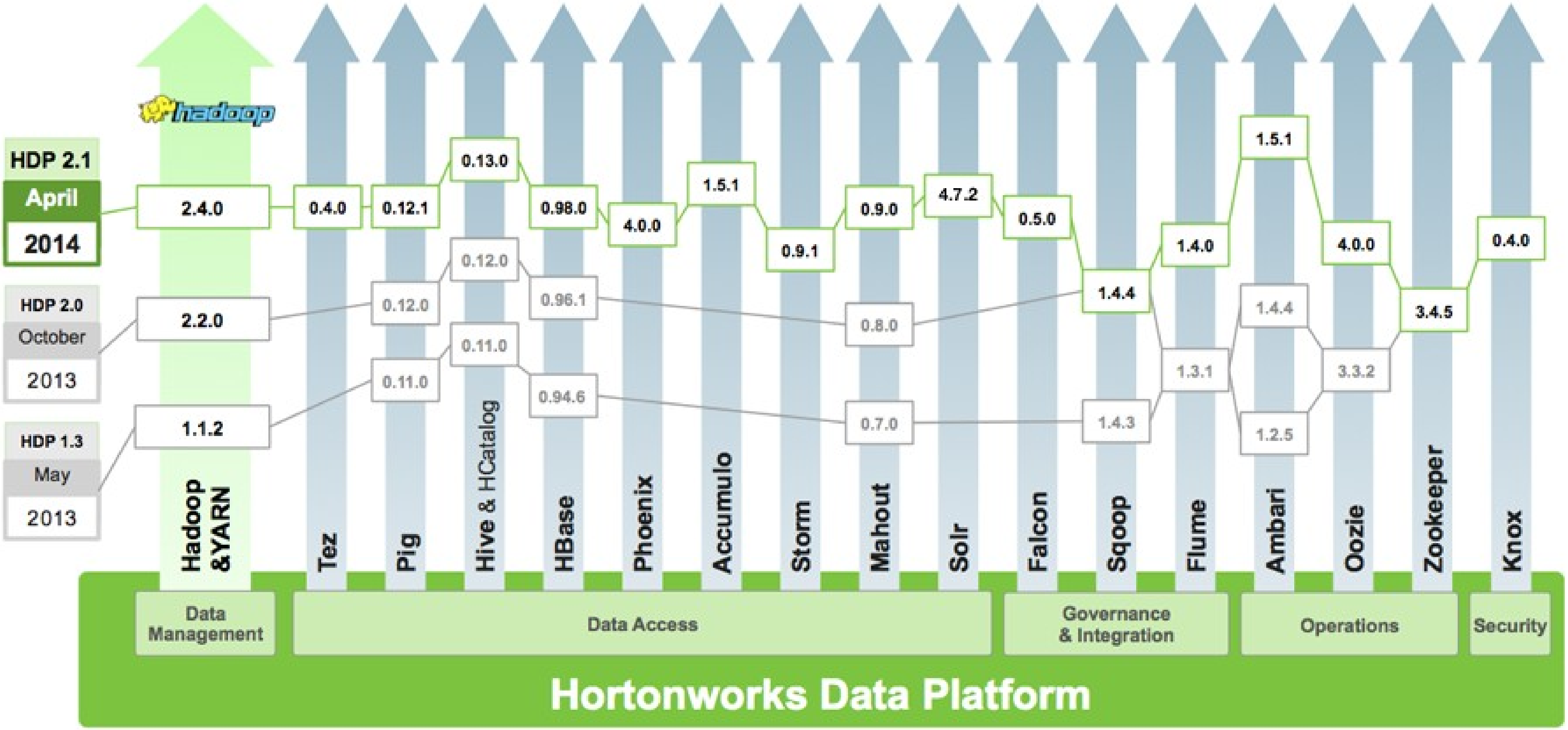

Relational
Non-relational
Analytic
Mapr
Piccolo
Hadoop
Dryad
Brisk
Hadapt
Infobright
Netezza
ParAccel
SAP Sybase IQ
Teradata
EMC
Calpont
IBM InfoSphere
Aster Data
Greenplum
VectorWise
HP Vertica
Operational
Oracle
IMB DB2
SQL Server
JustOne
Progress
InterSystems
Objectivity
Versant
Document
Lotus Notes
MarkLogic
McObject
MySQL
Ingres
PostgreSQL
SAP Sybase ASE
EnterpriseDB
NoSQL
CouchDB
MongoDB
RavenDB
Cloudant
Key Value
Couchbase
Riak
Redis
Membrain
Cassandra
Voldemort
BerkeleyDB
'as-a-Service'
App Engine Datastore
SimpleDB
NewSQL
HandlerSocket
Akiban
Amazon RDS
SQL Azure
Database.com
Xeround
FathomDB
MySQL Cluster
Clustrix
Drizzle
GenieDB
ScalArc
Big tables
Hypertable
HBase
Graph
InfiniteGraph
Neo4J
GraphDB
Schooner MySQL
CodeFutures
Tokutek
ScaleBase
NimbusDB
Continuent
VoltDB
Translattice
Data Cache
SPRAIN
Cloud Enable-ment
Data Grid/Cache
Terracotta
GigaSpaces
Oracle Coherence
memcached
IBM eXtreme Scale
GridGain
ScaleOut
Vmware GemFire
InfiniSpan
CloudTran

/

NewSQL

- Just like NoSQL it is more of a movement than specific product or even product family
- The “New” refers to the Vendors and not the SQL
- Goal(s):
  - Bring the benefits of relational model to distributed architectures, or,
    - VoltDB, ScaleDB, etc.
  - Improve Relational DB performance to no longer require horizontal scaling
    - Tokutek, ScaleBase, etc.
    - “SQL-as-a-service”: Amazon RDS, Microsoft SQL Azure, Google Cloud SQL

Hadoop



hadoop
HDP 2.1
April
2014
HDP 2.0
October
2013
HDP 1.3
May
2013
2.4.0
2.2.0
1.1.2
0.4.0
0.12.1
0.12.0
0.11.0
0.13.0
0.12.0
0.11.0
0.98.0
0.96.1
0.94.6
4.0.0
1.5.1
0.9.1
0.9.0
0.8.0
0.7.0
4.7.2
0.5.0
1.4.4
1.4.3
1.4.0
1.3.1
1.5.1
1.4.4
1.2.5
4.0.0
3.3.2
3.4.5
0.4.0
Hadoop &YARN
Tez
Pig
Hive & HCatalog
HBase
Phoenix
Accumulo
Storm
Mahout
Solr
Falcon
Sqoop
Flume
Ambari
Oozie
Zookeeper
Knox
Data Management
Data Access
Governance & Integration
Operations
Security
Hortonworks Data Platform

Spark

- Apache Spark is an open source cluster computing system that aims to make data analytics fast — both fast to run and fast to write.
- To run programs faster, Spark offers a general execution model that can optimize arbitrary operator graphs, and supports in-memory computing, which lets it query data faster than disk-based engines like Hadoop.
- Written in Scala using Akka.io

Spark examples

Word Count

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Logistic regression

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}
println("Final separating plane: " + w)
```

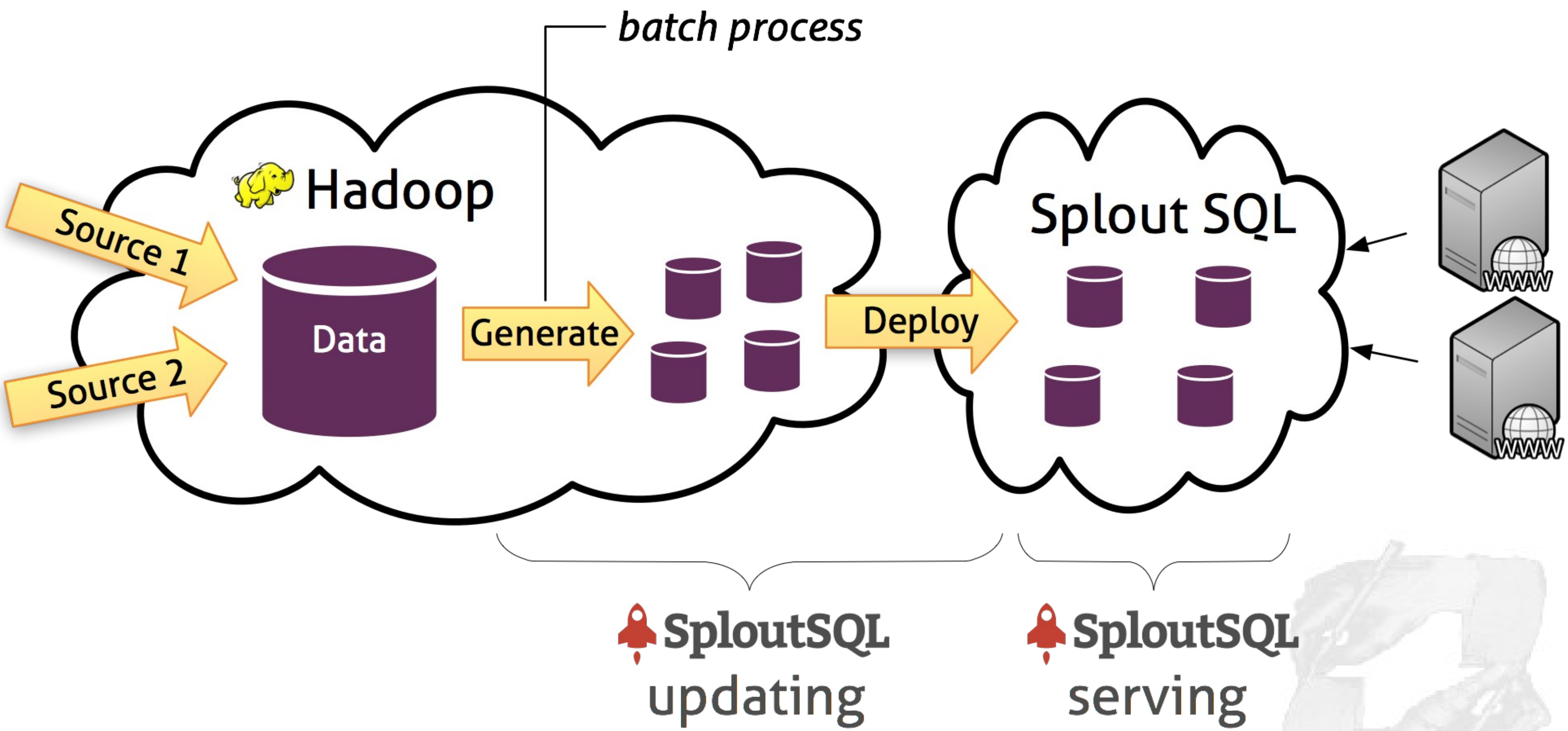
Spark

- Machine Learning Library (MLlib)
  - binary classification, regression, clustering and collaborative filtering, as well as an underlying gradient descent optimization primitive
- Bagel is a Spark implementation of Google’s Pregel graph processing framework
  - jobs run as a sequence of iterations called supersteps. In each superstep, each vertex in the graph runs a user-specified function that can update state associated with the vertex and send messages to other vertices for use in the next iteration.

Shark

- Shark is a large-scale data warehouse system for Spark designed to be compatible with Apache Hive. It can execute Hive QL queries up to 100 times faster than Hive without any modification to the existing data or queries.
  - REATE TABLE logs_last_month_cached AS SELECT * FROM logs WHERE time > date(...);
  - SELECT page, count(*) c FROM logs_last_month_cached GROUP BY page ORDER BY c DESC LIMIT 10;

Splout SQL



batch process
Hadoop
Source 1
Source 2
Data
Generate
Deploy
Splout SQL
WWW
WWW
SploutSQL
updating
SploutSQL
serving

Author: Swami Chandrasekaran
1. Fundamentals
Matrices & Linear Algebra Fundamentals
Hash Functions, Binary Tree, O(n)
Relational Algebra, DB Basics
Inner, Outer, Cross, Theta Join
CAP Theorem
Tabular Data
Entropy
Data Frames & Series
Sharding
OLAP
Multidimensional Data Model
ETL
Reporting Vs BI Vs Analytics
JSON & XML
NoSQL
Regex
Vendor Landscape
Env Setup
5%
2. Statistics
Pick a Dataset (UCI Repo)
Descriptive Statistics (mean, median, range, SD, Var)
Exploratory Data Analysis
Histograms
Percentiles & Outliers
Probability Theory
Bayes Theorem
Random Variables
Cumul Dist Fn (CDF)
Continuos Distributions (Normal, Poisson, Gaussian)
Skewness
ANOVA
Prob Den Fn (PDF)
Central Limit Theorem
Monte Carlo Method
Hypothesis Testing
p-Value
Chi² Test
Estimation
Confid Int (CI)
MLE
Kernel Density Estimate
Regression
Covariance
Correlation
Pearson Coeff
Causation
Least² Fit
Euclidean Distance
3. Programming
Python Basics
Working in Excel
R Setup R Studio
R Basics
Expressions
Variables
IBM SPSS
Rapid Miner
Vectors
Matrices
Arrays
Factors
Lists
Data Frames
Reading CSV Data
Reading Raw Data
Subsetting Data
Manipulate Data Frames
Functions
Factor Analysis
Install Pkgs
15%
4. Machine Learning
What is ML?
Numerical Var
Categorical Var
Supervised Learning
Unsupervised Learning
Concepts, Inputs & Attributes
Training & Test Data
Classifier
Prediction
Lift
Overfitting
Bias & Variance
Trees & Classification
Classification Rate
Decision Trees
Boosting
Naïve Bayes Classifiers
K-Nearest Neighbor
Logistic Regression
Ranking
Linear Regression
Perceptron
Regression
Classification
Hierarchical Clustering
K-means Clustering
Clustering
Neural Networks
Sentiment Analysis
Collaborative Filtering
Tagging
Vocabulary Mapping
Classify Text
50%
30%
5. Text Mining / NLP
Corpus
Named Entity Recognition
Text Analysis
UIMA
Term Document Matrix
Term Frequency & Weight
Support Vector Machines
Association Rules
Market Based Analysis
Feature Extraction
Using Mahout
Using Weka
Using NLTK
6. Visualization
Data Exploration in R (Hist, Boxplot etc)
Uni, Bi & Multivariate Viz
ggplot2
Histogram & Pie (Uni)
Tree & Tree Map
Scatter Plot (Bi)
Line Charts (Bi)
Spatial Charts
Survey Plot
Timeline
Decision Tree
D3.js
InfoVis
IBM ManyEyes
Tableau
40%
7. Big Data
Map Reduce Fundamentals
Hadoop Components
HDFS
Data Replication Principles
Setup Hadoop (IBM / Cloudera / HortonWorks)
Name & Data Nodes
Job & Task Tracker
M/R Programming
Sqoop: Loading Data in HDFS
Flume, Scribe: For Unstruct Data
SQL with Pig
DWH with Hive
Scribe, Chukwa For Weblog
Using Mahout
Zookeeper Avro
Storm: Hadoop Realtime
Rhadoop, RHIPE
rmr
Cassandra
MongoDB, Neo4j
60%
8. Data Ingestion
Summary of Data Formats
Data Discovery
Data Sources & Acquisition
Data Integration
Data Fusion
Transformation & Enrichment
Data Survey
Google OpenRefine
How much Data?
Using ETL
80%
9. Data Munging
Dimensionality & Numerosity Reduction
Normalization
Data Scrubbing
Handling Missing Values
Unbiased Estimators
Binning Sparse Values
Feature Extraction
Denoising
Sampling
Stratified Sampling
Principal Component Analysis
10. Toolbox
MS Excel w/ Analysis ToolPak
Java, Python
R, R-Studio, Rattle
Weka, Knime, RapidMiner
Hadoop Dist of Choice
Spark, Storm
Flume, Scibe, Chukwa
Nutch, Talend, Scraperwiki
Webscraper, Flume, Sqoop
tm, RWeka, NLTK
RHIPE
D3.js, ggplot2, Shiny
IBM Languageware
Cassandra, MongoDB
100%

But before CAP Theorem

NoSQL: How to

Brewer's CAP Theorem

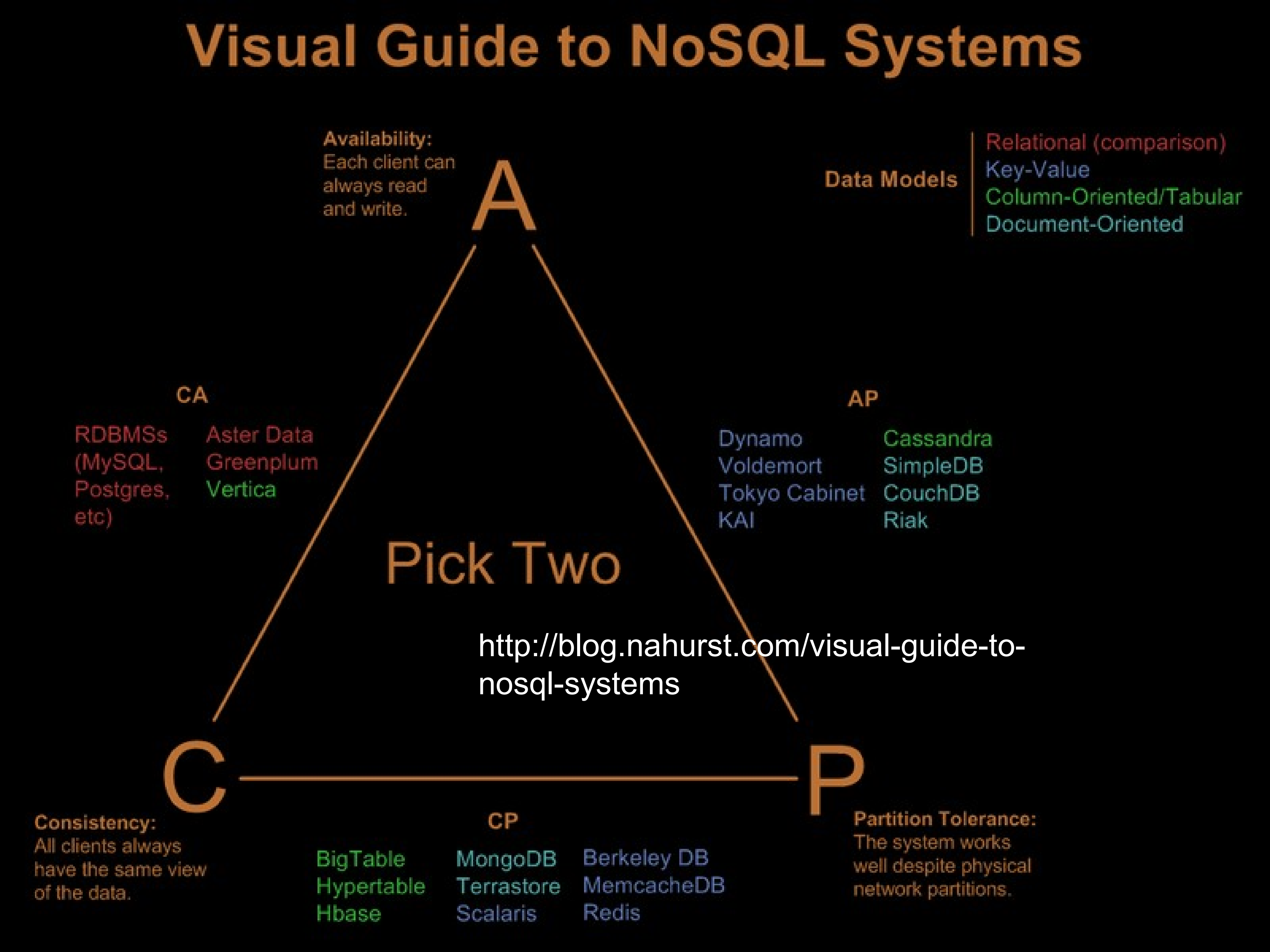
A distributed system can support only two of the following characteristics:

- **C**onsistency (all copies have same value)
- **A**vailability (system can run even if parts have failed)
- **P**artition Tolerance (network can break into two or more parts, each with active systems that can not influence other parts)

Brewer’s CAP Theorem

Very large systems will partition at some point:

- it is necessary to decide between Consistency and Availability,
- traditional DBMS prefer Consistency over Availability and Partition,
- most Web applications choose Availability (except in specific applications such as order processing)



http://blog.nahurst.com/visual-guide-to-nosql-systems