Like most new and upcoming technologies, NoSQL is shrouded in a mist of fear, uncertainty, and doubt.

Three Groups

Those who love it — People in this group are exploring how NoSQL fi ts in an application stack. They are using it, creating it, and keeping abreast with the developments in the world of NoSQL.

Those who deny it — Members of this group are either focusing on NoSQL's shortcomings or are out to prove that it's worthless.

Those who ignore it — Developers in this group are agnostic either because they are waiting for the **technology to mature**, or they **believe NoSQL is a passing fad** and ignoring it will shield them from the rollercoaster ride of "a hype cycle," or have simply not had a chance to get to it.

DEFINITION

- NoSQL is literally a combination of two words: No and SQL.
- The implication is that NoSQL is a technology or product that counters SQL.
- The creators and early adopters of the buzzword *NoSQL* probably wanted to say *No RDBMS* but were infatuated by the nicer sounding NoSQL and stuck to it.
- Few others have tried to salvage the original term by proposing that NoSQL is actually an acronym that expands to “Not Only SQL.”
- Whatever the literal meaning, NoSQL is used today as an umbrella term for all databases and data stores that **don’t follow the popular and well established RDBMS principles** and often relate to large data sets **accessed and manipulated on a Web scale**.
- This means NoSQL is not a single product or even a single technology.
- It represents a class of products and a collection of diverse, and sometimes related, concepts about data storage and manipulation.

CHALLENGES OF RDBMS

- RDBMS assumes a **well defined structure** in data.
- It assumes that the **data is dense and is largely uniform**.
- **RDBMS builds on a prerequisite that the** properties of the data can be defined up front and that its **interrelationships are well established** and systematically referenced.
- It also assumes **that indexes can be consistently defined** on data sets and that such indexes can be **uniformly leveraged for faster querying.**
- Unfortunately, RDBMS starts to show signs of giving way as soon as these assumptions **don't hold true.**
- RDBMS can certainly deal with some irregularities and lack of structure but in the **context of massive sparse data sets with loosely defined structures, RDBMS appears a forced fit.**
- With **massive data** sets the typical **storage mechanisms and access methods also get stre**tched.
- Denormalizing tables, dropping constraints, and relaxing transactional guarantee can help an RDBMS scale, but after these modifi cations an RDBMS starts resembling a NoSQL product
- NoSQL alleviates the problems that RDBMS imposes and makes it easy to work with large sparse data, but in turn takes away the power of transactional integrity and flexible indexing and querying

History

- Non-relational databases are **not new**.
- Non-relational databases thrived through the advent of mainframes and **have existed in specialized and specific domains** — for example, hierarchical directories for storing authentication and authorization credentials.
- However, the **non-relational stores that have appeared in the world of NoSQL are a new incarnation**, which were born in the world of **massively scalable Internet applications.**
- With **Google**, it is clear that the widely adopted relational database management system (RDBMS) has its own **set of problems when applied to massive amounts of data**.
- The problems relate to efficient processing, effective parallelization, scalability, and costs.
- Google has, over the past few years, **built out a massively scalable infrastructure for its search engine** and other applications, including Google Maps, Google Earth, GMail, Google Finance, and Google Apps.
- Google's approach was **to solve the problem at every level of the application stack**.
- The goal was to build a scalable infrastructure for parallel processing of large amounts of data.
- Google therefore created a **full mechanism** that included a distributed filesystem, a column-family-oriented data store, a distributed coordination system, and a MapReduce-based parallel algorithm execution environment.

- The release of **Google's papers to the public spurred a lot of interest among open-source Developers**. This open-source alternative is Hadoop, its sub-projects, and its related projects.

- Later, **Amazon** decided to share some of its own success story.

- Amazon presented its ideas of a distributed highly available and eventually consistent data store named Dynamo

- With **endorsement of NoSQL** from two leading web giants — Google and Amazon — several new products emerged in this space.

- A lot of developers **started toying with the idea of using these methods** in their applications and many enterprises, from startups to large corporations, **became amenable to learning more** about the technology and possibly using these methods.

- In **less than 5 years**, NoSQL and related concepts for **managing big data have become widespread** and use cases have emerged from many well-known companies, including Facebook, Netfl ix, Yahoo, EBay, Hulu, IBM, and many more

Challenges and solutions around large data and parallel processing.

Big Data

Just how much data qualifies as big data? This is typically the **size where the data set is large enough to start spanning multiple storage units**. It's also the **size at which traditional RDBMS techniques start showing the first signs of stress**.

DATA SIZE MATH

A byte is a unit of digital information that consists of 8 bits.

In the International System of Units (SI) scheme every 1,000 (10 power 3) multiple of a byte is given a distinct name, which is as follows:

Kilobyte (kB) — 10 power 3
Megabyte (MB) — 10 power 6
Gigabyte (GB) — 10 power 9
Terabyte (TB) — 10 power 12
Petabyte (PB) — 10 power 15
Exabyte (EB) — 10 power 18
Zettabyte (ZB) — 10 power 21
Yottabyte (YB) — 10 power 24

As the **size of data grows** and sources **of data creation become increasingly diverse**, the following **growing challenges will get further amplified**:

- **Efficiently storing and accessing large** amounts of data **is difficult**.
- The additional demands of **fault tolerance and backups** makes things even more **complicated**.
- **Manipulating large data sets** involves running immensely **parallel processes**.
- **Gracefully recovering from any failures** during such a run and **providing results** in **a reasonably short** period of time is complex.
- Managing the **continuously evolving schema and metadata** for semi-structured and un-structured data, generated by diverse sources, is a **convoluted problem**

Note: Therefore, the ways and means **of storing and retrieving large amounts of** data need **newer approaches** beyond our current methods.

NoSQL and related big-data solutions are **a first step** forward in that direction

Scalability

- Scalability is the **ability of a system to increase throughput** with **addition of resources to address load increases.**
- Scalability can be **achieved either by provisioning a large and powerful resource** to meet the **additional demands or it can be achieved by relying on a cluster of ordinary machines to work as a unit.**
- The involvement of **large, powerful machines is typically classified as vertical scalability**.
- Provisioning **super computers with many CPU cores** and large amounts of directly attached storage is a typical **vertical scaling solution**.
- Such vertical scaling options are typically **expensive and proprietary**. The **alternative** to vertical scalability is **horizontal scalability.**
- Horizontal scalability involves a **cluster of commodity systems** where the cluster **scales as load increases**.
- Horizontal scalability typically **involves adding additional nodes** to serve additional load.
- Processing data spread across a cluster of **horizontally scaled machines is complex**.
- The **MapReduce model possibly provides one of the best possible methods** to process large-scale data on a **horizontal cluster of machines**.

Definition and Introduction of Mapredcue

- MapReduce is a **parallel programming model** that allows **distributed processing on large data sets** on a **cluster of computers**
- MapReduce derives its ideas and **inspiration** from concepts in the world of **functional programming**
- In functional programming, a **map function applies an operation or a function to each element in a list**
  Eg Multiply-by-two function on a list [1, 2, 3, 4] would generate another list [2, 4, 6, 8].
- Like the map function, **functional programming** has a **concept of a reduce function** commonly known as a **fold function**
- Reduce or fold **function applies a function on all elements of a data structure**, such as a list, and **produces a single result or ou**
- So applying a reduce **function-like summation** on the list generated out of the map function, that is, [2, 4, 6, 8], would generate an output equal to 20.
- This same simple idea of map and reduce has **been extended to work on large data sets.**
- The idea **is slightly modified to work on collections of tuples or key/value pairs.**
- The map function applies a function on every key/value pair in the collection and **generates a new collection.**
- Then the reduce function works on the new generated collection and applies an aggregate function to compute a final output.

Say you have a collection of key/value pairs as follows:
[{ “94303”: “Tom”}, {“94303”: “Jane”}, {“94301”: “Arun”}, {“94302”: “Chen”}]
This simple map function on this collection could get the
names of all those who reside in a particular zip code
[{“94303”:[“Tom”, “Jane”]}, {“94301”:[“Arun”]}, {“94302”:[“Chen”]}]

Now a reduce function could work on this output to simply count the number of people who belong to particular zip code. The fi nal output then would be as follows:
[{“94303”: 2}, {“94301”: 1}, {“94302”: 1}]

SORTED ORDERED COLUMN-ORIENTED STORES

Google's Bigtable espouses a model where data in stored in a column-oriented way.

This contrasts with the row-oriented format in RDBMS.

The column-oriented storage allows data to be stored effectively.

It avoids consuming space when storing nulls by simply not storing a column when a value doesn't exist for that column.

Each unit of data can be thought of as a set of key/value pairs, where the unit itself is identified with the help of a primary identifier, often referred to as the primary key.

Bigtable and its clones tend to call this primary key the row-key.

Also, units are stored in an ordered-sorted manner.

The units of data are sorted and ordered on the basis of the row-key.

Consider a simple table of values that keeps information about a set of people.

Such a table could have columns like first_name, last_name, occupation, zip_code, and gender. A person's information in this table could be as follows:

first_name: John
last_name: Doe
zip_code: 10001
gender: male

Another set of data in the same table could be as follows:
first_name: Jane
zip_code: 94303

The row-key of the first data point could be 1 and the second could be 2.

Then data would be stored in a sorted ordered column-oriented store in a way that the data point with row-key 1 will be stored before a data point with row-key 2 and also that the two data points will be adjacent to each other.

Next, only the valid key/value pairs would be stored for each data point.

So, a possible column-family for the example could be name with columns first_name and last_name being its members.

Another column-family could be location with zip_code as its member.

A third column-family could be profile.

The gender column could be a member of the profile column-family.

In column-oriented stores similar to Bigtable, data is stored on a column-family basis.

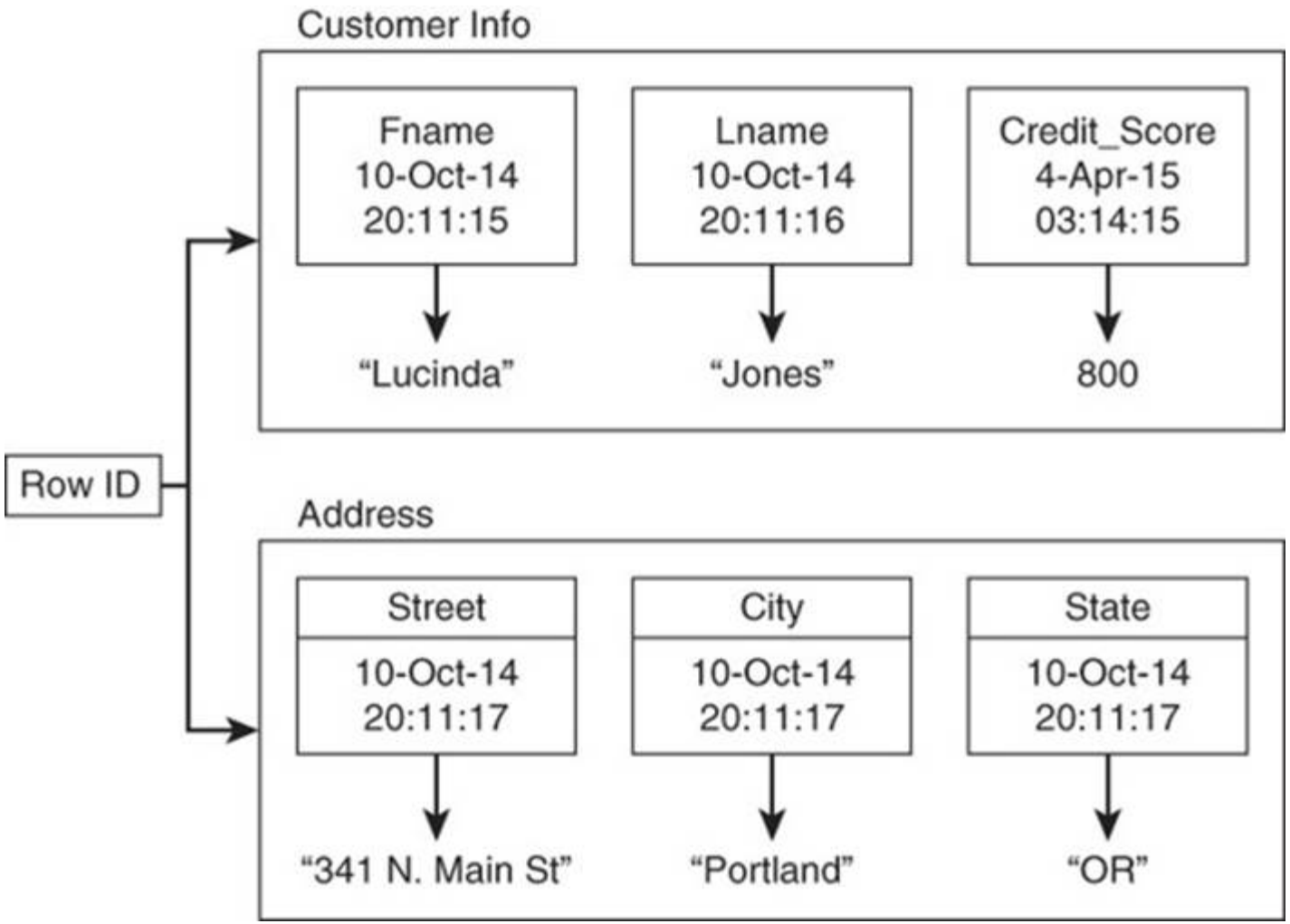
The underlying logical storage for this simple example consists of three storage buckets: name, location, and profile.

Within each bucket, only key/value pairs with valid values are stored.

Therefore, the name column-family bucket stores the following values:
For row-key: 1
first_name: John
last_name: Doe
For row-key: 2
first_name: Jane
The location column-family stores the following:
For row-key: 1
zip_code: 10001
For row-key: 2
zip_code: 94303
The profile column-family has values only for the data point with row-key 1 so it stores only
the following:
For row-key: 1
gender: male
In real storage terms, the column-families are not physically isolated for a given row.

All data pertaining to a row-key is stored together.

The column-family acts as a key for the columns it contains and the row-key acts as the key for the whole data set.
Example: Hbase, Hyper table

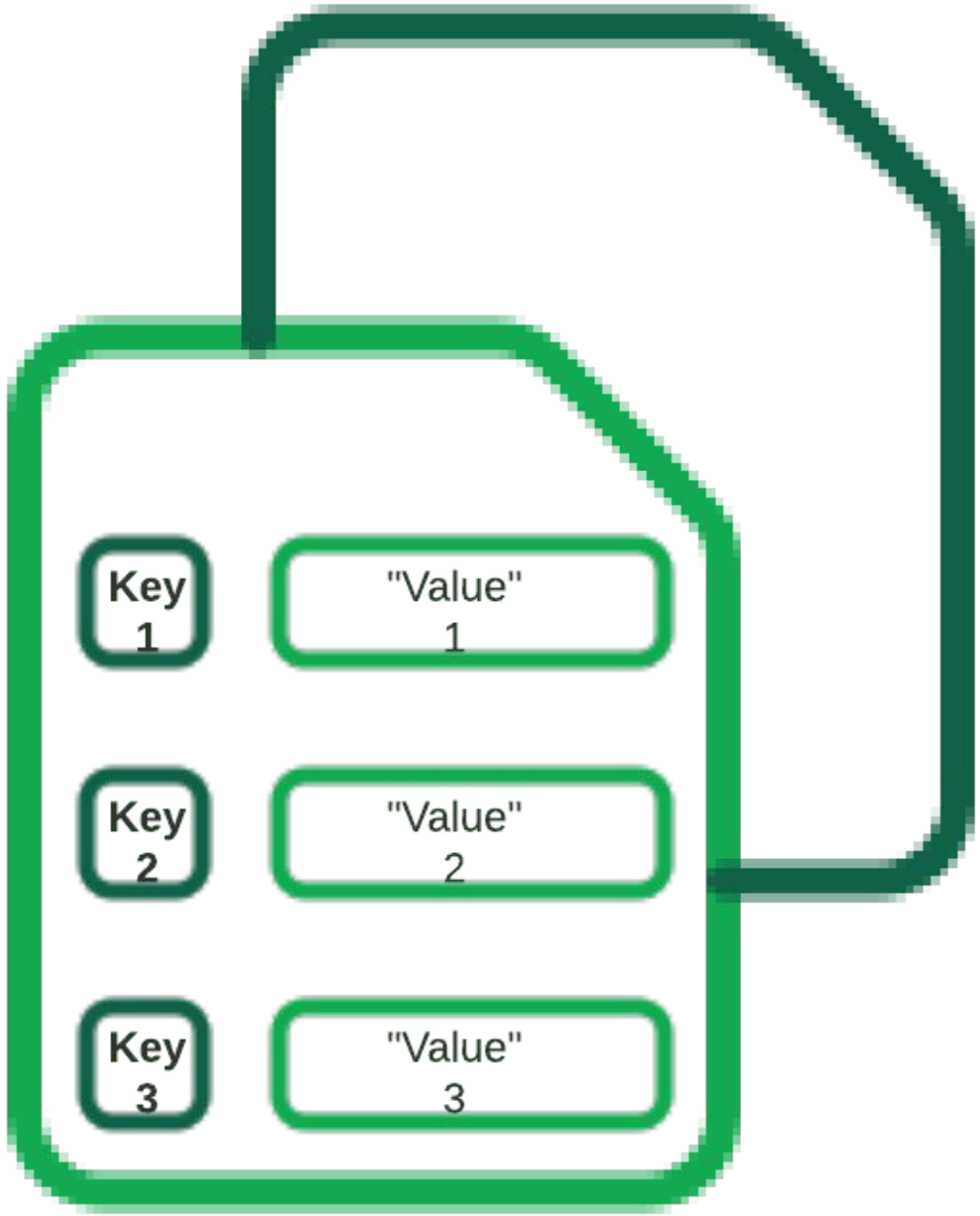

Customer Info
Fname
10-Oct-14
20:11:15
Lname
10-Oct-14
20:11:16
Credit_Score
4-Apr-15
03:14:15
"Lucinda"
"Jones"
800
Row ID
Address
Street
10-Oct-14
20:11:17
City
10-Oct-14
20:11:17
State
10-Oct-14
20:11:17
"341 N. Main St"
"Portland"
"OR"

KEY/VALUE STORES

The **key** of a key/value pair is a **unique value in the set and can be easily looked up to access the data.**

The **data is fetched by a unique key or a number of unique keys to retrieve the associated value with each key.**

The **values can be simple data types like strings and numbers or complex objects.**


Key 1
"Value" 1
Key 2
"Value" 2
Key 3
"Value" 3

key-value database, AKA key-value store, **associates a value (which can be anything from a number or simple string to a complex object) with a key, which is used to keep track of the object.**

A HashMap or an associative array is the simplest data structure that can hold a set of key/value pairs.

Such data structures are extremely popular because they provide a very effi cient, big O(1) average algorithm running time for accessing data.

What are the features of a key-value database?

A **key-value database is defined by the fact that it allows programs or users of programs to retrieve data by keys, which are essentially names, or identifiers, that point to some stored value.**

Common features

- Retrieving a value (if there is one) stored and associated with a given key
- Deleting the value (if there is one) stored and associated with a given key
- Setting, updating, and replacing the value (if there is one) associated with a given key

- Eg Membase, Berkely DB

DOCUMENT DATABASES

- Document databases are not document management systems.
- The word *document* in *document databases* connotes loosely structured sets of key/ value pairs in documents, typically JSON (JavaScript Object Notation), and not *documents* or *spreadsheets* (though these could be stored too).

- Document databases treat a document as a whole and avoid splitting a document into its constituent name/value pairs.
- At a collection level, this allows for putting together a diverse set of documents into a single collection.
- Document databases allow indexing of documents on the basis of not only its primary identifier but also its properties
- Eg- **MongoDB and CouchDB**

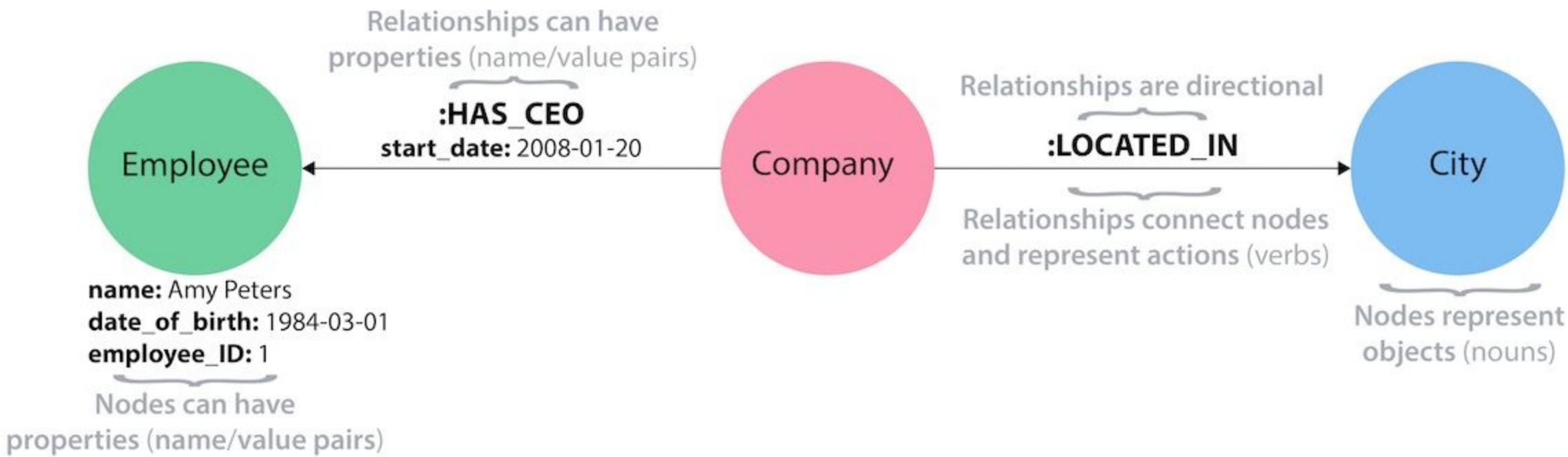
```
{_id: ObjectId("5effaa5662679b5af2c58829"),
 email: “email@example.com”,
 name: {given: “Jesse”, family: “Xiao”},
 age: 31,
 addresses: [{label: “home”,
              street: “101 Elm Street”,
              city: “Springfield”,
              state: “CA”,
              zip: “90000”,
              country: “US”},
             {label: “mom”,
              street: “555 Main Street”,
              city: “Jonestown”,
              province: “Ontario”,
              country: “CA”}]

}
```

GRAPH DATABASES

- A graph database is a database designed to treat the relationships between data as equally important to the data itself.
- It is intended to hold data without constricting it to a pre-defined model.
- Instead, the data is stored like we first draw it out - showing how each individual entity connects with or is related to others.

Why Graph Database

- We live in a connected world
- There are no isolated pieces of information, but rich, connected domains all around us.
- Only a database that natively embraces relationships is able to store, process, and query connections efficiently.
- While other databases compute relationships at query time through expensive JOIN operations, a graph database stores connections alongside the data in the model.
- Eg Neo4j and FlockDB


Relationships can have properties (name/value pairs)
:HAS_CEO
start_date: 2008-01-20
Employee
Company
Relationships are directional
:LOCATED_IN
City
Relationships connect nodes and represent actions (verbs)
Nodes represent objects (nouns)
name: Amy Peters
date_of_birth: 1984-03-01
employee_ID: 1
Nodes can have properties (name/value pairs)

Getting Initial Hands-on Experience

A Simple Set of Persistent Preferences Data

- Location-based services are gaining prominence as local businesses are trying to connect with users who are in the neighborhood and large companies are trying to customize their online experience and offerings based on where people are stationed.

- A few common occurrences of location-based preferences are visible in popular applications like Google Maps, which allows local search, and online retailers like Walmart.com that provide product availability and promotion information based on your closest Walmart store location.

- The location preferences are maintained by storing user identifier and a zip code .

- Data points like “John Doe, 10001,” “Lee Chang, 94129,” “Jenny Gonzalez 33101,” and “Srinivas Shastri, 02101” will need to be maintained

- To store such data in a flexible and extendible way, this example uses a non-relational database product named MongoDB.

Starting MongoDB and Storing Data

Assuming you have installed MongoDB successfully, start the server and connect to it.

You can start a MongoDB server by running the mongod program within the bin folder of the distribution.

When you start the MongoDB server by running mongod, you should see output on your console that looks similar to the following:

```
PS C:\applications\mongodb-win32-x86_64-1.8.1> .\bin\mongod.exe
C:\applications\mongodb-win32-x86_64-1.8.1\bin\mongod.exe
--help for help and startup options
Sun May 01 21:22:56 [initandlisten] MongoDB starting : pid=3300 port=27017
  dbpath=/data/db/ 64-bit
Sun May 01 21:22:56 [initandlisten] db version v1.8.1, pdfile version 4.5
Sun May 01 21:22:56 [initandlisten] git version:
a429cd4f535b2499cc4130b06ff7c26f41c00f04
Sun May 01 21:22:56 [initandlisten] build sys info: windows (6, 1, 7600, 2, '')
  BOOST_LIB_VERSION=1_42
Sun May 01 21:22:56 [initandlisten] waiting for connections on port 27017
Sun May 01 21:22:56 [websvr] web admin interface listening on port 28017
```

The simplest way to connect to the MongoDB server(Up and running) is to use the JavaScript shell/Mondobd shell available with the distribution.

- MongoDB **Mongo shell is an interactive JavaScript interface that allows you to interact with MongoDB instances** through the **command line**.
- The shell can be used for:

•Data manipulation

•Administrative operations such as maintenance of database instances

- MongoDB Mongo shell is the **default client for the MongoDB database server**.

- It's a **command-line interface (CLI), where the input and output are all console-based.**

- The Mongo shell is a **good tool to manipulate small sets of data.**

- Here are the top features that Mongo shell offers:

- Run all MongoDB queries from the Mongo shell.
- Manipulate data and perform administration operations.
- Mongo shell uses JavaScript and a related API to issue commands.
- See previous commands in the mongo shell with up and down arrow keys.
- View possible command completions using the tab button after partially entering a command.
- Print error messages, so you know what went wrong with your commands.

- Type **mongo** command to run the shell.

- By default, **MongoDB stores the data files in the /data/db (C:\data\db on Windows) directory and listens for requests on port 27017.**

- You can specify an **alternative data directory by specifying the directory path** using the dbpath option, as follows:

  mongod --dbpath /path/to/alternative/directory

- If your **MongoDB server runs on a different port**, you have to explicitly specify it in the command, as shown below:

  mongo –port 28010

  - By default, the mongo shell connects to the **"test" database** available on localhost which can be seen by typing command **db**

  - To explore a possible set of initial commands just type **help** on the mongo interactive console.

Run the **use** command to switch to a different database.

The commands to do the following are as follows

1. Switch to the prefs database.
2. Define the data sets that need to be stored.
3. Save the defined data sets in a collection, named location.

```
use prefs
w = {name: “John Doe”, zip: 10001};
x = {name: “Lee Chang”, zip: 94129};
y = {name: “Jenny Gonzalez”, zip: 33101};
z = {name: ”Srinivas Shastri”, zip: 02101};
db.location.save(w);
db.location.save(x);
db.location.save(y);
db.location.save(z);
```

Running db.location.find() reveals the following output:

```
> db.location.find()
{ “_id” : ObjectId(“4c97053abe6700000003857”), “name” : “John Doe”,
“zip” : 10001 }
{ “_id” : ObjectId(“4c970541be6700000003858”), “name” : “Lee Chang”,
“zip” : 94129 }
{ “_id” : ObjectId(“4c970548be6700000003859”), “name” : “Jenny Gonzalez”,
“zip” : 33101 }
{ “_id” : ObjectId(“4c970555be670000000385a”), “name” : “Srinivas Shastri”,
“zip” : 1089 }
```

ObjectId is MongoDB’s way of uniquely identifying each record or document in MongoDB terms.

Now add the following additional records to the location collection:
Don Joe, 10001
John Doe, 94129

You can accomplish this, via the mongo shell, as follows:

```
> a = {name:”Don Joe”, zip:10001};
{ “name” : “Don Joe”, “zip” : 10001 }
> b = {name:”John Doe”, zip:94129};
{ “name” : “John Doe”, “zip” : 94129 }
> db.location.save(a);
> db.location.save(b);
```

To get a list of only those people who are in the 10001 zip code, you could query as follows:

```
> db.location.find({zip: 10001});
{ “_id” : ObjectId(“4c97053abe67000000003857”), “name” : “John Doe”, “zip” : 10001 }
{ “_id” : ObjectId(“4c97a6555c760000000054d8”), “name” : “Don Joe”,
“zip” : 10001 }
```

To get a list of all those who have the name “John Doe,” you could query like so:

```
> db.location.find({name: “John Doe”});
{ “_id” : ObjectId(“4c97053abe67000000003857”), “name” : “John Doe”,
“zip” : 10001 }
{ “_id” : ObjectId(“4c97a7ef5c760000000054da”), “name” : “John Doe”,
“zip” : 94129 }
```

In both these queries that filter the collection, a query document is passed as a parameter to the find method.

The query document specifies the pattern of keys and values that need to be matched.

- Because a database includes **newer data sets**, it is possible the structure of the **collection will become a constraint and thus need modification.**

- **In traditional relational database** sense, you may need to **alter the table schema**.

- In relational databases, **altering table schemas** also means taking on a **complicated data migration** task to **make sure data in the old and the new schema exist together**.

- In **MongoDB**, modifying a **collection structure is trivial**. More accurately, **collections**, analogous to tables, are **schema-less** and so it allows you to **store disparate document types within the same collection.**

Consider an example where you need to store the location preferences of another user, whose name and zip code are identical to a document already existing in your database, say, another {name: “Lee Chang”, zip: 94129}.

To distinctly identify the second Lee Chang from the one in the database, an additional attribute, the street address, is added like so:

- anotherLee = {name:”Lee Chang”, zip: 94129, streetAddress:”37000 Graham Street”};

- db.location.save(anotherLee);

Now getting all documents, using find, returns the following data sets:

```
> db.location.find();
{ “_id” : ObjectId(“4c97053abe67000000003857”), “name” : “John Doe”,
“zip” : 10001 }
{ “_id” : ObjectId(“4c970541be67000000003858”), “name” : “Lee Chang”,
“zip” : 94129 }
{ “_id” : ObjectId(“4c970548be67000000003859”), “name” : “Jenny Gonzalez”,
“zip” : 33101 }
```

- You can access this data set from most **mainstream programming languages, because drivers for those exist.**
- This location preferences example is accessed from Java, PHP, Ruby, and Python.

Mongo Java Driver

First, download the latest distribution of the MongoDB Java driver from the MongoDB github code repository at http://github.com/mongodb.

Once again **start the local MongoDB server by running bin/mongod** from within the MongoDB distribution. Now use a Java program to connect to this server.

A sample Java program that connects to MongoDB, lists all the collections in the prefs database, and then lists all the documents within the location collection.

```
import java.net.UnknownHostException;
import java.util.Set;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.Mongo;
import com.mongodb.MongoException;

public class ConnectToMongoDB {
Mongo m = null;
DB db;
public void connect() {
try {
m = new Mongo("localhost", 27017 );
} catch (UnknownHostException e) {
e.printStackTrace();
} catch (MongoException e) {
e.printStackTrace(); } }

public void listAllCollections(String dbName) {
if(m!=null){
db = m.getDB(dbName);
Set<String> collections = db.getCollectionNames();
for (String s : collections) {

public void listLocationCollectionDocuments() {
if(m!=null){
db = m.getDB("prefs");
DBCollection collection = db.getCollection("location");
DBCursor cur = collection.find();
while(cur.hasNext()) {
System.out.println(cur.next());
}
} else {
System.out.println("Please connect to MongoDB
and then fetch the collection");
}
}
public static void main(String[] args) {
ConnectToMongoDB connectToMongoDB = new
ConnectToMongoDB();
connectToMongoDB.connect();
connectToMongoDB.listAllCollections("prefs");
connectToMongoDB.listLocationCollectionDocuments();
}
}
```

On running the program, the output is as follows:

```
location
system.indexes
{ “_id” : { “$oid” : “4c97053abe67000000003857”} , “name” : “John Doe” ,
“zip” : 10001.0}
{ “_id” : { “$oid” : “4c970541be67000000003858”} , “name” : “Lee Chang” ,
“zip” : 94129.0}
{ “_id” : { “$oid” : “4c970548be67000000003859”} , “name” : “Jenny Gonzalez” ,
“zip” : 33101.0}

{ “_id” : { “$oid” : “4c970555be6700000000385a”} , “name” : “Srinivas
Shastri” ,
“zip” : 1089.0}
{ “_id” : { “$oid” : “4c97a6555c760000000054d8”} , “name” : “Don Joe” ,
“zip” : 10001.0}
{ “_id” : { “$oid” : “4c97a7ef5c760000000054da”} , “name” : “John Doe” ,
“zip” : 94129.0}
{ “_id” : { “$oid” : “4c97add25c760000000054db”} , “name” : “Lee Chang” ,
“zip” : 94129.0 , “streetAddress” : “37000 Graham Street”}
```

Interfacing and Interacting with NoSQL

- This chapter **introduces the essential ways of interacting with NoSQL data stores.**
- The types of **NoSQL stores vary and so do the ways of accessing and interacting with them.**
- The chapter attempts to **summarize a few of the most prominent of these disparate ways of accessing and querying data in NoSQL databases.**

IF NO SQL, THEN WHAT?

A big reason for the **popularity of relational databases** is their standardized **access and query mechanism using SQL**.

SQL, short for structured query language, is the **language you speak when talking to relational databases**.

It **involves a simple intuitive syntax** and **structure that users become fluent in within a short period of time.**

Based on **relational algebra, SQL allows users to fetch records from a single collection or join records across tables**

To fetch all data from a table that maintains the names and e-mail addresses of all the people in an organization

SELECT * FROM people.

To get just a list of names of all the people from the people table

SELECT name FROM people.

To get a subset from this list, say only those who have a Gmail account

SELECT * FROM people where email LIKE '%gmail.com'.

To get a list of people with a list of books they like, assuming that names of people and titles of books they like are in a separate but associated table named books_people_like

SELECT people.name, books_people_like.book_title FROM people, books_people_like WHERE

people.name = books_people_like.person_name.

The table books_people_like has two columns: person_name and title, where person_name references the same set of values as the name column of the people table and title column stores book titles

Storing and Accessing Data

- A more detailed view into the world of NoSQL data storage and access.

- To explain the different ways of data storage and access in NoSQL, first classify them into the following types:

Document store — MongoDB and CouchDB
Key/value store (in-memory, persistent and even ordered) — Redis and BerkeleyDB
Column-family-based store — HBase and Hypertable
Eventually consistent key/value store — Apache Cassandra and Voldermot

Storing Data In and Accessing Data from MongoDB

The Apache web server **Combined Log Format captures the following request and response attributes for a web server:**

- **IP address of the client** — This value could be the IP address of the proxy if the client requests the resource via a proxy.
- **Identity of the client** — Usually this is not a reliable piece of information and often is not recorded.
- **User name as identifi ed during authentication** — This value is blank if no authentication is required to access the web resource.
- **Time when the request was received** — Includes date and time, along with timezone.
- **The request itself** — This can be further broken down into four different pieces: method used, resource, request parameters, and protocol.
- **Status code** — The HTTP status code.
- **Size of the object returned** — Size is bytes.
- **Referrer** — Typically, the URI or the URL that links to a web page or resource.
- **User-agent** — The client application, usually the program or device that accesses a web page or resource.

The corresponding line in the log file is as follows:
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700]
“GET /apache_pb.gif HTTP/1.0” 200
2326 ”http://www.example.com/start.html” ”Mozilla/4.08 [en] (Win98; I ;Nav)”

- ***MongoDB is a document store that can persist arbitrary collections of data as long as it can be represented using a JSON-like object hierarchy***

- *To present a flavor of the JSON format, a* ***log file element extracted from the access log can be represented as follows:***

{
“ApacheLogRecord”: {
“ip”: “127.0.0.1”,
“ident” : “-”,
“http_user” : “frank”,
“time” : “10/Oct/2000:13:55:36 -0700”,
“request_line” : {
“http_method” : “GET”,
“url” : “/apache_pb.gif”,
“http_vers” : “HTTP/1.0”,
},
“http_response_code” : “200”,
“http_response_size” : “2326”,
“referrer” : “http://www.example.com/start.html”,
“user_agent” : “Mozilla/4.08 [en] (Win98; I ;Nav)”, }, }

To insert the JSON-like document for the line in the log file into a collection named logdata*, you could do the following in the Mongo shell:*

```
doc = {
“ApacheLogRecord”: {
“ip”: “127.0.0.1”,
“ident” : “-”,
“http_user” : “frank”,
“time” : “10/Oct/2000:13:55:36 -0700”,
“request_line” : {
“http_method” : “GET”, “url” : “/apache_pb.gif”,
“http_vers” : “HTTP/1.0”,
},
“http_response_code” : “200”,
“http_response_size” : “2326”,
“referrer” : “http://www.example.com/start.html”,
“user_agent” : “Mozilla/4.08 [en] (Win98; I ;Nav)”,
},
};
db.logdata.insert(doc);
```

Mongo also provides a convenience method, named save*, which updates a record if it exists in the collection and inserts it if it’s not.*

Storing data through program

- The log file itself is a text file that stores each request in a separate row.
- To get the data from the text fi le, you would need to parse it and extract the values.
- A simple elementary Python program to parse this log file

```
import re
import fileinput
_lineRegex = re.compile(r'(\d+\.\d+\.\d+\.\d+) ([^ ]*) ([^ ]*)
\[([^\]]*)\] "([^"]*)" (\d+) ([^ ]*) "([^"]*)" "([^"]*)"')

class ApacheLogRecord(object):

def __init__(self, *rgroups ):
self.ip, self.ident, \
self.http_user, self.time, \
self.request_line, self.http_response_code, \
self.http_response_size, self.referrer, self.user_agent =
rgroups
self.http_method, self.url, self.http_vers =
self.request_line.split()

def __str__(self):
return ' '.join([self.ip, self.ident, self.time, self.request_line,
self.http_res ponse_code, self.http_response_size,
self.referrer,
self.user_agent])
```

```
class ApacheLogFile(object):

def __init__(self, *filename):
self.f = fileinput.input(filename)

def close(self):
self.f.close()

def __iter__(self):
match = _lineRegex.match
for line in self.f:
m = match(line)
if m:
try:
log_line = ApacheLogRecord(*m.groups())
yield log_line
except GeneratorExit:
pass
except Exception as e:
print "NON_COMPLIANT_FORMAT: ", line, "Exception: ",
e
```

After the data is available from the parser, you can persist the data in MongoDB using pymongo.

- In the Python example, you could save data in a dictionary (also referred to as map, hash map,or associative arrays in other languages) directly to MongoDB.
- This is because PyMongo — the driver — does the job of translating a dictionary to a BSON data format.
- To complete the example, create a utility function to publish all attributes of an object and their corresponding values as a dictionary like so:

```
def props(obj):
pr = {}
for name in dir(obj):
value = getattr(obj, name)
if not name.startswith('__') and not inspect.ismethod(value):
pr[name] = value
return pr
```

With this function in place, storing data to MongoDB requires just a few lines of code:

```
connection = Connection()
db = connection.mydb
collection = db.logdata
alf = ApacheLogFile(<path to access_log>)
for log_line in alf:
collection.insert(props(log_line))
alf.close()
```

Querying MongoDB

To list all the records in the logdata collection,

```
> var cursor = db.logdata.find()
  while (cursor.hasNext()) printjson(cursor.next());
```

This prints the data set in a nice presentable format like this:

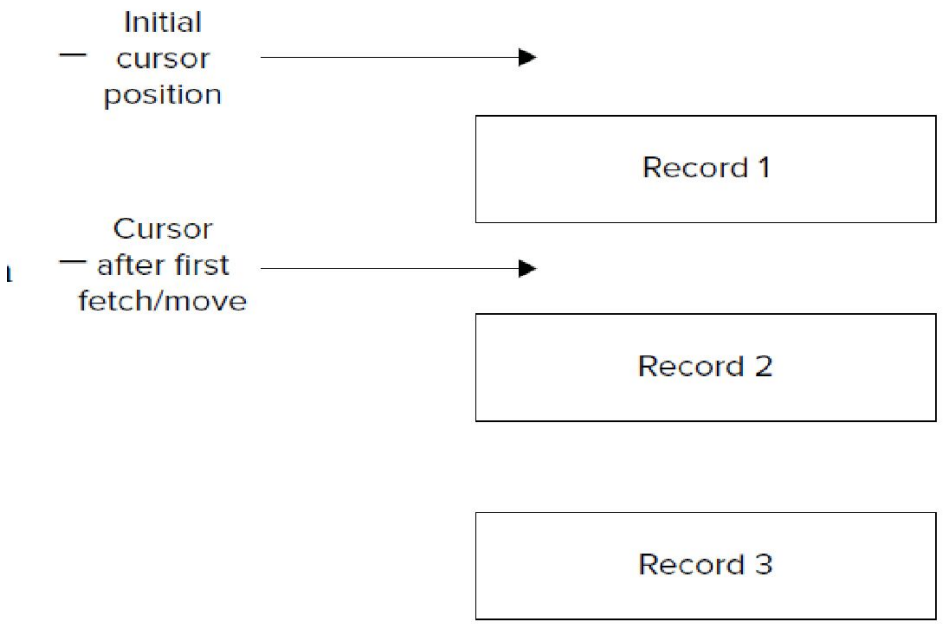
```
{
“_id” : ObjectId(“4cb164b75a91870732000000”),
“http_vers” : “HTTP/1.1”,
“ident” : “-”,
“http_response_code” : “200”,
“referrer” : “-”,
“url” : “/hi/tag/2009/”,
“ip” : “123.125.66.32”,
“time” : “09/Oct/2010:07:30:01 -0600”,
“http_response_size” : “13308”,
“http_method” : “GET”,
“user_agent” : “Baiduspider+(+http://www.baidu.com/search/spider.htm)”,
“http_user“ : “-“,
“request_line“ : “GET /hi/tag/2009/ HTTP/1.1“
}
```

Look at Figure to see how cursors work.

The method db.logdata.find() returns all the records in the logdata collection so you have the entire set to iterate over using the cursor.

The previous code sample simply iterates through the elements of the cursor and prints them out.

The printjson function prints out the elements in a nice JSON-style formatting for easy readability.



To get all **log file records where http_response_code is 200**, you can query like this:
db.logdata.find({ “http_response_code”: “200” });

This query takes a query document, { “http_response_code”: “200” }, defining the pattern, as an argument to the find method.

To **get all log fi le records where http_response_code is 200 and http_vers (protocol version) is HTTP/1.1, you can query as follows:**

db.logdata.find({ “http_response_code”:”200”, “http_vers”:”HTTP/1.1” })

To **get all log file records where the user_agent was a Baidu** search engine spider, you can query like so:
db.logdata.find({ “user_agent”: /baidu/i })

The query document actually contains a **regular expression and not an exact value.** The expression /baidu/i matches **any document that has baidu in the user_agent value**.

The **i fl ag suggests ignoring case**, so all phrases whether **baidu, Baidu, baiDU**, or BAIDU would be matched.

To get all log fi le records where user_agent starts with Mozilla, you can query as shown:
db.logdata.find({ “user_agent”: /^Mozilla/ })

To get all log fi le records **where response size is greater than 1111 k**, you could query like so:
db.logdata.find({ “http_response_size” : { $gt : 1111 }})

To get a **list of all URLs accessed by the MSN** bot as shown:
db.logdata.find({ “user_agent”:/msn/i }, { “url”:true })

To restrict the number of **rows returned in the last query to 10 as follows**:
db.logdata.find({ “user_agent”:/msn/i }, { “url”:true }).limit(10)

Storing Data In and Accessing Data from Redis

- Redis is a **persistent key/value store**.
- For efficiency it **holds the database in memory and writes to disks in an asynchronous thread**.
- The values it holds can be **strings, lists, hashes, sets, and sorted sets.**
- To begin, **start redis-cli and make sure it's working**
- Now **run redis-cli to connect to this server.**
- By default, the Redis server listens for connections on **port 6379.**
- To save a key/value pair — { akey: "avalue" } — simply type the following, from within the Redis distribution folder:
- ./redis-cli set akey "avalue"
- If you see OK on the console in response to the command you just typed, then things look good

Redis supports a few different data structures, namely:

Lists, or more specifi cally, linked lists — Collections that maintain an indexed list of elements in a particular order. With linked lists, access to either of the end points is fast irrespective of the number of elements in the list.
Sets — Collections that store unique elements and are unordered.
Sorted sets — Collections that store sorted sets of elements.
Hashes — Collections that store key/value pairs.
Strings — Collections of characters.

REDIS EXAMPLE

A **database stores a list of book titles.**

Each **book is tagged using an arbitrary set of tags.**

For example, I add "**The Omnivore's Dilemma" by "Michael Pollan"** to the list and tag it with the following: **"organic," "industrialization," "local," "written by a journalist," "best seller," and "insight"**

Add **"Outliers" by "Malcolm Gladwell**" and tag it with the following: **"insight," "best seller," and "written by a journalist."**

Now I **can get all the books on my list or all the books "written by a journalist" or all those that relate to "organic."**

I could also get a list of the books by a given author.

To implement - chose to use a set, because order isn't important.

Call this set books.

Each book, which is a member of the set of books, has the following properties:
Id
Title
Author
Tags (a collection)

Each tag is identifi ed with the help of the following properties:
Id
Name

Assuming the redis-server is running, open a redis-cli instance and input the following commands to create the first members of the set of books:

$ ./redis-cli incr next.books.id
(integer) 1
$ ./redis-cli sadd books:1:title "The Omnivore's Dilemma"
(integer) 1
$ ./redis-cli sadd books:1:author "Michael Pollan"

- The first command generates a sequence number by incrementing the set member identifier to the next id.
- Because you have just started creating the set, the output of the increment is logically “1”.
- The next two commands create a member of the set named books.
- The member is identified with the id value of 1
- The member itself has two properties — title and author — the values for which are strings.
- The sadd command adds a member to a set.

Next, add a bunch of tags to the member you added to the set named books. Here is how you do it:

```
$ ./redis-cli sadd books:1:tags 1
(integer) 1
$ ./redis-cli sadd books:1:tags 2
(integer) 1
$ ./redis-cli sadd books:1:tags 3
(integer) 1
$ ./redis-cli sadd books:1:tags 4
(integer) 1
$ ./redis-cli sadd books:1:tags 5
(integer) 1
$ ./redis-cli sadd books:1:tags 6
(integer) 1
```

Note- It is a suggested practice and convention to use a scheme like object-type:id:field for key names.

Therefore, a key such as books:1:tags implies a tag collection for a member identifi ed by the id 1 within a set named “books”

After adding a bunch of tags to the first member of the set of books, you can defi ne the tags themselves like so:

```
$ ./redis-cli sadd tag:1:name “organic”
(integer) 1
$ ./redis-cli sadd tag:2:name “industrialization”
(integer) 1
$ ./redis-cli sadd tag:3:name “local”
(integer) 1
$ ./redis-cli sadd tag:4:name “written by a journalist”
(integer) 1
$ ./redis-cli sadd tag:5:name “best seller”
(integer) 1
$ ./redis-cli sadd tag:6:name “insight”
(integer) 1
```

With tags defined, establish the reciprocal relationship to associate books that have the particular tags.

The first member has all six tags so you add it to each of the tags as follows:

```
$ ./redis-cli sadd tag:1:books 1
(integer) 1
$ ./redis-cli sadd tag:2:books 1
(integer) 1
$ ./redis-cli sadd tag:3:books 1
(integer) 1
$ ./redis-cli sadd tag:4:books 1
(integer) 1
$ ./redis-cli sadd tag:5:books 1
(integer) 1
$ ./redis-cli sadd tag:6:books 1
(integer) 1
```

Next create a second member of the set like so:

```
$ ./redis-cli incr next.books.id
(integer) 2
$ ./redis-cli sadd books:2:title “Outliers”
(integer) 1
$ ./redis-cli sadd books:2:author “Malcolm Gladwell”
(integer) 1
```

Add the tags for the second member and establish the reverse relationships to the tags themselves as follows:

```
$ ./redis-cli sadd books:2:tags 6
(integer) 1
$ ./redis-cli sadd books:2:tags 5
(integer) 1
$ ./redis-cli sadd books:2:tags 4
(integer) 1

$ ./redis-cli sadd tag:4:books 2
(integer) 1
$ ./redis-cli sadd tag:5:books 2
(integer) 1
$ ./redis-cli sadd tag:6:books 2
(integer) 1
```

Querying Redis

List the title and author of member 1, identified by the id 1, of the set of books as follows:

```
$ ./redis-cli smembers books:1:title
1.  “The Omnivore\xe2\x80\x99s Dilemma”

$ ./redis-cli smembers books:1:author
1. “Michael Pollan”
```

List all the tags for this book like so:

```
$ ./redis-cli smembers books:1:tags
1. “4”
2. “1”
3. “2”
4. “3”
5. “5”
6. “6”
```

List title, author, and tags of the second book, identifi ed by the id 2, like so:

```
$ ./redis-cli smembers books:2:title
1. “Outliers”
$ ./redis-cli smembers books:2:author
1. “Malcolm Gladwell”
$ ./redis-cli smembers books:2:tags
1. “4”
2. “5”
3. “6”
```

List all books that have the tag, identified by id 1 as follows:

```
$ ./redis-cli smembers tag:1:books
“1”
```

Tag 1 was identifi ed by the name organic and you can query that like so:

```
$ ./redis-cli smembers tag:1:name
“organic”
```

Some tags like tag 6, identified by the name insight, have been attached to both the books in the set. You can confirm that by querying the set of books that have tag 6 like so:

```
$ ./redis-cli smembers tag:6:books
1. “1”
2. “2”
```

Next, you can list the books that have both tags 1 and 6, like so:

```
$ ./redis-cli sinter tag:1:books tag:6:books
“1”
```

The sinter command allows you to query for the intersection of two or more sets.

You know both books 1 and 2 have tags 5 and 6, so a sinter between the books of tags 5 and 6 should list both books.

```
$ ./redis-cli sinter tag:5:books tag:6:books
1. “1”
2. “2”
```

To create a union of all members that contain tags 1 and 6, you can use the following command:

```
$ ./redis-cli sunion tag:1:books tag:6:books
1. “1”
2. “2”
```

Both books 1 and 2 contain tags 5 and 6, so a difference set operation between books which have tag 5 and those that have tag 6 should be an empty set. Let’s see if it’s the case

```
$ ./redis-cli sdiff tag:5:books tag:6:books
(empty list or set)
```

Storing Data In and Accessing Data from HBase

HBase could very well be considered the NoSQL flag bearer. It's an open-source implementation of the Google Bigtable,

> **HBase** is a column-oriented non-relational database management system that runs on top of Hadoop Distributed File System

Example - Feed of blog posts,
where you can extract and save the following pieces of information:
Blog post title
Blog post author
Blog post content or body
Blog post header multimedia (for example, an image)
Blog post body multimedia (for example, an image, a video, or an audio fi le)

To store this data create a collection named blogposts and save pieces of information into two categories, post and multimedia.
So, a possible entry, in JSON-like format, could be as follows:

```
{
“post” : {
“title”: “an interesting blog post”,
“author”: “a blogger”,
“body”: “interesting content”,
},
“multimedia”: {
“header”: header.png,
“body”: body.mpeg,
},
}
```

or could be like so:

```
{
“post” : {
“title”: “yet an interesting blog post”,
“author”: “another blogger”,
“body”: “interesting content”,
},
“multimedia”: {
“body-image”: body_image.png,
“body-video”: body_video.mpeg,
},
}
```

The two sample data sets share the post and multimedia categories, but don't necessarily have the same set of fields.

- In **HBase jar**gon it means they have the **same column-families — post and multimedia — but don't have the same set of columns.**
- Effectively, there are **four columns** within the multimedia column-family, namely, header, body, body-image, and body-video, and in **some data points** these columns have **no value (null**).
- In a traditional **relational database** you would have to **create all four columns and set a few values to null as required.**
- In HBase and column databases the **data is stored by column and it doesn't need to store values when they are null.**
- Thus, these are great for **sparse data sets**.

After the server is up and running, connect to the HBase local server by starting the shell like so:

```
bin/hbase shell
```

When connected, create the HBase collection blogposts with its two column-families, post and multimedia, as follows:

```
hbase(main):001:0> create ‘blogposts’, ‘post’, ‘multimedia’
```

Populate the two data points like so:

```
hbase(main):001:0> put ‘blogposts’, ‘post1’, ‘post:title’, ‘an interesting blog post’

hbase(main):003:0> put ‘blogposts’, ‘post1’, ‘post:author’, ‘a blogger’
hbase(main):004:0> put ‘blogposts’, ‘post1’, ‘post:body’, ‘interesting content’
hbase(main):005:0> put ‘blogposts’, ‘post1’, ‘multimedia:header’, ‘header.png’
hbase(main):006:0> put ‘blogposts’, ‘post1’, ‘multimedia:body’, ‘body.png’
```

```
hbase(main):012:0> put ‘blogposts’, ‘post2’, ‘post:title’, ‘yet an interesting blog post’
hbase(main):014:0> put ‘blogposts’, ‘post2’, ‘post:title’, ‘yet another blog post’
hbase(main):016:0> put ‘blogposts’, ‘post2’, ‘post:author’, ‘another blogger’
hbase(main):017:0> put ‘blogposts’, ‘post2’, ‘post:author’, ‘another blogger’
hbase(main):018:0> put ‘blogposts’, ‘post2’, ‘post:author’, ‘another blogger’
hbase(main):019:0> put ‘blogposts’, ‘post2’, ‘multimedia:body-image’,‘body_image.png’
hbase(main):021:0> put ‘blogposts’, ‘post2’, ‘post:body’, ‘interesting content’
hbase(main):022:0> put ‘blogposts’, ‘post2’, ‘multimedia:body-video’, ‘body_video.mpeg’
```

Note The two sample data points are **given ids of post1 and post2**, respectively

Querying HBase

To get all data pertaining to post1, simply query like so:

```
hbase(main):024:0> get ‘blogposts’, ‘post1’
```

```
COLUMN                    CELL
multimedia:body           timestamp=1302059666802,
value=body.png
multimedia:header         timestamp=1302059638880,
value=header.png
post:author               timestamp=1302059570361,
value=a blogger
post:body                 timestamp=1302059604738,
value=interesting content
post:title                timestamp=1302059434638,
value=an interesting blog post
5 row(s) in 0.1080 seconds
```

To get a filtered list containing only the title column for post1, query like so
:

```
hbase(main):026:0> get ‘blogposts’, ‘post1’, { COLUMN=>’post:title’ }

COLUMN                                        CELL
post:title                                    timestamp=1302059434638,
value=an interesting blog post
1 row(s) in 0.0480 seconds
```

You may recall I reentered data for the post2 title, so you could query for both versions like so:

```
hbase(main):027:0> get ‘blogposts’, ‘post2’, { COLUMN=>’post:title’, VERSIONS=>2 }

COLUMN                                        CELL
post:title                                    timestamp=1302059851203,
value=yet another blog post
post:title                                    timestamp=1302059819904,
value=yet an interesting blog post
2 row(s) in 0.0440 seconds
```

Note : By default, HBase returns only the latest version

Storing Data In and Accessing Data from Apache Cassandra

To get started, go to the Apache Cassandra installation folder and start the server in the foreground by running the following command:

```
bin/cassandra –f
```

When the server starts up, run the cassandra-cli or command-line client like so:

```
bin/cassandra-cli -host localhost -port 9160
```

Now query for available keyspaces like so:

```
show keyspaces
```

create a new keyspace called BlogPosts with the help of the following script:

```
/*schema-blogposts.txt*/
create keyspace BlogPosts
with replication_factor = 1
and placement_strategy = ‘org.apache.cassandra.locator.SimpleStrategy’;

use BlogPosts;
```

```
create column family post
with comparator = UTF8Type
and read_repair_chance = 0.1
and keys_cached = 100
and gc_grace = 0
and min_compaction_threshold = 5
and max_compaction_threshold = 31;
create column family multimedia
with comparator = UTF8Type
and read_repair_chance = 0.1
and keys_cached = 100
and gc_grace = 0
and min_compaction_threshold = 5
and max_compaction_threshold = 31;
```

Next, add the blog post sample data points like so:

```
Cassandra> use BlogPosts;
Authenticated to keyspace: BlogPosts
cassandra> set post['post1']['title'] = 'an interesting blog post';
Value inserted.
cassandra> set post['post1']['author'] = 'a blogger';
Value inserted ……
```

Querying Apache Cassandra

Assuming you are still logged in to the cassandra-cli session and the BlogPosts keyspace is in use, you can query for Post1 data like so:

```
get post['post1'];
=> (column=author, value=6120626c6f67676572,
timestamp=1302061955309000)
=> (column=body,
value=696e7465726573374696e6720636f6e74656e74,
timestamp=1302062452072000)
=> (column=title,
value=616e20696e7465726573374696e6720626c6f67207
06f7374,
timestamp=1302061834881000)
Returned 3 results.
```

Java program to list all elements of the logdata MongoDB collection

```
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.Mongo;
public class JavaMongoDBClient {
Mongo m;
DB db;
DBCollection coll;
public void init() throws Exception {
m = new Mongo( “localhost” , 27017 );
db = m.getDB( “mydb” );
coll = db.getCollection(“logdata”);
}
public void getLogData() {
DBCursor cur = coll.find();
while(cur.hasNext()) {
System.out.println(cur.next());
}
}
```

```
public static void main(String[] args) {

try{
JavaMongoDBClient javaMongoDBClient = new
JavaMongoDBClient();
javaMongoDBClient.init();
javaMongoDBClient.getLogData();
} catch(Exception e) {
e.printStackTrace();
}
}
}
```

Java program to connect and query HBase

```
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.io.RowResult;
import java.util.HashMap;
import java.util.Map;
import java.io.IOException;
public class HBaseConnector {
public static Map retrievePost(String postId) throws
IOException {
HTable table = new HTable(new HBaseConfiguration(),
“blogposts”);
Map post = new HashMap();
RowResult result = table.getRow(postId);
for (byte[] column : result.keySet()) {
post.put(new String(column), new
String(result.get(column).getValue()));
}
return post;}
```

```
public static void main(String[] args) throws IOException {
Map blogpost = HBaseConnector.retrievePost(“post1”);
System.out.println(blogpost.get(“post:title”));
System.out.println(blogpost.get(“post:author”));
}
}
```

Language Bindings for Python

Python interacts with Apache Cassandra using Pycassa.

```
import pycassa

connection = pycassa.connect('BlogPosts')
```

Once connected, get the post column-family like so:

```
column_family = pycassa.ColumnFamily(connection, 'post')
```

Now you can get the data in all the columns of the column-family with the help of a call to the get() method as follows:

```
column_family.get()
```

Language Bindings for Ruby

List all the tags for the first book, with id 1, in the books collection like so:

```
irb(main):006:0> r.smembers('books:1:tags')
⇒   ["3", "4", "5", "6", "1", "2"]
```

You can also list the books that have both tags 5 and 6 (refer to the example earlier in this chapter), like so:

```
irb(main):007:0> r.sinter('tag:5:books', 'tag:6:books')
=> ["1", "2"]
```

Understanding the Storage Architecture

Column-oriented databases are among the **most popular types of non-relational databases**.

Publication from Google on big table established beyond a doubt that **a cluster of inexpensive hardware can be leveraged to hold huge amounts data**, way more than a single machine can hold, and be **processed effectively and efficiently within a reasonable timeframe.**

Three key themes emerged:

1. Data needs to be **stored in a networked filesystem** that can expand to multiple machines.

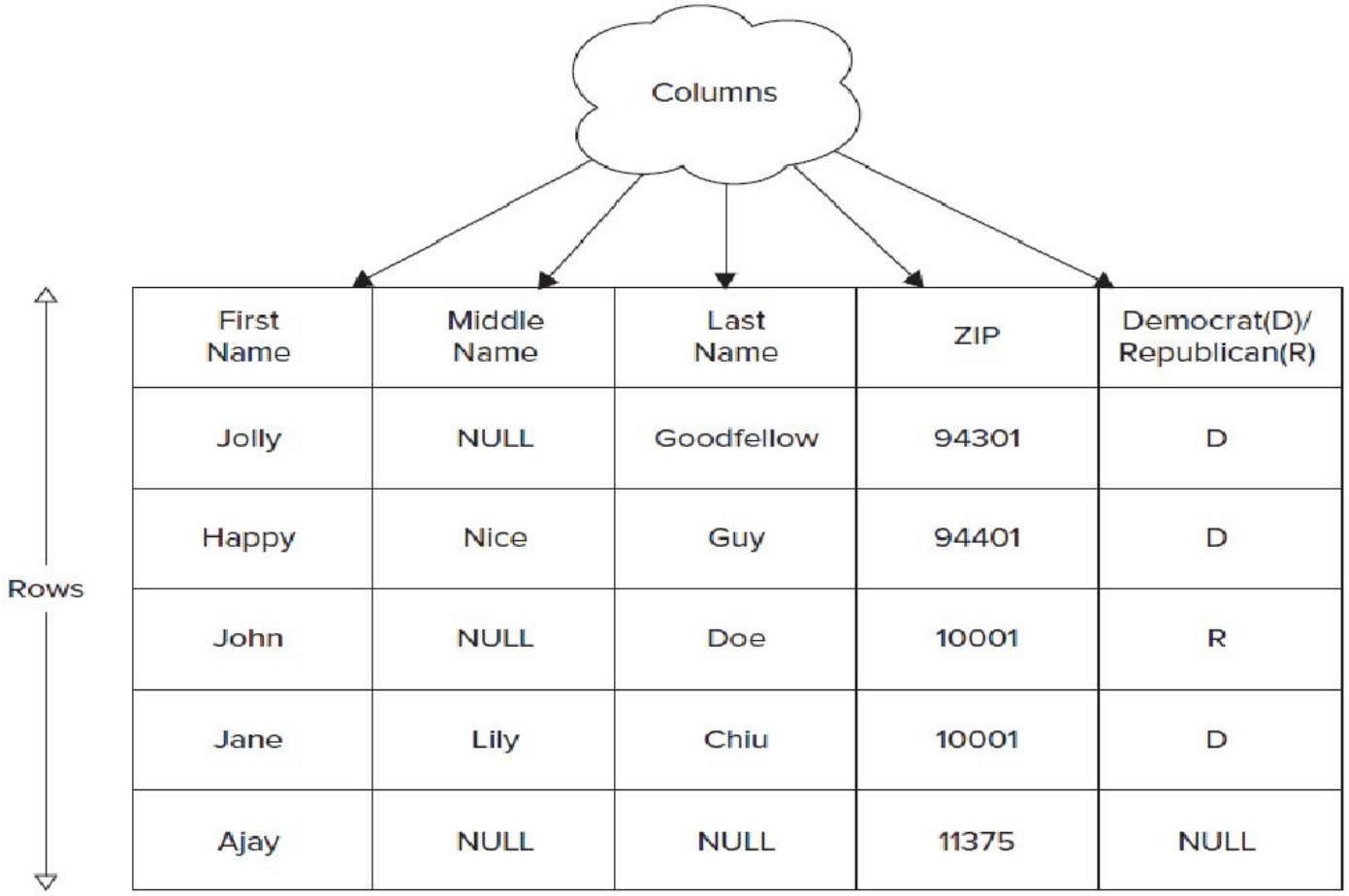
2. Data needs to **be stored in a structure that provides more flexibility than the traditional normalized relational database structures.**

The storage scheme needs to allow for **effective storage of huge amounts of sparse data sets**. It needs to accommodate for **changing schemas without the necessity of altering the underlying tables**.

3. Data needs to be **processed in a way that computations on it can be performed in isolated subsets of the data and then combined to generate the desired output.**

WORKING WITH COLUMN-ORIENTED DATABASES

Using Tables and Columns in Relational Databases



First Name	Middle Name	Last Name	ZIP	Democrat(D)/ Republican(R)
Jolly	NULL	Goodfellow	94301	D
Happy	Nice	Guy	94401	D
John	NULL	Doe	10001	R
Jane	Lily	Chiu	10001	D
Ajay	NULL	NULL	11375	NULL

FIGURE 4-1

1)RDBMS table has **a few columns**, sometimes tens of them-> **Millions of rows could** potentially be held in a relational table □may bring the data access to a **halt, unless special considerations like denormalization are applied**.

2) As you begin to **use your table** □ may need to **alter it to hold a few additional attributes**

As **newer records are stored**□ may have **null values for these attributes**□ the **existing records.**

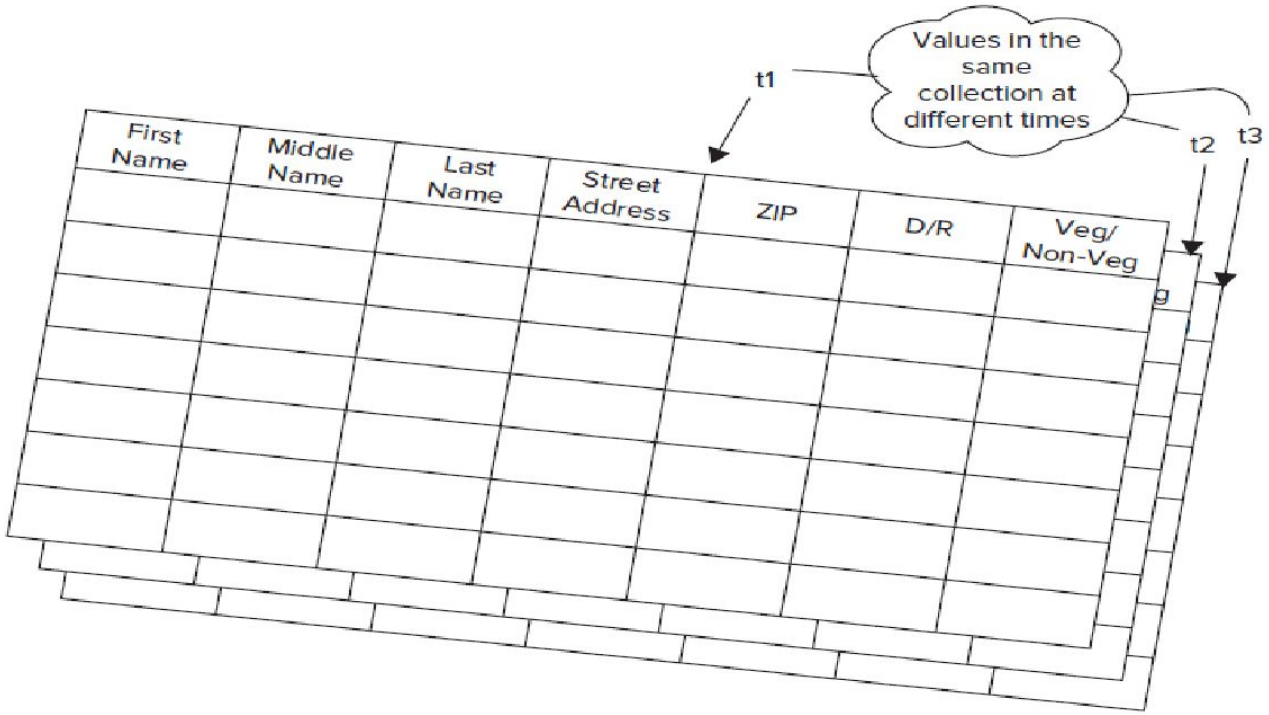
3)Keeping **greater variety of attributes** the likelihood **of sparse data sets** □ **sets with null in many cells**□ becomes **increasingly real.**

First Name	Middle Name	Last Name	Street Address	ZIP	D/R	Veg/ Non-Veg

Consider that this **data is evolving and you have to store each version of the cell value as it evolves.**

Think of it like a **three-dimensional Excel spreadsheet**, where the **third dimension is time**.

Then the **values as they evolve through time could be thought of as cell values in multiple spreadsheets** put one behind the other in chronological order



Therefore
Altering the table as data evolves,
storing a lot of sparse cells,
and
working through value versions can get complex

Contrasting Column Databases with RDBMS

- First and foremost, a **column-oriented database imposes minimal need for upfront schema definition** and can easily **accommodate newer columns as the data evolves**.
- In a typical column-oriented store, you **predefine a column-family and not a column**.
- A **column-family** is a set of **columns grouped together into a bundle**.

- In a **column database, a column-family is analogous to a column in an RDBMS**.
- Both are typically **defined before data is stored in tables and are fairly static in nature.**
- **Columns** in RDBMS define the **type of data they can store**.
- **Column-families have no such limitation**; they can contain **any number of columns**, which can store **any type of data, as far as they can be persisted as an array of bytes.**

Each **row of a column-oriented database table stores data values in only those columns for which it has valid values.**

	name	location	preferences
	first name=>"...", last name=>"..."	zip=>"..."	d/r=>"...", veg/non-veg=>"..."

Continuously **evolving data would get stored** in a column database as shown in Figure

	time	name	location	preferences
	t9	first name=>"...", last name=>"..."	zip=>"..."	d/r=>"...", veg/non-veg=>"..."
	t8			
	t7			
	t5			

On physical stores, **data isn't stored as a single table but is stored by column-families**

A **row-key uniquely identifies a row in a column database.**

Rows are ordered and split into bundles, containing contiguous values, as **data grow**

row-key	time	name
	t9	
	t8	
	t7	
	t5	

row-key	time	location
	t9	
	t8	

Column Databases as Nested Maps of Key/Value Pairs

Although **thinking of column databases as tables** with special properties is easy to understand, it **creates confusion.**

Often terms **like columns and tables immediately conjure ideas of relational** databases and **forces developers using column databases like relational stores**

Oftentimes, it's easier **to think of column databases as a set of nested maps**. **Maps** or hash maps, which are also referred to as associative arrays, are **pairs of keys and their corresponding values.**

Viewing the running example as a multidimensional nested map, you **could create the first two levels of keys in JSON-like representation,** like so:

```
{
```

```
“row_key_1” : {
“name” : {
... “location” : {
...
},
“preferences” : {
...
}
},
},
},
  “row_key_2” : {
  “name” : {
  ...
  },
  “location” : {
  ...
  },
  “preferences” : {
  ...
  }
  },
  “row_key_3” : {
```

The first-level key is the **row-key that uniquely identif es a record in a column database.**

The **second-level key is the column-family identif er.** Three column-families — name, location, and preferences — were defined earlier. Those three appear as second-level keys.

Going by the pattern**, the third-level key is the column identifier**.

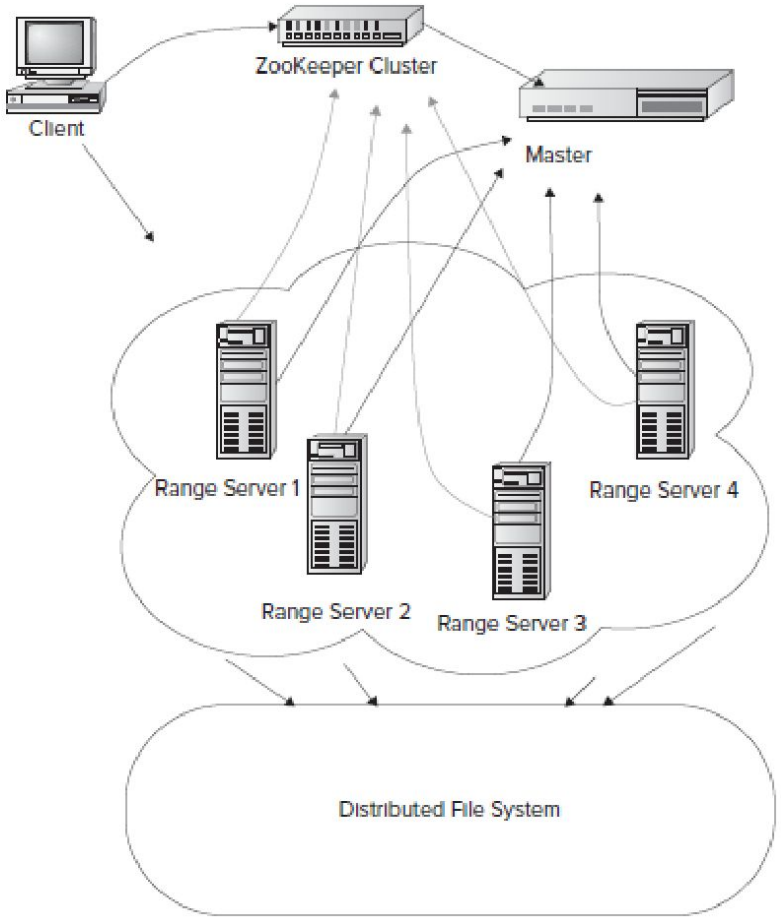
The map is like so:

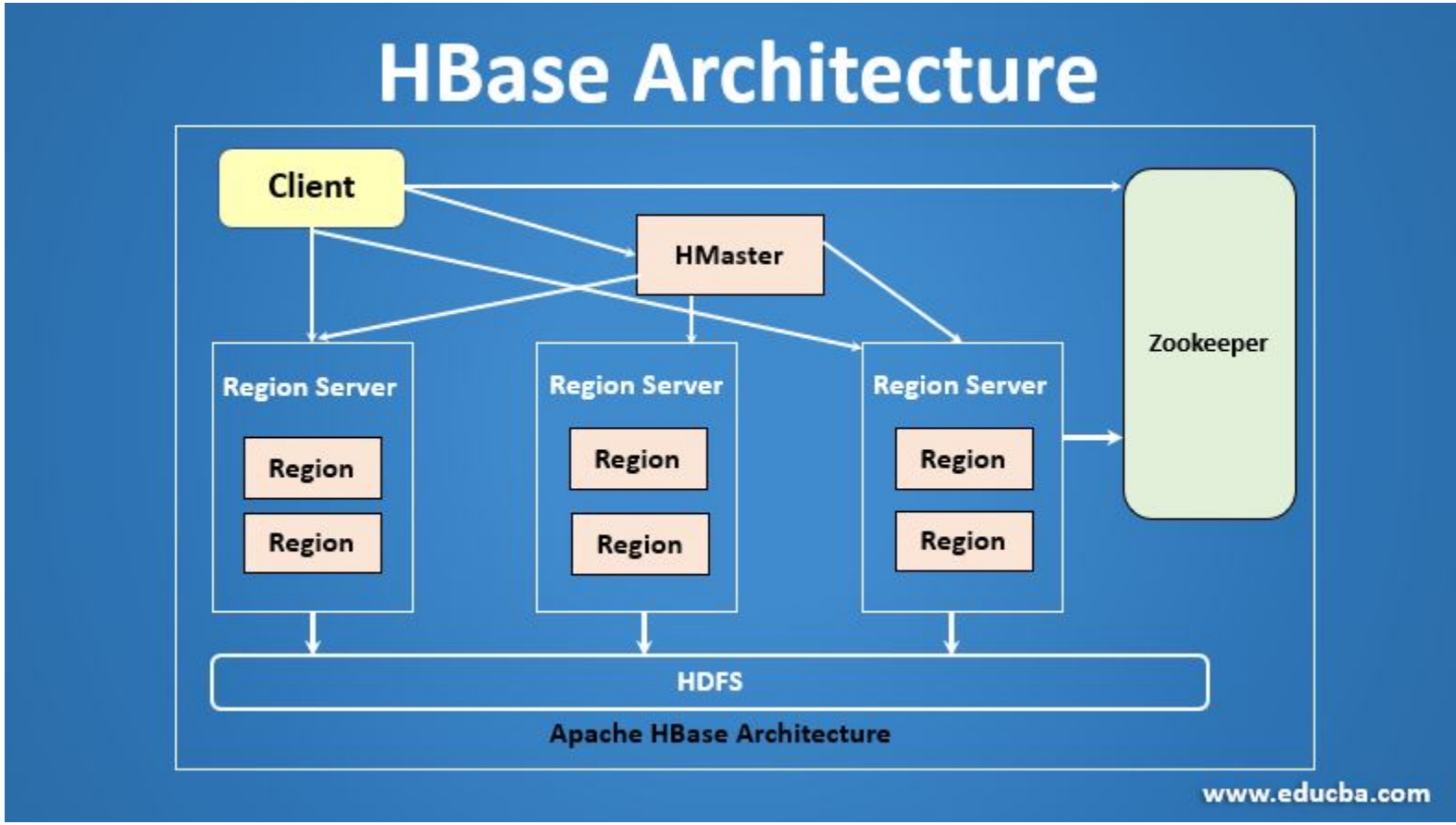
```
{
“row_key_1” : {
“name” : {
“first_name” : “Jolly”,
“last_name” : “Goodfellow”
}
}
},
“location” : {
“zip”: “94301”
},
“preferences” : {
“d/r” : “D”
}
},
“row_key_2” : {
“name” : {
“first_name” : “Very”,
“middle_name” : “Happy”,
“last_name” : “Guy”
},
“location” : {
“zip” : “10001”
```

Finally, adding **the version element to it, the third level can be expanded to include timestamped versions.**

```
{
“row_key_1” : {
“name” : {
“first_name” : {
1 : “Jolly”
},
“last_name” : {
1 : “Goodfellow”
}
},
“location” : {
“zip”: {
1 : “94301”
}
},
“preferences” : {
“d/r” : {
1 : “D”,
5 : “R”
}
}
},
...
```

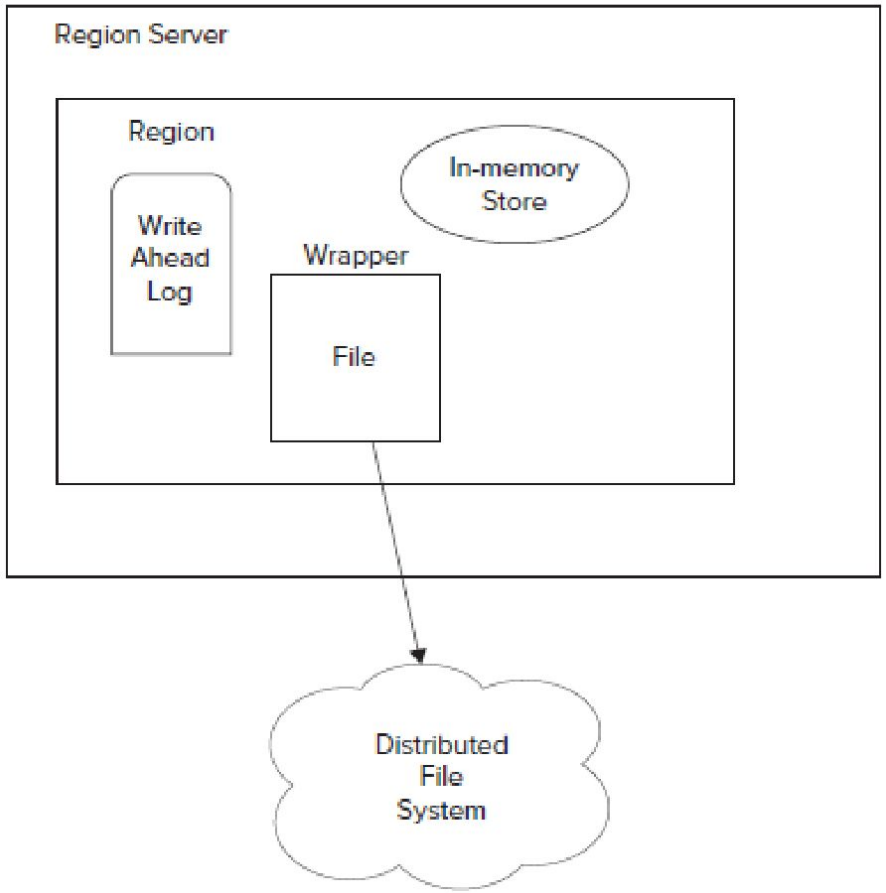
HBASE DISTRIBUTED STORAGE ARCHITECTURE





- HBase deployment adheres to **a master-worker pattern**.
- Therefore, there is usually a master and a **set of workers**, commonly known as **region servers.**
- When HBase starts, the **master allocates a set of regions to a region server.**
- Each **region stores an ordered set of rows**, where each row is identified by a unique **row-key.**
- As the number of **rows stored in a region grows** in size beyond a configured threshold, the region is split into two and rows are divided between the two new ranges.

- HBase stores **columns in a column-family together**.
- Therefore, each **region maintains a separate store for each column-family**
- Each **store in turn maps to a physical file** that is stored in the underlying distributed filesystem.
- For each store, HBase **abstracts access to the underlying filesystem with the help of a thin wrapper that acts as the intermediary between the store and the underlying physical file.**



Each region has an in-memory store, or cache, and a write-ahead-log (WAL)

When data is written to a region, it's first written to the write-ahead-log.

Soon afterwards, it's written to the region's in-memory store.

If thein-memory store is full, data is flushed to disk and persisted in the underlying distributed storage.

If distributed file system **is HDFS HBase column-family store ends up residing in an HDFS datanode**.

HBase leverages **a filesystem API to avoid strong coupling with HDFS and so this API acts as the intermediary for conversations between an HBase store and a corresponding HDFS fi le.**

ZooKeeper

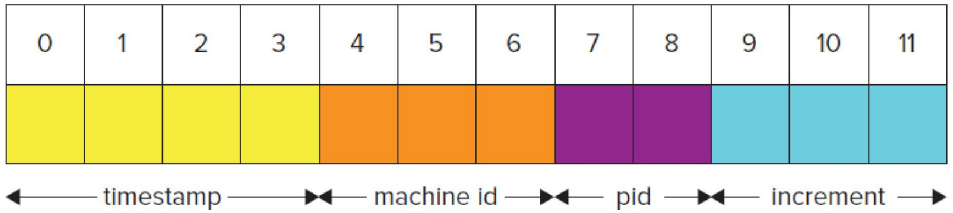
- A ZooKeeper cluster typically **front-ends an HBase cluster for new clients and manages configuration.**

- To access HBase the first time, **a client accesses two catalogs via ZooKeeper**.
- These catalogs are named **-ROOT- and .META.**
- The catalogs maintain **state and location information for all the regions.**
- **-ROOT- keeps information of all .META. tables and a .META. file keeps records the table that holds the data.**
- When a **client wants to access a specific row** it **first asks ZooKeeper for the -ROOT- catalog**.
- The **-ROOT- catalog locates the .META. catalog relevant for the row, which in turn provides all the region details for accessing the specific row.**
- Using **this information the row is accessed.**

DOCUMENT STORE INTERNALS

- MongoDB is a **document store,** where documents are grouped together into **collections.**
- Collections can be **conceptually thought of as relational tables.**
- However, collections **don't impose the strict schema constraints** that relational tables do.
- Arbitrary **documents could be grouped together in a single collection**.
- **Collections can be segregated using namespaces**
- Each **document** is stored in **BSON format**. BSON is a **binary-encoded representation of a** JSON-type document

Each document has a unique identifier,



- a 4-byte ***timestamp value***, representing the ObjectId's creation, measured in seconds since the Unix epoch
- a 5-byte ***random value*** **generated once per process**. This random value is unique to the machine and process.
- a 3-byte ***incrementing counter***, initialized to a random value

High performance is an important philosophy that pervades much of **MongoDB design**. One such **choice is demonstrated in the use of memory-mapped files for storage**.

Storing Data in Memory-Mapped Files

A **memory-mapped file is a segment of virtual memory** that is assigned **byte-for-byte to a file**
This implies that **applications can interact with such files as if they were parts of the primary memory.**

This obviously **improves I/O performance as compared to usual disk read and write.**

- **Memory mapping also introduces a few limitations**.

- For example, MongoDB's implementation restricts **data size to a maximum of 2 GB** on 32-bit systems.

- The **size of each document can be no larger than 8 MiB** and **namespaces supported is 24,000**

Guidelines for Using Collections and Indexes in MongoDB

- Although there is **no formula to determine the optimal number of collections in a database**, it's advisable to **stay away from putting a lot of disparate data into a single collection**

- A good rule of **thumb is to ask yourself whether you often need to query across the varied data set.**

- If your answer is **yes you should keep the data together, otherwise portioning it into separate collections is more efficient**

- Sometimes, a **collection may grow indefinitely and threaten to hit the 2 GB database size limit.**
- Then it may be worthwhile to use **capped collections.**
- **Capped collections** in MongoDB are like a **stack that has a predefined size.**
- When a capped collection **hits its limit, old data records are deleted.**
- Old records are identified on the basis of the **Least Recently Used (LRU) algorithm.**
- **Document fetching in capped collection follows a Last-In-First-Out (LIFO) strategy.**

MongoDB Reliability and Durability

- **MongoDB does not always respect atomicity** and does not define transactional integrity or isolation levels **during concurrent operations**.

- So it's possible for **processes to step on each other's toes while updating a collection**

  - Only a certain **class of operations**, called **modifier operations, offers atomic consistency**.

MongoDB defines a few modifier operations for atomic updates:

- `$inc` — *Increments the value of a given field*
- `$set` — *Sets the value for a field*
- `$unset` — *Deletes the field*
- `$push` — *Appends value to a field*
- `$pushAll` — *Appends each value in an array to a field*
- `$addToSet` — *Adds value to an array if it isn't there already*
- `$pop` — *Removes the last element in an array*
- `$pull` — *Removes all occurrences of values from a field*
- `$pullAll` — *Removes all occurrences of each value in an array from a field*
- `$rename` — *Renames a field*

To avoid **complete loss during a system failure,** it's advisable to set up **replication**.

Two MongoDB instances can be set up in a master-slave arrangement to replicate and keep the data in synch.

In the current versions of MongoDB**, replica pairs of master and slave have been replaced with replica sets,** where **three replicas are in a set**.

One of the three assumes the **role of master** and the **other two act as slaves**

Horizontal Scaling

- MongoDB supports **auto-sharding for scaling horizontally with ease**.

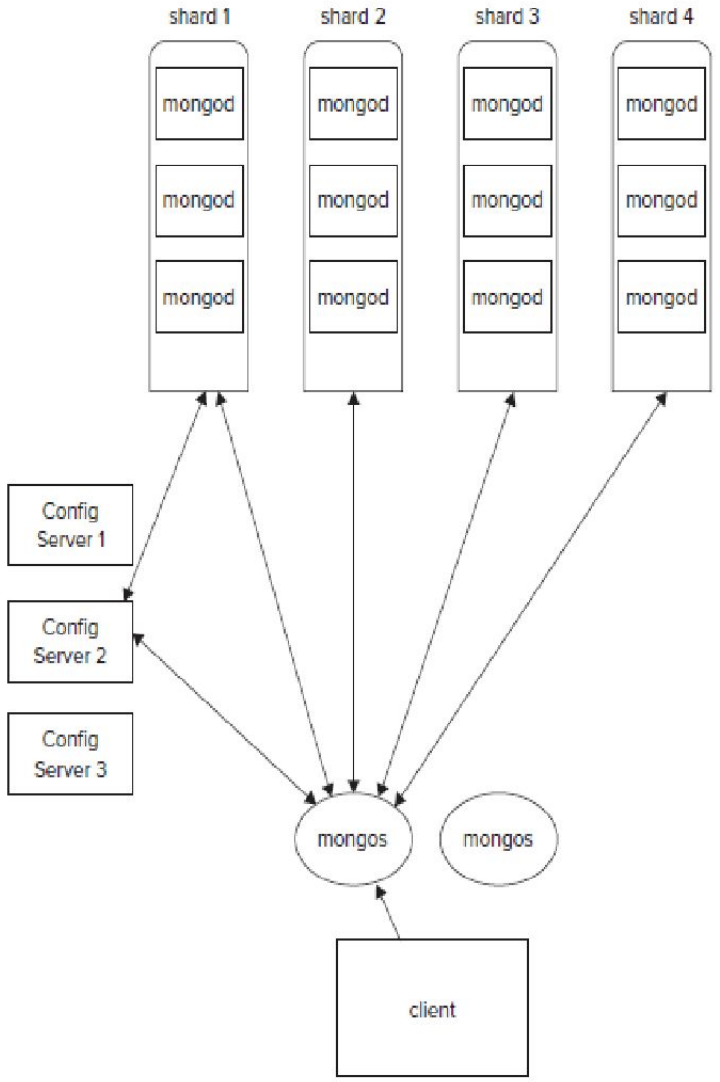
- MongoDB allows **ordered collections to be saved across multiple machines**. Each **machine that saves part of the collection is then a shard**.

- So, **a large collection could be split into four shards** and each shard in turn may be replicated three times.

- A collection can be **sharded based on any valid shard key pattern**.

- Any **document field of a collection or a combination of two or more document**
- **fields** in a collection can be **used as the basis of a shard key.**

- All **definitions about the shards and the chunks** they maintain are **kept in metadata catalogs in a config server.**



Sharding architecture topology for MongoDB.

- Client processes reach out to a MongoDB cluster via a mongos process.
- A mongos process does not have a persistent state and pulls state from the config servers.
- There can be one or more mongos processes for a MongoDB cluster.
- Mongos processes have the responsibility of routing queries appropriately and combining results where required

Performing CRUD Operations

MongoDB

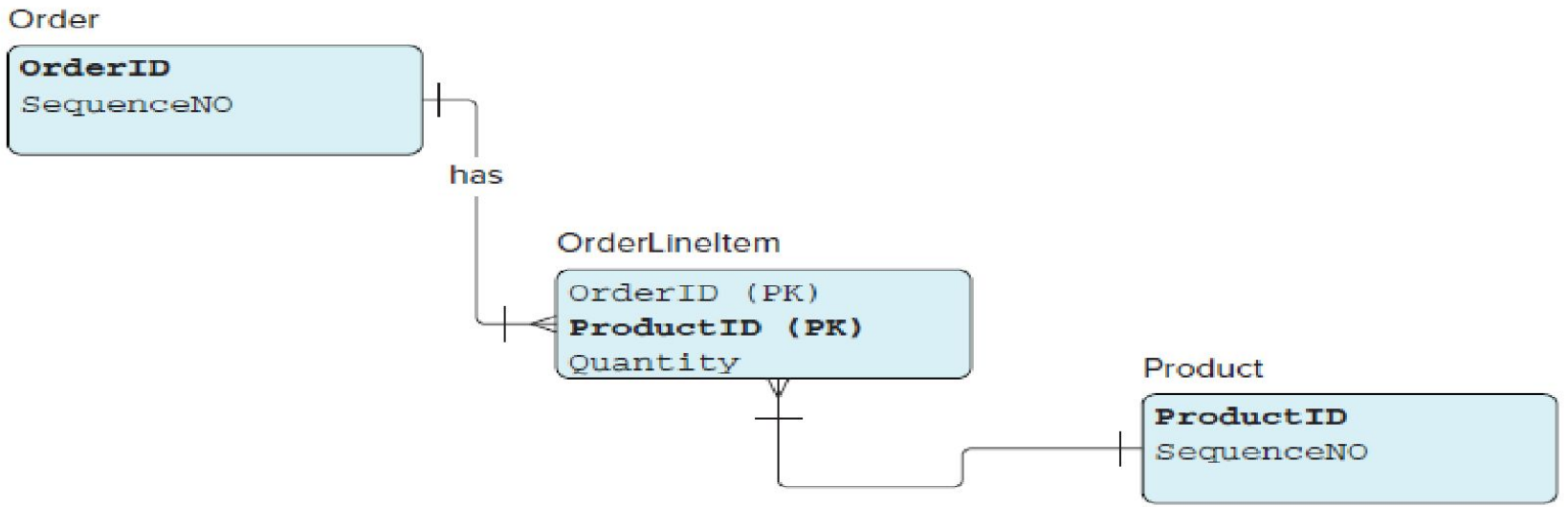
A typical example ---**retail system --which creates and manages order records**.

Each **person's purchase** at this fictitious store **is an order**.

An **order consists of a bunch of line items**.

Each order **line item includes a product** (an item) and **number of units of that product purchased**.

A line item also has a **price attribute,** which is calculated by **multiplying the unit price of the product by the number of units purchased.**



```
t = {
... order_date: new Date(),
... “line_items”: [ ...
... ]
... };
```

```
{
“order_date” : “Sat Oct 30 2010 22:30:12 GMT-0700 (PDT)”,
“line_items” : [
{
“item” : {
“name” : “latte”,
“unit_price” : 4
},
“quantity” : 1
},
{
“item” : {
“name” : “cappuccino”,
“unit_price” : 4.25
},
“quantity” : 1
{ “item” : {
“name” : “regular”,
“unit_price” : 2
},
“quantity” : 2
}
]
}
```

```
>db.orders.save(t);
> db.orders.find();
{ “_id” :
ObjectId(“4cccff35d3c7ab3d1941b103”),
“order_date” : “Sat Oct 30 2010
22:30:12 GMT-0700 (PDT)”, “line_items” : [
...
] }
```

Restructure this example in a way that **doesn't store the unit price data** for a product **in the nested document** but **keeps it separately in another collection, which stores information on products**.

The restructured orders data is stored in a collection called orders2 as follows:

```
t2 = {
... order_date: new Date(),
... "line_items": [
... {
... "item_name":"latte",
... "quantity":1
... },
... {
... "item_name":"cappuccino",
... "quantity":1
... },
... {
... "item_name":"regular",
"quantity":2
... } ... ] ... }; {
"order_date" : "Sat Oct 30 2010 23:03:31 GMT-0700
(PDT)",
"line_items" : [
{
"item_name" : "latte",
"quantity" : 1
},
{
"item_name" : "cappuccino",
"quantity" : 1
},
{
"item_name" : "regular",
"quantity" : 2 } ] }

> db.orders2.save(t2);
```

Next, save the product data, wherein item name and unit price are stored, as follows:

```
> p1 = {
... “_id”: “latte”,
... “unit_price”:4
... };
{ “_id” : “latte”, “unit_price” : 4 }
```

- **db.products.save(p1);**

> db.products.find();
{ “_id” : “latte”, “unit_price” : 4 }

Now, you could manually link the two collections and retrieve related data sets like this

```
> order1 = db.orders2.findOne();
```

```
db.products.findOne( { _id: order1.line_items[0].item_name } );
{ “_id” : “latte”, “unit_price” : 4 }
```

Alternatively, part of this manual process can be automated with the help of DBRef

Add latte, cappuccino, and regular, with their respective unit prices, to the product2 collection as follows:

```
> p4 = {“name”:”latte”, “unit_price”:4};
{ “name” : “latte”, “unit_price” : 4 }
> p5 = {
... “name”: “cappuccino”,
... “unit_price”:4.25
... };
{ “_id” : “cappuccino”, “unit_price” : 4.25 }
> p6 = {
... “name”: “regular”,
... “unit_price”:2
... };
{ “_id” : “regular”, “unit_price” : 2 }
> db.products2.save(p4);
> db.products2.save(p5);
> db.products2.save(p6);
```

Verify that all the three products are in the collection:

```
> db.products.find();
{ “_id” : ObjectId(“4ccd1209d3c7ab3d1941b105”), “name”
: “latte”,
“unit_price” : 4 }
{ “_id” : ObjectId(“4ccd1373d3c7ab3d1941b106”), “name”
: “cappuccino”,
“unit_price” : 4.25 }
{ “_id” : ObjectId(“4ccd1377d3c7ab3d1941b107”), “name”
: “regular”,
“unit_price” : 2 }
```

Next, define a new orders collection, called orders3, and use DBRef to establish the relationship between orders3 and products.

```
t3 = {
... order_date: new Date(),
... “line_items”: [
... {
... “item_name”: new DBRef(‘products2’, p4._id),
... “quantity:1
... },
... {
... “item_name”: new DBRef(‘products2’, p5._id),
... “quantity”:1
... },
... {
... “item_name”: new DBRef(‘products2’, p6._id),
... “quantity”:2
... }
... ]
... };
db.orders3.save(t3);
```

Using the Create Operation in Column-Oriented Databases

Create and maintain a large catalog of different types of **products**, where the amounts of information on the **type, category, characteristics, price, and source** of the product could vary widely.

Then you may want to **create a table with type, characteristics, and source as three column-families**

Create the products table:
hbase(main):001:0> create ‘products’, ‘type’, ‘characteristics’, ‘source’

To create a record with the following fi elds:
type:category = “coffee beans”
type:name = “arabica”
type:genus = “Coffea”
characteristics: cultivation_method = “organic”
characteristics: acidity = “low”
source: country = “yemen”
source: terrain = “mountainous

Put it into the products table like so:

```
hbase(main):001:0> put ‘products’, ‘product1’, ‘type:category’, ‘coffee beans’
0 row(s) in 0.0710 seconds
hbase(main):002:0> put ‘products’, ‘product1’, ‘type:name’, ‘arabica’
0 row(s) in 0.0020 seconds
hbase(main):003:0> put ‘products’, ‘product1’, ‘type:genus’, ‘Coffea’
0 row(s) in 0.0050 seconds
hbase(main):004:0> put ‘products’, ‘product1’,
‘characteristics: cultivation_method’, ‘organic’
0 row(s) in 0.0060 seconds
hbase(main):005:0> put ‘products’, ‘product1’, ‘characteristics: acidity’, ‘low’
0 row(s) in 0.0030 seconds
hbase(main):006:0> put ‘products’, ‘product1’, ‘source: country’, ‘yemen’
0 row(s) in 0.0050 seconds
hbase(main):007:0> put ‘products’, ‘product1’, ‘source: terrain’, ‘mountainous’
0 row(s) in 0.0050 seconds
hbase(main):008:0>
```

```
hbase(main):008:0> get ‘products’,
‘product1’
COLUMN CELL
characteristics: acidity
timestamp=1288555025970,
value=lo
characteristics: cultivatio
timestamp=128854998029,
value=organic
n_method
source: country
timestamp=1288555050543,
value=yemen
source: terrain
timestamp=1288555088136,
value=mountainous
type:category
timestamp=1288554892522,
value=coffee beans
type:genus
timestamp=1288554961942,
value=Coffea
type:name
timestamp=1288554934169,
value=Arabica
```

If you put in a value for “type:category” a second time stored as “beans” instead of its original value of “coffee beans” as follows

hbase(main):009:0> put ‘products’, ‘product1’, ‘type:category’, ‘beans’

Now, if you get the record again, the output is as follows:

```
hbase(main):010:0> get 'products', 'product1'
COLUMN                          CELL
characteristics: acidity        timestamp=1288555025970, value=low
characteristics: cultivatio     timestamp=1288554998029, value=organic
n_method
source: country                 timestamp=1288555050543, value=yemen
source: terrain                 timestamp=1288555088136, value=mountainous
type:category                   timestamp=1288555272656, value=beans
type:genus                      timestamp=1288554961942, value=Coffea
type:name                       timestamp=1288554934169, value=Arabica
7 row(s) in 0.0370 seconds
```

To look at the last four versions of the type:category field, run the following command:

```
hbase(main):011:0> get ‘products’, ‘product1’, { COLUMN => ‘type:category’, VERSIONS => 4 }
COLUMN CELL
type:category timestamp=1288555272656, value=beans
type:category timestamp=1288554892522, value=coffee beans
```

Using the Create Operation in Key/Value Maps

You can create a Redis string key/value map like so:

```
./redis-cli set akey avalue
```

You can confi rm that the value is created successfully with the help of the get command as follows:

```
./redis-cli get akey
```

The response, as expected, is avalue

Using the Create Operation in Key/Value Maps

Redis is a simple, yet **powerful, data structure server that lets you store values as a simple key/value pair or as a member of a collection.**

Each **key/value pair can be a standalone map of strings or reside in a collection.**

A **collection** could be any of the following types: **list, set, sorted set, or hash.**

A standalone key/value string pair is like a variable that can take string values.

You can **create a Redis string key/value** map like so:
./redis-cli set akey avalue

You can **confirm that the value is created** successfully with the help of the **get command as follows:**
./redis-cli get akey
The response, as expected, is avalue.

The familiar set and get commands for a string can't be used for Redis collections

List

lpush and rpush creates and populates a list.

A nonexistent **list can be created along with its first member** as follows:

./redis-cli lpush **list_of_books** 'MongoDB: The Definitive Guide'

The **range operation** to **verify and see the first few members of the list** — list_of_books — like so:

./redis-cli lrange list_of_books 0 -1

1. "MongoDB: The Definitive Guide"

Range -Index of the first element, 0, and the index of the last element, -1, to get all elements in the list.

Members can be added to a list, either on the **left or on the right**, and can be popped from either direction as well.

This allows you to leverage **lists as queues or stacks**.

For a **set data structure**, a **member can be added using the SADD operation**. Therefore, you can add **'a set member' to aset like so**:
./redis-cli sadd aset 'a set member'

You can easily add '**a sset member' to a sorted set, called azset**, like so:

./redis-cli zadd azset 1 'a sset member'

The value **1 is the position or score of the sorted set member**.

You can add another member, 'sset member 2', to this sorted set as follows:
./redis-cli zadd azset 4 'sset member 2'

The sorted set **range command is called zrange** and you can **ask for a range containing the first five values as follows:**

./redis-cli zrange azset 0 4
1. “a sset member”
2. “sset member 2”

Adding a value to **azset at score 3** like so:
./redis-cli zadd azset 3 ‘member 3’

./redis-cli zrange azset 0 4
reveals:
1. “a sset member”
2. “member 3”
3. “sset member 2”

Adding a **value at position or score 3 again**, like so:
./redis-cli zadd azset 3 ‘member 3 again’

and running the zrange query like so:
./redis-cli zrange azset 0 4

the **members have been re-positioned to accommodate** the new member, like so:
1. “a sset member”
2. “member 3”
3. “member 3 again”
4. “sset member 2”

ACCESSING DATA

Accessing Documents from MongoDB

First, **get all the documents** in the orders collection like so:
db.orders.find()

Get all the **orders after October 25, 2010**, that is, with order_date greater than October 25, 2010.

```
var refdate = new Date(2010, 9, 25);
db.orders.find({“order_date”: {$gt: refdate}});
```

To get all documents from the orders collection where line item name is latte

```
db.orders.find({ “line_items.item.name” : “latte” })
```

Accessing Data from HBase

The **easiest and most efficient query to run** on HBase is one that is **based on the row-key**.

Rowkeys in HBase are **ordered** and ranges of these **contiguous row-keys are stored together**.

It's a good idea to **relate the row-key semantically to the data contained** in the table

Orders table Row-keys that are a **combination of the item or product name, the order date, and possibly category.**

Depending on how the data is to be most often **accessed**, the combination **sequence of these three fields could vary.**

So, **if orders will be most often accessed in chronological order**, you may want to create row-keys like so:
<date> + <timestamp> + <category> + <product>

However, if orders would most often be accessed by the category and product names, then create a row-key like so:
<category> + <product> + <date> + <timestamp>

Updating and Modifying Data in MongoDB, HBase, and Redis

Databases like **MongoDB are meant to be sharded and scalable**.
In such a situation, **locking across distributed shards can be complex** and make the process of data updates very slow.

Available atomic methods are as follows:

$set — Set a value
$inc — Increment a particular value by a given amount
$push — Append a value to an array
$pushAll — Append several values to an array
$pull — Remove a value from an existing array
$pullAll — Remove several value(s) from an existing array

HBase supports a row-level read-write lock. This means rows are locked when any column in that
row is being modified, updated, or created

Querying NoSQL Stores

To understand the MongoDB query language capabilities and see how it performs, start by loading a data set into a MongoDB database.

Eg: MOVIELENS- Three Files

- movies.dat - <MovieID>::<Title>::<Genres> -
  1::Toy Story (1995)::Animation|Children's|Comedy
- ratings.dat- UserID::MovieID::Rating::Timestamp
- users.dat- UserID::Gender::Age::Occupation::Zip-code

1. Extract and loads the data from the users, movies, and ratings data files to respective MongoDB collections

2. On your Mongo JavaScript shell, first get a count of all the values in the ratings collection as follows:

db.ratings.count();
In response, you should see 1000209

3. get a sample set of the ratings data

db.ratings.find();

Note : The ratings data, for example, { “_id” : ObjectId(“4cdcf1ea5a918708b0000001”), “user_ id” : 1, “movie_id” : 1193, “rating” : 5, “timestamp” : “978300760” }

4. to get a list of all records in the movies collection that have Titanic or titanic in their title,

```
db.movies.find({ title: /titanic/i});
```

This query returns the following set of documents:

```
{ “_id” : 1721, “title” : “Titanic (1997)”, “genres” : [ “Drama”, “Romance” ] }
{ “_id” : 2157, “title” : “Chambermaid on the Titanic, The (1998)”, “genres” : “Romance” }
{ “_id” : 3403, “title” : “Raise the Titanic (1980)”, “genres” : [ “Drama”, “Thriller” ] }
{ “_id” : 3404, “title” : “Titanic (1953)”, “genres” : [ “Action”, “Drama” ] }
```

The title field in the MovieLens data set includes the year the movie was released.
Within the title field, the release year is included in parentheses. So, if you remembered or happen to know that
Titanic was released in the year 1997, you can write a more tuned query expression as follows:

```
db.movies.find({ title: /titanic.*\(1997\).*/i});
```

This returns just one document:

```
{ "_id" : 1721, "title" : "Titanic (1997)", "genres" : [ "Drama", "Romance" ] }
```

To get all ratings for the movie *Titanic*, which has an id of 1721, you could query like so:

```
db.ratings.find({ movie_id: 1721 });
```

To find out the number of available ratings for *Titanic*, you can count them as follows:

```
db.ratings.find({ movie_id: 1721 }).count();
```

The response to the count is 1546.
The ratings are on a 5-point scale. To get a list and count of only the 5-star ratings for the movie *Titanic* you can further filter the record set like so:

```
db.ratings.find({ movie_id: 1721, rating: 5 });
db.ratings.find({ movie_id: 1721, rating: 5 }).count();
```

To find out the distinct set of ratings by users (from the possible set of integers between 1 and 5, both inclusive), you could query
as follows:

db.runCommand({ distinct: ‘ratings’, key: ‘rating’, query: { movie_id: 1721} });

Ratings for *Titanic* include all possible cases between 1 and 5 (both inclusive) so the response is like so:

{ “values” : [1, 2, 3, 4, 5], “ok” : 1 }

runCommand takes the following arguments:

Collection name for the field labeled distinct

Field name for key, whose distinct values would be listed

Query to optionally filter the collection

Note: db.ratings.distinct(“rating”); could also be used

- You know from the distinct values that *Titanic* has all possible ratings from 1 to 5.
- To see how these ratings break down by each rating value on the 5-point scale, you could group the counts like so:

```
db.ratings.group(
... { key: { rating:true },
... initial: { count:0 },
... cond: { movie_id:1721 },
... reduce: function(obj, prev) { prev.count++; }
... }
... );
```

The output of this grouping query is an array as follows:

```
[
{
“rating” : 4,
“count” : 500
},
{
“rating” : 1,
“count” : 100
},
{
“rating” : 5,
“count” : 389
},
```

Theoretically, the example could easily be morphed into a case where **ratings for each movie are grouped** by the **rating points** by simply using the following group operation:

```
db.ratings.group(
... { key: { movie_id:true, rating:true },
... initial: { count:0 },
... reduce: function(obj, prev) { prev.count++; }
... }
... );
```

In real cases, though, this **wouldn't work for the ratings collection of 1 million items**. You would be greeted instead with the following error message:

```
Fri Nov 12 14:27:03 uncaught exception: group command failed: {
"errmsg" : "exception: group() can't handle more than 10000 unique keys",
"code" : 10043,
"ok" : 0
}
```

A map function applies a function to every member of the collection and emits a key/value pair for each member as an outcome of this process.

The key/value output of a map function is consumed by the reduce function.

The reduce function runs an aggregation function across all key/value pairs and generates an output in turn.

The map function to count the number of female (F) and male (M) respondents in the users collection is as follows:

```
> var map = function() {
... emit({ gender:this.gender }, { count:1 }); ... };
```

```
> var reduce = function(key, values) {
... var count = 0;
... values.forEach(function(v) {
... count += v['count'];
... });
...
... return { count:count };
... };
```

A reduce function **takes a key/value pair emitted by the map function**. In this particular reduce function, **each value in the key/value pair is passed through a function that counts the number of occurrences of a particular type.**

Finally, **running this map and reduce function pair against the users collection** leads to an output of the **total count of female and male members in the users collection**.

The mapReduce run and result extraction commands are as follows:

```
> var ratings_respondents_by_gender = db.users.mapReduce(map, reduce);
> ratings_respondents_by_gender
{
“result” : “tmp.mr.mapreduce_1290399924_2”,
“timeMillis” : 538,
“counts” : {
“input” : 6040,
“emit” : 6040,
“output” : 2
},
“ok” : 1,
}
> db[ratings_respondents_by_gender.result].find();
{ “_id” : { “gender” : “F” }, “value” : { “count” : 1709 } }
{ “_id” : { “gender” : “M” }, “value” : { “count” : 4331 } }
```

To **verify the output**, filter the users collection for gender values “F” and “M” and count the number of documents in each filtered sub-collection.

The commands for fi ltering and counting the users collection for gender values “F” and “M” is like so:

```
> db.users.find({ “gender”:”F” }).count();
1709
> db.users.find({ “gender”:”M” }).count();
4331
```

Next, **modify the map function slightly and run the map and reduce functions against the ratings collection to count the number of each type of rating** (1, 2, 3, 4 or 5) for each movie

```
> var map = function() {
... emit({ movie_id:this.movie_id,
rating:this.rating }, { count:1 });
... };
> var reduce = function(key, values) {
... var count = 0;
... values.forEach(function(v) {
... count += v['count'];
... });
...
... return { count: count };
... };
> var group_by_movies_by_rating =
db.ratings.mapReduce(map, reduce);
>
db[group_by_movies_by_rating.result].find(
```

To get a count of each type of rating for the movie *Titanic*, identifi ed by movie_id 1721, you simply filter the MapReduce output using nested property access method like so:

- db[group_by_movies_by_rating.result].find({ “_id.movie_id”:1721 });

{ “_id” : { “movie_id” : 1721, “rating” : 1 }, “value” : { “count” : 100 } }
{ “_id” : { “movie_id” : 1721, “rating” : 2 }, “value” : { “count” : 176 } }
{ “_id” : { “movie_id” : 1721, “rating” : 3 }, “value” : { “count” : 381 } }
{ “_id” : { “movie_id” : 1721, “rating” : 4 }, “value” : { “count” : 500 } }
{ “_id” : { “movie_id” : 1721, “rating” : 5 }, “value” : { “count” : 389 } }

Next, you could calculate the average rating for each movie in the ratings collection as follows:

```
> var map = function() {
... emit({ movie_id:this.movie_id }, {
rating:this.rating, count:1 });
... };
> var reduce = function(key, values) {
... var sum = 0;
... var count = 0;
... values.forEach(function(v) {
... sum += v['rating'];
... count += v['count'];
... });
...
... return { average:(sum/count) };
... };
 var average_rating_per_movie =
    db.ratings.mapReduce(map, reduce)

 db[average_rating_per_movie.result]
find();
```

ACCESSING DATA FROM COLUMN-ORIENTED DATABASES LIKE HBASE

The Historical Daily Market Data

The data fi elds are partitioned logically into three different types:

- Combination of exchange, stock symbol, and date served as the unique id
- The open, high, low, close, and adjusted close are a measure of price
- The daily volume

The row-key can be created using a combination of the exchange, stock symbol, and date. So

NYSE,AA,2008-02-27 could be structured as NYSEAA20080227 to be a row-key for the data.

All **price-related information can be stored in a column-family named price and volume data can be stored in a column-family named volume.**

The table itself is named historical_daily_stock_price.

To get the row data for NYSE, AA,
2008-02-27, you can query as follows:

get ‘historical_daily_stock_price’, ‘NYSEAA20080227’

You can get the open price as follows:
get ‘historical_daily_stock_price’, ‘NYSEAA20080227’, ‘price:open’

```
import org.apache.hadoop.hbase.client.HTable;
Import
org.apache.hadoop.hbase.HBaseConfigura
tion;
import
org.apache.hadoop.hbase.io.RowResult;
import java.util.HashMap;
import java.util.Map;
import java.io.IOException;

public class HBaseConnector {
public static Map retrievePriceData(String
rowKey) throws IOException {
HTable table = new HTable(new
HBaseConfiguration(),
“historical_daily_stock_price”);
Map stockData = new HashMap();
RowResult result = table.getRow(rowKey);
for (byte[] column : result.keySet()) {
stockData.put(new String(column), new
String(result.get(column).getValue()));
}
return stockData;
}
public static void main(String[] args) throw
IOException {
Map stock_data =
HBaseConnector.retrievePriceData
(“NYSEAA20080227”);
System.out.println(stock_data.get
(“price:open”));
System.out.println(stock_data.get
(“price:high”));
}
}
```

For the purpose of demonstration, the NYC Data Mine public raw data on parking spaces available online at www.nyc.gov/data is used.

```
import csv
import redis
f = open(“parking_facilities.csv”, “r”)
parking_facilities = csv.DictReader(f, delimiter=’,’)
r = redis.Redis(host=’localhost’, port=6379, db=0)
```

```
def add_parking_facility(license_number,
facility_type,
entity_name,
camis_trade_name,
address_bldg,
address_street_name,
address_location,
address_city,
address_state,
address_zip_code,
telephone_number,
number_of_spaces):
if r.sadd(“parking_facilities_set”, license_number):
r.hset(“parking_facility:%s” % license_number,
“facility_type”, facility_type)
r.hset(“parking_facility:%s” % license_number,
“entity_name”, entity_name)
r.hset(“parking_facility:%s” % license_number,
“camis_trade_name”, camis_trade_name)
r.hset(“parking_facility:%s” % license_number,
“address_bldg”, address_bldg)
r.hset(“parking_facility:%s” % license_number,
“address_street_name”, address_street_name)
r.hset(“parking_facility:%s” % license_number,
“address_location”, address_location)
```

```
r.hset("parking_facility:%s" % license_number,
"address_city", address_city)
r.hset("parking_facility:%s" % license_number,
"address_state", address_state)
r.hset("parking_facility:%s" % license_number,
"address_zip_code", address_zip_code)
r.hset("parking_facility:%s" % license_number,
"telephone_number", telephone_number)
r.hset("parking_facility:%s" % license_number,
"number_of_spaces", number_of_spaces)
return True
else:
return False
```

```
if __name__ == "__main__":
for parking_facility_hash in parking_facilities:
add_parking_facility(parking_facility_hash['License
Number'],
parking_facility_hash['Facility Type'],
parking_facility_hash['Entity Name'],
parking_facility_hash['Camis Trade Name'],
parking_facility_hash['Address Bldg'],
parking_facility_hash['Address Street Name'],
parking_facility_hash['Address Location'],
parking_facility_hash['Address City'],
parking_facility_hash['Address State'],
parking_facility_hash['Address Zip Code'],
parking_facility_hash['Telephone Number'],
parking_facility_hash['Number of Spaces'])
print "added parking_facility with %s" %
parking_facility_hash['License
Number']
```

To get a list of all license numbers in the set

SMEMBERS parking_facilities_set

All 1,912 license numbers in the set would be printed out.

For each parking facility the attributes are stored in a hash, which is identifi ed by the key of the form parking_facility:<license_number>.

Thus, to see all keys in the hash associated with license number 1105006 you can use the following command:

HKEYS parking_facility:1105006

The response is as follows:

1. “facility_type”
2. “entity_name”
3. “camis_trade_name”
4. “address_bldg”
5. “address_street_name”
6. “address_location”
7. “address_city”
8. “address_state”
9. “address_zip_code”
10. “telephone_number”
11. “number_of_spaces”

The license number 1105006 was fi rst

If you need the **list of members to appear in a certain order**, use the sorted sets instead of the set.

All you may need to do to use a **sorted set** is to replace the line if

r.sadd(“parking_facilities_set”, license_number): with the following:
if r.zadd(“parking_facilities_set”, license_number):

Now, you can query for **specific values in the hash**, say facility type, as follows:

HGET parking_facility:1105006 facility_type
The response is “Parking Lot”.

You can **also print out all values using the HVALS** command as follows:
HVALS parking_facility:1105006
The response is:
1. “Parking Lot”
2. “CENTRAL PARKING SYSTEM OF NEW YORK, INC”
3. “”
4. “41-61”
5. “KISSENA BOULEVARD”
6. “”
7. “QUEENS”
8. “NY”
9. “11355”
10. “2126296602”
11. “808”

Print out **all the keys and the corresponding values in a hash**. You can do that using the HGETALL command as follows:

HGETALL parking_facility:1105006

The response is as follows:

1. “facility_type”
2. “Parking Lot”
3. “entity_name”
4. “CENTRAL PARKING SYSTEM OF NEW YORK, INC”
5. “camis_trade_name”
6. “”
7. “address_bldg”
8. “41-61”
9. “address_street_name”
10. “KISSENA BOULEVARD”
11. “address_location”
12. “”
13. “address_city”
14. “QUEENS”
15. “address_state”
16. “NY”
17. “address_zip_code”
18. “11355”
19. “telephone_number”
20. “2126296602”
21. “number_of_spaces”
22. “808”

You may want to print out only the address_city and the address_zip_code as follows:

```
HMGET parking_facility:1105006 address_city address_zip_code
```

The response is:
1. “QUEENS”
2. “11355”

To get a count of the number of keys, you can use the HLEN command as follows:

```
HLEN parking_facility:1105006
```

The response is 11.

If you wanted to check if address_city was one of these, you can use the HEXISTS command to verify if it exists as a key.

The command is used as follows:

HEXISTS parking_facility:1105006 address_city

Modifying Data Stores and Managing Evolution

CHANGING DOCUMENT DATABASES

Schema-less Flexibility- Advantages

- Sparse data sets can be stored effectively because fi elds that are null needn’t be stored at all.
- As documents evolve, adding additional fi elds is trivial

create a MongoDB collection named contacts and add the two documents to the collection

```
use mydb
db.contacts.insert({ name:”John Doe”, organization:”Great Co”,
email:”john.doe@example.com” });
db.contacts.insert({ name:”Wei Chin”, company:”Work Well”, phone:”123-456-7890” });
```

Now, you could add an additional fi eld, email, to the document that relates to “Wei Chin” as follows:

```
var doc = db.contacts.findOne({
_id:ObjectId(“4d2bbb43febd3e2b32bed965”) });
doc.email = “wei.chin@example.com”;
db.contacts.save(doc);

db.contacts.find();
```

The response should be as follows:

```
{ “_id” : ObjectId(“4d2bbad6febd3e2b32bed964”), “name” : “John Doe”,
“organization” : “Great Co”, “email” : “john.doe@example.com” }
{ “_id” : ObjectId(“4d2bbb43febd3e2b32bed965”), “name” : “Wei Chin”,
“company”:
“Work Well”, “phone” : “123-456-7890”, “email” : “wei.chin@example.com” }
```

Exporting and Importing Data from and into MongoDB

mongoimport

If you have the **data to import in a single file and it is either in JSON format or is text data in comma or tab-separated** format, the **mongoimport** utility can help you import the data into a MongoDB collection.

mongoexport

The exact opposite of loading data into a MongoDB collection is to **export data out of a collection.**

If JSON or CSV format is what you need, you can use this tool to export the data out of a collection

mongodump

The mongoimport and mongoexport utilities **help import into and export from a single collection and deal with human-readable data formats**.

If the **purpose is simply to take a hot backup, you could rely on mongodump to dump a complete database copy in a binary format**

SCHEMA EVOLUTION IN COLUMN-ORIENTED DATABASES

- HBase is **not completely schema-less**.
- There is a **loosely defined schema**, especially in terms of the **column-family definitions.**
- A **column-family is a fairly static** definition that **partitions the more dynamic and flexible column definitions** into logical bundles

- All data **updates** in HBase are **overwrites with newer versions of the record** and **not in-place updates** of existing records

- Hbase is **configured to keep the last three versions** but you **could configure to store more than three version**s for each.

Eg Create 'mytable', { NAME => 'afamily', VERSIONS => 15 }

- Although you can **define a large value for the number of versions** to keep, using this data **it is not easy to retrieve and query the value because there is no built-in index based on versions**.

- Also, **versions have a timestamp but querying across this time series for the data set is not** a feature that is **easy or efficient to implement**.

Columns in HBase **don't need to be defined up front** so they provide **a flexible way of managing evolving schemas.**

Column-families, on the other hand, are **more static**.

However, **columns can't be renamed or assigned easily from one column-family to the other.**

Making such changes **requires creation of the new columns, migration of data from the existing columns to the new column, and then potentially deletion of the old columns**

HBASE DATA IMPORT AND EXPORT

The data in a table, say ‘blogposts’, can be **exported out to the local filesystem or exported to HDFS.**

You can **export the data to the local fi lesystem as follows:**
bin/hbase org.apache.hadoop.hbase.mapreduce.Driver export blogposts path/to/local/filesystem

You can **export the same data to HDFS as follows**:
bin/hbase org.apache.hadoop.hbase.mapreduce.Driver export blogposts hdfs://namenode/path/to/hdfs

Like export, you can also **import the data into an HBase** table.

You could import the **data from the**
local fi lesystem or from HDFS.

Analogous to export, **import from the local fi lesystem** is as follows:

```
bin/hbase org.apache.hadoop.hbase.mapreduce.Driver import blogposts
path/to/local/filesystem
```

Importing from HDFS is similar. You could **import the data like** so:

```
bin/hbase org.apache.hadoop.hbase.mapreduce.Driver import blogposts
hdfs://namenode/path/to/hdfs
```

DATA EVOLUTION IN KEY/VALUE STORES

- **Key/value stores usually support fairly limited data sets** and either hold string or object values.

- **Key/value databases don’t hold documents, data structures or objects and so have little sense of a schema beyond the key/value pair itself.**

- Therefore, the **notion of schema evolution isn’t that relevant to key/value databases**

Indexing and Ordering Data Sets

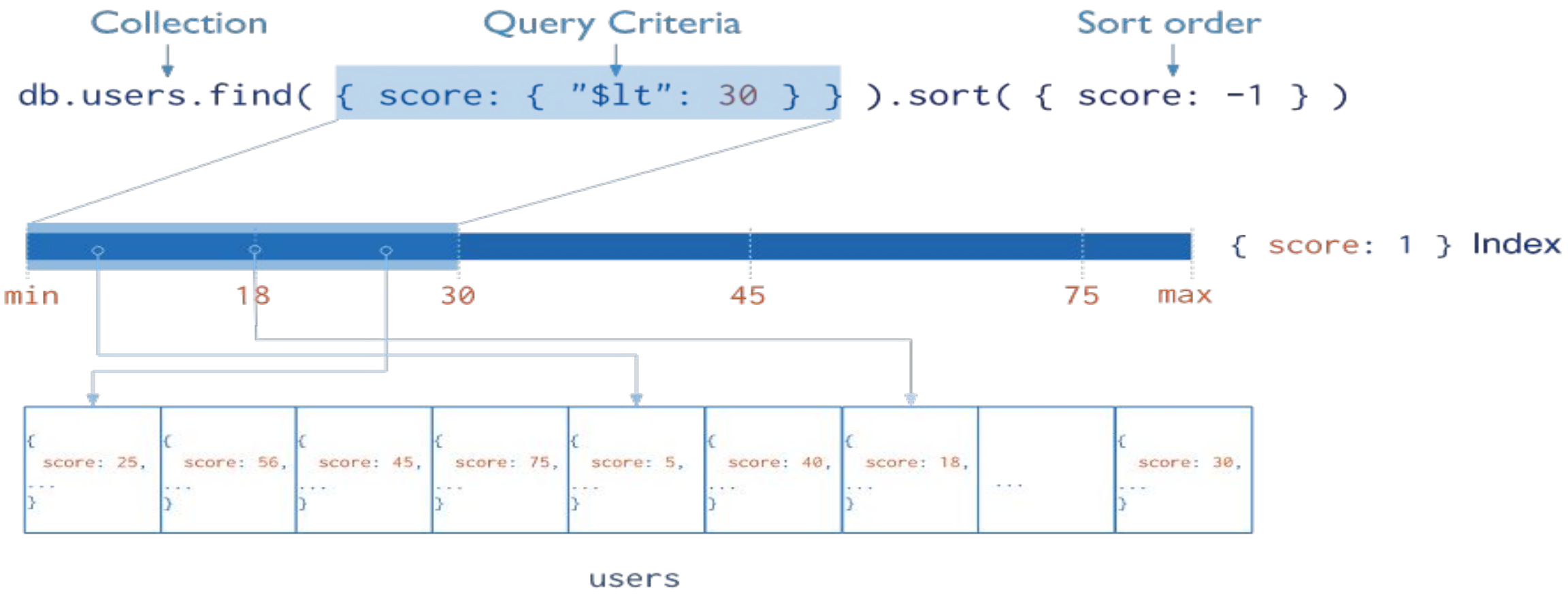
When you need to search for a term or a word in a book, you have two options, namely:

- Scan the entire book page by page to search for the term or word.
- Look up the index at the end to fi nd the pages where the term or word can be found and then browse to those specific pages.

In an analogous manner you have two choices when accessing a record in a database:

- Look through the entire collection or data set item by item.
- Leverage the index to get to the relevant data quickly.

The following diagram illustrates a query that selects and orders the matching documents using an index:



ESSENTIAL CONCEPTS BEHIND A DATABASE INDEX

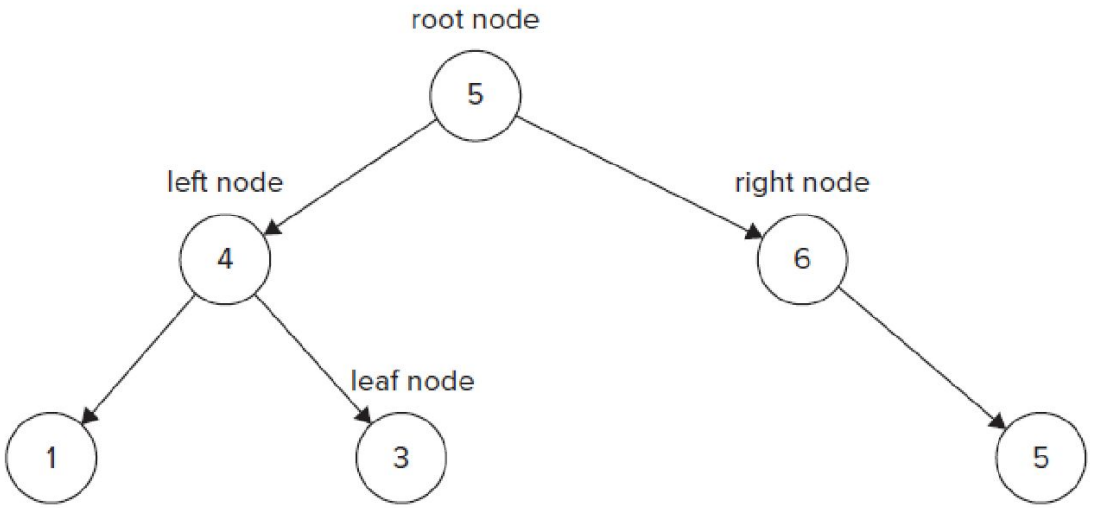
- There is **no single universal formula** for **creating an index** but most useful methods hinge on a **few common ideas**.

- The building blocks of these common ideas reside in **hash functions and B-tree and B+-tree data structures**
- A hash function is a **well-defined mathematical function to convert a large, and often variable-sized and complex, data value to a single integer** or a set of bytes

- A **tree data structure distributes a set of values in a tree-like structure**.

- **Values** are structured in a **hierarchical manner** with links or pointers between certain nodes in the tree

- A B-tree is a **generalization of a binary tree**. It allows **more than two child nodes for a parent node.**



INDEXING AND ORDERING IN MONGODB

- By default, Mongodb **creates an index on the _id property** for all collections it contains.
- To understand the **significance and impact of an index** you also need to have a few **tools to measure query performance with and without an index**
- In MongoDB, measuring **query performance is facilitated by built-in tools that explain query plans and identify slow-running queries**.

db.ratings.find();

To run explain plan for this query you can run this query:
db.ratings.find().explain();

The output of the explain plan would be something like this:

```
{
“cursor” : “BasicCursor”,
“nscanned” : 1000209,
“nscannedObjects” : 1000209,
“n” : 1000209,
“millis” : 1549,
“indexBounds” : {
}
}
```

The output says it took 1,549 milliseconds to return 1,000,209 (more than 1 million) documents. In returning these 1,000,209 documents, 1,000,209 items were examined

cursor — The cursor used to return the result sets of the query. A cursor can be of two types: **basic cursor and B-tree cursor**. Basic cursor implies table scan and B-tree cursor means an index was used.

nscanned — Number of entries scanned. When using an index, it would correspond to the number of index entries.

nscannedObjects — **Number of documents scanned.**

n — The **number of documents returned.**

millis — The time, in milliseconds, taken to run the query.
indexBounds — Represents the **minimum and maximum index keys within which the query was matched.** This fi eld is relevant only when an index is used.

```
db.movies.find({title: /Toy Story/i});
```

The output should be as follows:

```
{ “_id” : 1, “title” : “Toy Story (1995)”, “genres” : [ “Animation”, “Children’s”, “Comedy” ] }
{ “_id” : 3114, “title” : “Toy Story 2 (1999)”, “genres” : [ “Animation”, “Children’s”, “Comedy” ] }
```

db.movies.find({title: /Toy Story/i}).explain();

The output should be as follows:

```
{
“cursor” : “BasicCursor”,
“nscanned” : 3883,
“nscannedObjects” : 3883,
“n” : 2,
“millis” : 6,
“indexBounds” : {
}
}
```

Run a count, using **db.movies.count();,** on the movies collection **to verify the number of documents and you will observe that it matches with the nscanned and nscannedObjects**

To list all ratings that relate to Toy Story (more accurately Toy Story (1995)) you can query as follows:

```
db.ratings.find({movie_id: 1});
```

To see the query plan for the previous query run explain as follows:

```
db.ratings.find({movie_id: 1}).explain();
```

The output should be as follows:

```
{
"cursor" : "BasicCursor",
"nscanned" : 1000209,
"nscannedObjects" : 1000209,
"n" : 2077,
"millis" : 484, "indexBounds" : { } }
```

Query is **not running optimally** because the nscanned and nscannedObjects count reads **1,000,209, which is all the documents in the collection**

Create the index by running the following command:

```
db.ratings.ensureIndex({ movie_id:1 });
```

Then rerun the earlier query as follows:

```
db.ratings.find({movie_id: 1});
```

Verify the query plan after that as follows:

```
db.ratings.find({movie_id: 1}).explain();
```

The output should be:

```
{ “cursor” : “BtreeCursor
movie_id_1”,
“nscanned” : 2077,
“nscannedObjects” :
2077,
“n” : 2077,
“millis” : 2, “indexBounds”
: {
“movie_id” : [
[
1,
1 ]  ] } }
```

created an index on movie_id using a **descending order** sort using the following command:

```
db.ratings.ensureIndex({ movie_id:-1 });
```

Getting a list, or more accurately an array, of **all indexes** is easy. You can query as follows:

```
db.ratings.getIndexes();
```

If required, you could force a query to use a particular index using the hint method

```
db.ratings.find({ movie_id:1 }).hint({ movie_id:-1 });
```

```
db.ratings.find({ movie_id:1 }).hint({ movie_id:-1
}).explain();
```

The output of the query explain plan is as follows:

```
{
“cursor” : “BtreeCursor movie_id_-1”,
“nscanned” : 2077,
“nscannedObjects” : 2077,
“n” : 2077,
“millis” : 17,
“indexBounds” : {
“movie_id” : [
[
1,
1 ] ] }
}
```

- Although an index was used and only a select **few documents were scanned it took 17 ms to return the result set.**

- This is much **less than the 484 ms** used for the table scan but is substantially **more than the 2 ms** the ascending order index took to return the result set.

- This is possibly because in this case, the **movie_id is 1 and is at the beginning of the ascending order list** and the **results were cached from a previous query.**

- **Ascending order** indexes **do not deterministically outperform descending order indexes** when accessing documents in the beginning of the list and **vice versa**

To find the movie_id at the top

```
db.movies.find().sort({ _id:-1 });
```

Because only **one item is required**, you could simply run the query like so:

```
db.movies.find().sort({ _id:-1 }).limit(1);
```

- The **movie_id 3952 corresponds to Contender, The (2000).**

- To **get ratings for the movie** *The Contender*, you could use either the **ascending or the descending ordered** index on movie_id.

- Because the **objective here is to analyze how both of these indexes perform for an item that satisfies boundary conditions**, you can use both of them one after the other

```
db.ratings.find({ movie_id:3952 }).hint({ movie_id:1 });
db.ratings.find({ movie_id:3952 }).hint({ movie_id:1 }).explain();
```

The output of the query plan is like so:

```
{
“cursor” : “BtreeCursor movie_id_1”,
“nscanned” : 388,
“nscannedObjects” : 388,
“n” : 388,
“millis” : 2,
“indexBounds” : {
“movie_id” : [
[
3952,
3952  ]  ]  } }
```

The query and query plan commands for the descending order movie_id index is as follows:

```
db.ratings.find({ movie_id:3952 }).hint({ movie_id:-1 });
db.ratings.find({ movie_id:3952 }).hint({ movie_id:-1 }).explain();
{
“cursor” : “BtreeCursor movie_id_-1”,
“nscanned” : 388,
“nscannedObjects” : 388,
“n” : 388,
“millis” : 0,
“indexBounds” : {
“movie_id” : [
[
3952,
3952  ] ] } }
```

The theory that **values at the extremes don’t always benefit from indexes** that start out at the corresponding end seems to be **true**.

Every **run could produce a unique output.**

For example, **values could be cached and so may never hit the underlying data structures on a rerun.**

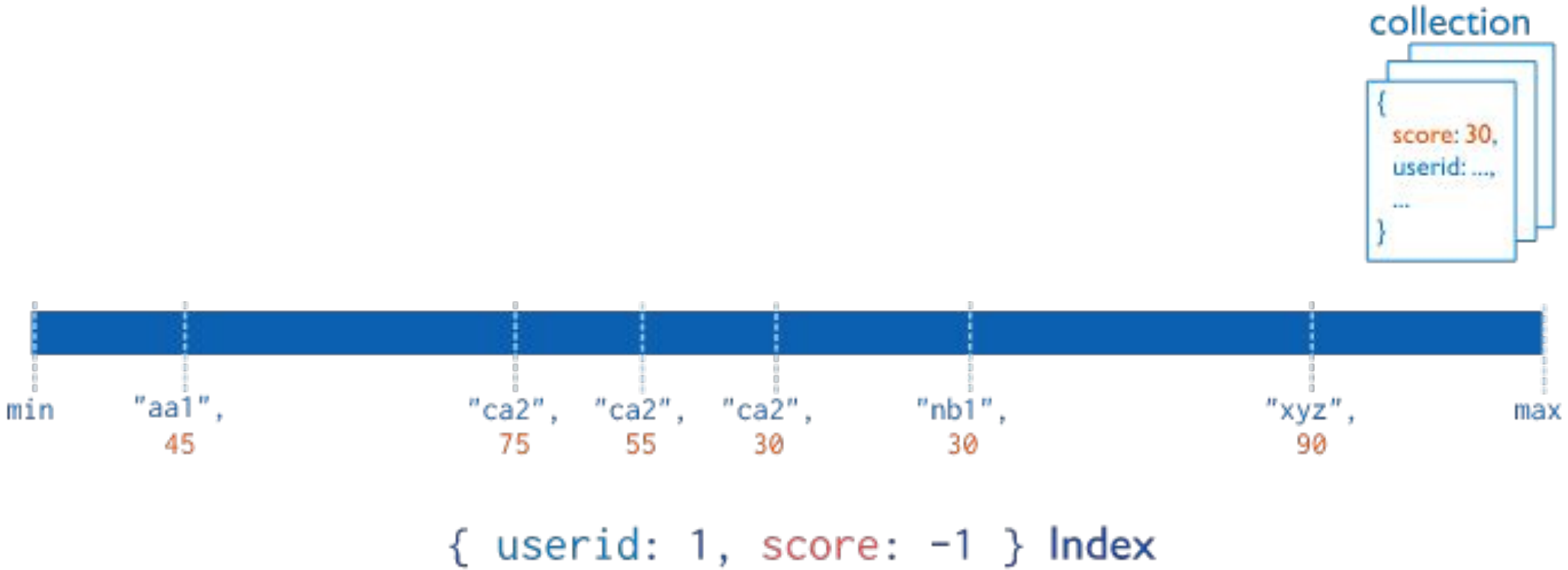
Sometimes, after **numerous modifications** to a collection it may be worthwhile to **rebuild indexes**.
To **rebuild all indexes** for the ratings collection you can run this command:
db.ratings.reIndex();

Rebuilding indexes is **not required in most cases unless the size of the collection has changed** in a considerable way or the **index seems to be occupying an unusually large amount of** disk space.

Indexes can be dropped with the dropIndex command:
db.ratings.dropIndex({ movie_id:-1 });
You can also drop all indexes if need
be. All indexes (except the one of the _id fi eld) can be dropped as follows:
db.ratings.dropIndexes();

Compound and Embedded Keys

MongoDB supports *compound indexes*, where a **single index structure holds references to multiple fields within a collection's documents.** The following diagram illustrates an example of a compound index on two fields:



```
db.ratings.ensureIndex({ movie_id:1, rating:-1 });
```

Compound keys are **not restricted to two keys**.

You can include as many keys as you like.

```
db.ratings.ensureIndex({ movie_id:1, rating:-1, user_id:1 });
```

This index can be used to query for any of these following combinations:

movie_id, rating, and user_id
movie_id and rating
movie_id

Compound indexes can also include nested (or embedded) fields.

```
{
“_id” : ObjectId(“4d0688c6851e434340b173b7”),
“name” : “joe”,
“age” : 27,
“address” : {
“city” : “palo alto”,
“state” : “ca”,
“zip” : “94303”,
“country” : “us”
} }
```

You can create an **index on the zip fi eld** of the address fi eld as follows:

db.people2.ensureIndex({ “address.zip”:1 });

Next, you can create a compound index for the name and address.zip fi elds:

db.people2.ensureIndex({ name:1, “address.zip”:1 });

You can also choose the entire sub-document as the key of an index so you can create a single index for the address fi eld:

db.people2.ensureIndex({ address:1 });

This **indexes the entire document and not just the zip fi eld** of the document.

Such an **index can be used if an entire document is passed as a query document to get a subset of the collection.**

A MongoDB collection fi eld can also contain an array instead of a document

```
{
“_id” : ObjectId(“4cccff35d3c7ab3d1941b103”),
“order_date” : “Sat Oct 30 2010 22:30:12 GMT-0700 (PDT)”,
“line_items” : [
{
“item” : {
“name” : “latte”,
“unit_price” : 4
},
“quantity” : 1
},
{
“item” : {
“name” : “cappuccino”,
“unit_price” : 4.25
},
“quantity” : 1
},
{
“item” : {
“name” : “regular”,
“unit_price” : 2
},
“quantity” : 2
}
]
}
```

Indexes — MongoDB Manual

You could index with line_items:
db.orders.ensureIndex({ line_items:1 });

db.orders.ensureIndex({ “line_items.item”:1 });
movielens

db.orders.ensureIndex({ “line_items.item.name”:1 });

So, you could query by this nested name fi eld as follows:
db.orders.find({ “line_items.item.name”:”latte” });

Creating Unique and Sparse Indexes

You could create a sparse index by explicitly specifying it as follows:

```
db.ratings.ensureIndex({ movie_id:1 }, { sparse:true });
```

```
db.movies.ensureIndex({ title:1 }, { unique:true });
```

Sparse Indexes — MongoDB Manual

Keyword-based Search and Multikeys

In some cases, though, **creating a traditional index may not be enough**, especially **when you don't want to rely on regular expressions** and need to do **a full text search**

We know that a **field that contains an array of values can be indexed**. In such instances, **MongoDB creates multikeys: one for each unique value in the array.**

Multikey Indexes — MongoDB Manual

```
{
“_id” : ObjectId(“4d06bf4c851e434340b173c3”),
“title” : “NoSQL Sessions at Silicon Valley Cloud Computing Meetup in January 2011”,
“creation_date” : “2010-12-06”,
“tags” : [
“amazon dynamo”,
“big data”,
“cassandra”,
“cloud”,
“couchdb”,
“google bigtable”,
“hbase”,
“memcached”,
“mongodb”,
“nosql”,
“redis”,
“web scale”
]
}
```

Now, you could easily **create a multikey index on the tags** field as follows:

```
db.blogposts.ensureIndex
({ tags:1 });
```

So far it’s like any other index but next you **could search by any one of the tag values like so**:

```
db.blogposts.find({ tags:”nosql” });
```

This feature can be used to build out a complete **keyword-based search**

INDEXING IN APACHE CASSANDRA

Apache Cassandra is a **hybrid between a column-oriented database and a pure key/value data store**.

Like column-oriented databases, **Cassandra supports row-key-based order and index by default. In addition, Cassandra also supports secondary indexes**.

```
use CarDataStore;
Authenticated to keyspace: CarDataStore
[default@CarDataStore] get Cars['Prius'];
=> (column=make, value=746f796f7461, timestamp=1301824068109000)
=> (column=model, value=70726975732033, timestamp=1301824129807000)
Returned 2 results.
```

The Cars column-family has two columns: make and model.

To make querying by values in the make column more efficient, create a **secondary index on the values in that column.**

Since the column already exists, modify the definition to include an index. You can update the column-family and column definition as follows:

[default@CarDataStore] **update column family Cars** with comparator=UTF8Type
... and **column_metadata=[{column_name**: **make**, validation_class: UTF8Type, **index_type: KEYS**}, ... {**column_name: model**, validation_class: UTF8Type}];

The update command **created an index on the column make**. The type of **index created is of type KEYS**.

Now, query for all values that have a make value of toyota

[default@CarDataStore] get Cars where make = ‘toyota’;

RowKey: Prius
=> (column=make, value=toyota, timestamp=1301824068109000)
⇒ (column=model, value=prius 3, timestamp=1301824129807000)

RowKey: Corolla
=> (column=make, value=toyota, timestamp=1301824154174000)
=> (column=model, value=le, timestamp=1301824173253000)
2 Rows Returned.

Try another query, but this **time filter the Cars data by model value** of prius 3 as follows:
[default@CarDataStore] get Cars where model = ‘prius 3’;
No indexed columns present in index clause with operator EQ

Try another query where you combine both make and
model as follows:

```
[default@CarDataStore] get Cars where model = ‘prius 3’ and make = ‘toyota’;
------------------
RowKey: Prius
=> (column=make, value=toyota, timestamp=1301824068109000)
=> (column=model, value=prius 3, timestamp=1301824129807000)
1 Row Returned.
```

The index **works again** because at least **one of the filter criteria has an indexed set of values**.

I. CREATE DATABASE IN MONGODB.

II. list all databases

III. create a collection by the name “Student” and insert id, student_name, Grade, Hobbies

IV. search for documents from the “Students” collection where name is Aryan

V. display only the StudName and Grade from all the documents of the Students collection. The identifier_id should be suppressed and NOT displayed.

VI. To find documents from the Students collection where the StudName begins with “M”.

VII. find documents from the Students collection where the StudName has an “e” in any position.

VIII. find the number of documents in the Students collection.

IX. sort the documents from the Students collection in the descending order of

x. Import data from a CSV fileGiven a CSV file “sample.txt” in the D:drive, import the file into the MongoDBcollection, “SampleJSON”. The collection is in the database “test”.

mongoimport --db Student --collection airlines --type csv –headerline --fi/home/hduser/Desktop/airline.csv

XI Create a collection by name “food” and add to each document add a “fruits” array

XII To find those documents from the “food” collection which has the “fruits array” constitute of “grapes”, “mango” and “apple”.

XIII To find in “fruits” array having “mango” in the first index position.

XIV Aggregate Function :
Create a collection Customers with fields custID, AcctBal, AcctType.
Now group on “custID” and compute the sum of “AccBal”.
db.Customers.aggregate ({$group : { _id : “$custID”,TotAccBal : {$sum:”$AccBal”} } });

Using NoSQL in the Cloud

This chapter **walks through the details of two NoSQL options** in the cloud: **Google's Bigtable data store and Amazon SimpleDB**

GOOGLE APP ENGINE DATA STORE

The Google App Engine (GAE) provides a **sandboxed deployment environment** for applications, which are written using **either the Python programming** language or a language that can run on a **Java Virtual Machine** (JVM).

Google provides **developers with a set of rich APIs** and an **SDK to build applications** for the app engine.

GAE Python SDK: Installation, Setup, and Getting Started

To get started you need to install Python and the GAE Python SDK.

You can download Python from python.org and the GAE Python SDK is available online at http://code.google.com/appengine/downloads.html#Google_App_Engine_SDK_for_Python

Task Manager: A Sample Application

Consider a **simple task management application** in which a user can defi ne a task, track its status, and check it as done once completed.

To define a task, the **user needs to give it a name and a description.**

Tags can be added to **categorize it and start, and expected due dates** could be specified.

Once completed, the **end date can be recorded**.

Tasks **belong to a user** and in the first version of the application they are not shared with anyone other than the owner

To **model a task**, it would be **helpful to list the properties**, specify the **data type for each property**, state whether it's required or optional, and mention whether it is single or multiple valued.

Table 10-1 lists a task's properties and its characteristics.

TABLE 10-1: Properties of a Task

PROPERTY NAME	DATA TYPE	REQUIRED	SINGLE OR MULTIPLE VALUED
`Name`	String	Yes	Single
`Description`	String	No	Single
`start_date`	Date	Yes	Single
`due_date`	Date	No	Single
`end_date`	Date	No	Single
`Tags`	array (list collection)	No	Multiple

```
import datetime
from google.appengine.ext import db
class Task(db.Model):
name = db.StringProperty()
description = db.StringProperty()
start_date = db.DateProperty()
due_date = db.DateProperty()
end_date = db.DateProperty()
tags = db.StringListProperty()
```

the Task class is **modified to specify constraints**:

```
import datetime
from google.appengine.ext import db
class Task(db.Model):
name =
db.StringProperty(required=True)
description = db.StringProperty()
start_date =
db.DateProperty(required=True)
due_date = db.DateProperty()
end_date = db.DateProperty()
tags = db.StringListProperty()
```

- The **GAE data store** can be thought of as an **object store** where each **entity is an object.**

- That means **data store entities or members could be instances of a Python class, like Task**

A key **uniquely identifies an entity** among all entities in the data store.
A key is a **combined identifi er** that includes:

- Inheritance path
- Entity kind
- Entity ID or entity key name

A specific entity of the kind, **Task, has an ID**, which can be thought of as the primary key. An ID can be either of the following:

- Application provided value, named key_name, which is a string
- System-generated (i.e., GAE data store) unique numeric ID

So, you could **create and save an entity** as follows:

```
task = Task(name = “Design task manager app”,
description = “Design the task management application.
Create the initial blueprint and the app architecture.”,
start_date = datetime.datetime.now().date())
task.put()
```

You can query for the key like so:

```
my_entity_key = task.key()
```

The output is a numeric value appended to the kind, which in this case is Task

You wanted to use **task1 as the key** for an entity of the Task kind, you could instantiate a task entity like so:

```
another_task = Task(key_name = “task1”,
name = “Yet another task”,
description = “Yet another task is, as the name says, yet another task.”,
start_date = datetime.datetime(2011, 2, 1, 12, 0, 0).date())
```

Now, querying for the key using another_task.key() returns Task: task1

Essentials of Data Modeling for GAE in Python

```
class Task(db.Model):
name = db.StringProperty(required=True)
description = db.StringProperty()
start_date =
db.DateProperty(required=True)
due_date = db.DateProperty()
end_date = db.DateProperty()
tags = db.StringListProperty()
```

The **Task model class extends the db.Model class**.

The **Model class is one of the three** built-in model classes provided in the data modeling API.

The other two classes are named **Expando and PolyModel**.

The Model class is the **most rigid** and formal of the three model classes

The **Task class**, which is a **Model type**, defines **six properties**.

Each of the **six properties** have a **well-defi ned type**, where the **type is defined using a subclass of the Property class**

A **Property class defines a data type for the property's value.**

It also defines how values are validated and stored in the data store

TABLE 10-2: Property Types and Corresponding Classes in GAE Python API

VALUE TYPE	PROPERTY CLASS	SORT ORDER	ADDITIONAL NOTES	GAE API DEFINE DATA TYPE
str, Unicode	StringProperty	Unicode	< 500 characters. str treated as ASCII for sorting.	No
db.Text	TextProperty	not orderable	long string (>500 characters).	Yes
db.ByteString	ByteStringProperty	byte order	< 500 bytes. Db.ByteString extends str and represents unencoded string of bytes.	Yes
db.Blob	BlobProperty	not orderable	Byte strings up to 1 MB.	Yes
Bool	BooleanProperty	False < True		No
int, long (64 bit)	IntegerProperty	Numeric		No
Float	FloatProperty	Numeric	If float and int together then int < float, which means 5 < 4.5.	No
datetime .datetime	DateTimeProperty, DateProperty, TimeProperty	chronological		No
List of supported value types	ListProperty, StringListProperty	If ASC, by least element If DESC, by greatest element		No
Null			Python 'None'.	No

- Google accounts — `users.User`
- Email — `db.Email`
- IM — `db.IM` (Instant Messaging ID)
- Postal address — `db.PostalAddress`
- Phone number — `db.PhoneNumber`
- Category — `db.Category`
- Link — `db.Link`
- Rating — `db.Rating`
- Geographical point — `db.GeoPt`

A data store **key is modeled** using the Key class in the google.appengine.ext.db module.

Expando

You are allowed to **add properties as required** and **the set of properties could vary between two entities of the same kind.**

Also, **two entities may choose to store a different data type for the same property.**

In order to **model such dynamic and flexible schemas,** the GAE Python API defi nes a model class **named Expando.**

Properties can be of two types:
Fixed properties
Dynamic properties

Properties **defined as attributes of a model** class are fixed properties.
Properties **added as attributes to a model** instance are dynamic properties.

```
import datetime
from google.appengine.ext import db
class Task(db.Expando):
name = db.StringProperty(required=True)
description = db.StringProperty()
start_date = db.DateProperty(required=True)
due_date = db.DateProperty()
end_date = db.DateProperty()
tags = db.StringListProperty()

t1 = Task(name="task1", start_date=datetime.datetime.now().date())
t1.description = "this is task 1"
t1.tags = ["important", "sample"]
t1.collaborator = "John Doe"

t2 = Task(name="task2", start_date=datetime.datetime.now().date())
t2.description = "this is task 2"
t2.tags = ["important", "sample"]  t2.resources = ["resource1", "resource2"]
```

PolyModel

The **PolyModel class allows you to define an inheritance hierarchy** among a **set of model classes.**

Once a **hierarchical structure is established** via class inheritance, you can **query for a class type and get qualifying entities of both the class and its subclasses in the result set**

```
from google.appengine.ext import db
from google.appengine.ext.db import polymodal

class Task(polymodel.PolyModel):
name = db.StringProperty(required=True)
description = db.StringProperty()
start_date = db.DateProperty(required=True)
due_date = db.DateProperty()
end_date = db.DateProperty()
tags = db.StringListProperty()

class IndividualTask(Task):
owner = db.StringProperty()

class TeamTask(Task):
team_name = db.StringProperty()
collaborators = db.StringListProperty()
```

Queries and Indexes

To retrieve **five Task entities with start_date of January 1, 2011** and print their names you could query like so:

```
q = db.GqlQuery("SELECT * FROM Task" + "WHERE start_date = :1",
datetime.datetime(2011, 1, 1, 12, 0, 0).date())
results = q.fetch(5)
for task in results:
print "name: %s" Task  % (task.name)
```

Alternatively, you could get the same result by querying **using the Query interface** like so:

```
q = Task.all()
q.filter("start_date =", datetime.datetime(2011, 1, 1, 12, 0, 0).date())
results = q.fetch(5)
for task in results:
print "Task name: %s" % (task.name)
```

you could **order the result set by name** as follows:

```
q = db.GqlQuery(“SELECT * FROM Task” +
“WHERE start_date = :1” +
“ORDER BY name”, datetime.datetime(2011, 1, 1, 12, 0,
0).date())
```

Every **valid query is served with the help of an underlying index**.

To put it another way, **no query can run if there is no corresponding index for it.**

Allowed Filters and Result Ordering

The app engine allows you to use the following operators on property values:

=

>

>=

<=

!

IN

You can define **multiple inequality filters on a single property** but you are **not allowed to have multiple inequality fi lters on different properties** in the **same query.**

Multiple inequality filters on a single property are split into multiple queries where the result sets are merged before returned. So a query as follows:

```
SELECT * FROM Task WHERE start_date >= :a_specified_date
AND start_date <= :another_specified_date
```

is run in **two parts**, where one part matches all rows where start_date is greater than or equal to a specifi ed date and another matches all rows where start_date is less than or equal to another start date.

Finally, the results from both queries are merged.

When **ordering queries** that **involve inequality filters** it is required that **you first order by the property on which the inequality filter operator is applied.**

You can **include other properties in the ordering only after the property on which the inequality filter is applied**

The IN operator works on **values that contain a list.**

The **IN operator evaluates membership**.

An entity is returned in the result if even one element in the list matches the filter.

For example, a_prop = [1, 2] will match both a_prop =1 and a_prop = 2.

However, a_prop = [1, 2] will not match if the query specifi es a_prop > 1 and a_prop < 2

A query and its explicitly defined index could be like so:

```
q = db.GqlQuery(“SELECT * FROM Task” +
“WHERE start_date >= :1” + “tags IN :2” + “ORDER BY start_date”,
datetime.datetime(2011, 1, 1, 12, 0, 0).date(), [“Important”, “Sample”])
```

```
indexes:
- kind: Task
properties:
- name: start_date
- name: tags
```

If you need **to traverse through the entire result s**et, you **need use the Query interface as an iterable object**.

An example could be as follows:

```
q = db.GqlQuery(“SELECT * FROM Task” +
“WHERE start_date = :1” +
“ORDER BY name”, datetime.datetime(2011, 1, 1, 12, 0, 0).date())
for task in q:
print “Task name: %s” % (task.name)
```

```
import datetime
from google.appengine.ext import db
class Task(db.Model):
name = db.StringProperty(required=True)
description = db.StringProperty()
start_date = db.DateProperty(required=True)
due_date = db.DateProperty()
end_date = db.DateProperty()
tags = db.StringListProperty()
status = db.StringProperty(choices=('in progress', 'complete', 'not started'))
def update_as_complete(key, status):
obj = db.get(key) if status == 'complete': obj.status = 'complete'
obj.end_date = datetime.datetime.now().day()
obj.put()
q = db.GqlQuery("SELECT * FROM Task" +
"WHERE name = :1", "task1")                completed_task = q.get()
db.run_in_transaction(update_as_complete, completed_task.key(), "complete")
```

The Task class from the Python example can be created as a JDO-aware plain old Java object (POJO) like so:

```
package taskmanager;
import com.google.appengine.api.datastore.Key;
import java.util.Date;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;
@PersistenceCapable
public class Task {
@PrimaryKey
@Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
private Key key;
@Persistent
private String name;
@Persistent
private String description;
@Persistent
private Date startDate;
@Persistent
private String status;
public Greeting(String name, String description, Date startDate,
String status) {  this.name = name;  this.description = description
```

```
this.startDate = startDate;
this.status = status;
}
public Key getKey() {
return key;
}
public User getName() {
return name;
}
public String getDescription() {
return description;
}
public Date getStartDate() {
return startDate;
}
public String getStatus() {
return status;
}
public void setName(String name) {
this.name = name;
}
```

```
public void setDescription(String description) {
this.description = description;
}
public void setStartDate(Date startDate) {
this.startDate = startDate;
}
public void setStatus(String status) {
this.status = status;
} }
```

```
package taskmanager;
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManagerFactory;
public final class PMF {
private static final PersistenceManagerFactory
pmfInstance =
JDOHelper.getPersistenceManagerFactory("transactions-
optional");
private PMF() {}
public static PersistenceManagerFactory get() {
return pmfInstance;
}
}
```

Finally, you can save an object as follows:

```
String name = “task1”;
String description = “a task”;
Date startDate = new Date();
String status = “task created”;
Task task = new Task(name, description, startDate,
status);
PersistenceManager pm =
PMF.get().getPersistenceManager();
try {
pm.makePersistent(task);
} finally {
pm.close();
}
```

Then you can query for all tasks using the JDO Query Language (JDOQL), which is similar to GQL, like so:

```
PersistenceManager pm = PMF.get().getPersistenceManager();
String query = “select from “ + Task.class.getName();
List<Task> tasks = (List<Task>) pm.newQuery(query).execute();
```

AMAZON SIMPLEDB

Amazon SimpleDB is a ready-to-run database alternative to the app engine data store. It's elastic and is a fully managed database in the cloud.

Getting Started with SimpleDB

Amazon SimpleDB is offered as a part of the Amazon Web Services (AWS) offerings. Getting started is as simple as setting up a SimpleDB account at http://aws.amazon.com/sdb

SimpleDB is a very simple database by design.

It imposes a few restrictions and provides a very simple API to interact with your data

The highest level of **abstraction** in SimpleDB is an **account.**

Think of it as a database instance in a traditional RDBMS setup.

Each **account can have one or more domains and each domain is a collection of items.**

Within a domain you can persist items**. Items can be of any type as long as they can be defined using attribute-value pairs.**

Therefore, **each item is a collection of attribute-value pairs.**

SimpleDB can be thought of as a document database in the cloud, expandable on demand.

It would be both easy and appropriate to store data
on SimpleDB that is in the following log fi le data format (from Chapter 3):

```
{
“ApacheLogRecord”: {
“ip”: “127.0.0.1”,
“ident” : “-”,
“http_user” : “frank”,
“time” : “10/Oct/2000:13:55:36 -0700”,
“request_line” : {
“http_method” : “GET”,
“url” : “/apache_pb.gif”,
“http_vers” : “HTTP/1.0”,
},
“http_response_code” : “200”,
“http_response_size” : “2326”,
“referrer” : “http://www.example.com/start.html”,
“user_agent” : “Mozilla/4.08 [en] (Win98; I ;Nav)”,
},
}
```

available commands in the SimpleDB API

CreateDomain — Create a domain to store your items.
DeleteDomain — Delete an existing domain.
ListDomains — List all the domains within your account.
DomainMetadata — Get information on a domain

Use the PutAttributes method to insert or update an item.

An item is a collection of attribute-value pairs.

Inserting an item implies creating the set of attribute-value pairs that logically forms an item.

BatchPutAttributes is also available to carry out multiple put operations in a single call.

DeleteAttributes allows you to delete an item, an attribute-value pair, or just an attribute-value from your domain.

BatchDeleteAttributes allows multiple delete operations in a single call.

You can get the attribute-value pairs of a single item by using the GetAttributes operation

The easiest way to test this SimpleDB API is to run the operations using a command-line client.

A Perl-based command-line client could be used.

Name of this command-line client is amazon-simpledb-cli.

You can download a copy of this client from its project page, accessible online at http://code.google.com/p/amazon-simpledb-cli/.

To install amazon-simpledb-cli, fi rst make sure you have Perl installed on your machine

To get started, fi rst make sure to get (or update)
the following Perl modules:

Getopt::Long
Pod::Usage
Digest::SHA1
Digest::HMAC
XML::Simple
Bundle::LWP
Crypt::SSLeay

You can install Getopt::Long like so:

perl -MCPAN -e 'install Getopt:Long

Once the required modules are installed and updated you can install the downloaded AWS Perl module as follows:

1.First, unzip the downloaded distribution as follows: unzip AmazonSimpleDB-*-perllibrary.zip

2. Then get the Perl sitelib like so: sitelib=$(perl -MConfig -le 'print $Config{sitelib}')

3. Finally, copy the Amazon module to the sitelib as follows: sudo scp -r AmazonSimpleDB-*-perl-library/src/Amazon $sitelib

After the AWS Perl module is installed, get the amazon-simpledb-cli script like so:
sudo curl -Lo /usr/local/bin/simpledb http://simpledb-cli.notlong.com
and set the script permissions to allow everyone to execute the script as follows:
sudo chmod +x /usr/local/bin/simpledb

The program is now all ready to be used. Next, make sure to locate the AWS credentials — the AWS access key and the AWS access secret key, which are available from your account page — and have them handy to test the amazon-simpledb-cli script

You can create a domain as follows:

```
simpledb create-domain domain1
```

You can add items to this domain as follows:

```
simpledb put domain1 item1 key1=valueA key2=value2 anotherKey=someValue
simpledb put domain1 item2 key1=valueB key2=value2 differentKey=aValue
```

Then you can edit item1 and add another attribute-value pair to it as follows:

```
simpledb put domain1 item1 yetAnotherKey=anotherValue
```

You can replace an attribute-value pair with a newer one as follows:

```
simpledb put-replace domain1 item1 key1=value1 newKey1=newValue1
```

You can delete an attribute or just the value of an attribute. Examples could be:

```
simpledb delete mydomain item1 anotherKey
simpledb delete mydomain item2 key2=value2
```

At the account level you can list all domains like so:
simpledb list-domains

You can list all item names in a domain like so:
simpledb select ‘select itemName() from domain1’

Or choose to filter the list of items using a SQL-like syntax and list all matching items and its attributes as follows:
simpledb select ‘select * from domain1 where key1=”valueA”’

If you would like to list all attributes on a specifi c item, say item1, then you could use simpledb like so:
simpledb get domain1 item1

If you would like to restrict the output to only a specifi ed set of attributes, you can pass in the attribute names to the last command as follows:
simpledb get mydomain item1 newKey1 key2

If you don’t need a domain any more and want to remove a domain and all its constituents, you can run a simpledb command like so:
simpledb delete-domain domain1

A call like:
simpledb put domain1 item1 key1=valueA key2=value2 anotherKey=someValue
is translated to:
https://sdb.amazonaws.com/
?Action=PutAttributes
&DomainName=domain1
&ItemName=item1
&Attribute.1.Name=key1
&Attribute.1.Value=valueA
&Attribute.2.Name=key2
&Attribute.2.Value=value2
&Attribute.3.Name=anotherKey
&Attribute.3.Value=someValue
&AWSAccessKeyId=[valid access key id]
&SignatureVersion=2
&SignatureMethod=HmacSHA256
&Timestamp=2011-01-29T15%3A03%3A05-07%3A00
&Version=2009-04-15
&Signature=[valid signature]

Accessing SimpleDB Using Java

simple Java program that interacts with SimpleDB using the AWS SDK

```
import java.util.ArrayList;
import java.util.List;
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.auth.PropertiesCredentials;
import com.amazonaws.services.simpledb.AmazonSimpleDB;
import com.amazonaws.services.simpledb.AmazonSimpleDBClient;
import com.amazonaws.services.simpledb.model.Attribute;
import com.amazonaws.services.simpledb.model.BatchPutAttributesRequest;
import com.amazonaws.services.simpledb.model.CreateDomainRequest;
import com.amazonaws.services.simpledb.model.Item;
import com.amazonaws.services.simpledb.model.ReplaceableAttribute;
import com.amazonaws.services.simpledb.model.ReplaceableItem;
```

```
public class SimpleDBExample {
public static void main(String[] args) throws Exception {
AmazonSimpleDB sdb = new AmazonSimpleDBClient(new PropertiesCredentials(
SimpleDBExample.class.getResourceAsStream(“aws_credentials.properties”)));
try {
String aDomain = “domain1”;
sdb.createDomain(new CreateDomainRequest(aDomain));
// Put data into a domain
sdb.batchPutAttributes(new BatchPutAttributesRequest(myDomain,
createSampleData()));
} catch (AmazonServiceException ase) {
System.out.println(“Error Message: “ + ase.getMessage());
System.out.println(“HTTP Status Code: “ + ase.getStatusCode());
System.out.println(“AWS Error Code: “ + ase.getErrorCode());
System.out.println(“Error Type: “ + ase.getErrorType());
System.out.println(“Request ID: “ + ase.getRequestId());
} catch (AmazonClientException ace) {
System.out.println(“Error Message: “ + ace.getMessage());
}
}
```

```
private static List<ReplaceableItem> createSampleData()
{
List<ReplaceableItem> myData = new
ArrayList<ReplaceableItem>();
sampleData.add(new
ReplaceableItem(“item1”).withAttributes(
new ReplaceableAttribute(“key1”, “valueA”, true),
new ReplaceableAttribute(“key2”, “value2”, true),
new ReplaceableAttribute(“anotherKey”, “someValue”,
true)
);
sampleData.add(new
ReplaceableItem(“item2”).withAttributes(
new ReplaceableAttribute(“key1”, “valueB”, true),
new ReplaceableAttribute(“key2”, “value2”, true),
new ReplaceableAttribute(“differentKey”, “aValue”, true)
)
return myData;
}
}
```

AWS SimpleDB interaction from Python. Boto, available online at http://code.google.com/p/boto/, is the most popular choice for connecting to SimpleDB from Python. To get started, download the latest source of boto from its Github mirror as follows:

```
git clone https://github.com/boto/boto.git
```

Then change into the cloned repository directory and run `python install setup.py` to install boto. Once installed, fi re up a Python interactive session and you can easily create a new domain and add items to it as follows:

```
import boto
sdb = boto.connect_sdb('<your aws access key>', '<your aws secret key'>)
domain = sdb.create_domain('domain2')
item = domain.new_item('item1')
item['key1'] = 'value1'
item['key2'] = 'value2'
item.save()
```