CSE 424
Big Data

NoSQL

Slides 3

Instructor: Asst. Prof. Dr. Hüseyin ABACI

Outline

- NoSQL - Characteristics
- NoSQL - Key-Value Databases
  - Amazon DynamoDB
- NoSQL - Document Databases
  - MongoDB
- NoSQL - Column Family Databases
  - HBase
- NoSQL - Graph Databases
  - Neo4j

Bahga, Arshdeep, and Vijay Madisetti. Big data science & analytics: A hands-on approach. VPT, 2016.

NoSQL - Characteristics

- Schema-less data model – Data can exist in its raw form.
- Scale out rather than scale up – More nodes can be added to obtain additional storage with a NoSQL database, in contrast to having to replace the existing node with a better, higher performance/capacity one.
- Highly available – This is built on cluster-based technologies that provide fault tolerance out of the box.
- Lower operational costs – Many NoSQL databases are built on Open Source platforms with no licensing costs. They can often be deployed on commodity hardware.
- Eventual consistency – Data reads across multiple nodes but may not be consistent immediately after a write. However, all nodes will eventually be in a consistent state.
- BASE, not ACID – BASE compliance requires a database to maintain high availability in the event of network/node failure, while not requiring the database to be in a consistent state whenever an update occurs. The database can be in a soft/inconsistent state until it eventually attains consistency. As a result, in consideration of the CAP theorem, NoSQL storage devices are generally AP or CP.

NoSQL – Characteristics (cont’d)

- API driven data access – Data access is generally supported via API based queries, including RESTful APIs, whereas some implementations may also provide SQL-like query capability.
- Auto sharding and replication – To support horizontal scaling and provide high availability, a NoSQL storage device automatically employs sharding and replication techniques where the dataset is partitioned horizontally and then copied to multiple nodes.

NoSQL – Characteristics (cont'd)

- Distributed query support – NoSQL storage devices maintain consistent query behavior across multiple shards.
- Polyglot persistence – The use of NoSQL storage does not mandate retiring traditional RDBMSs. In fact, both can be used at the same time, thereby supporting polyglot persistence, which is an approach of persisting data using different types of storage technologies within the same solution architecture. This is good for developing systems requiring structured as well as semi/unstructured data.
- Aggregate-focused – Unlike relational databases that are most effective with fully normalized data, NoSQL storage devices store de-normalized aggregated data (an entity containing merged, often nested, data for an object) thereby eliminating the need for joins and extensive mapping between application objects and the data stored in the database.

- **However,**
- Do not provide ACID guarantees, therefore less suitable for applications such as transaction processing (e.g. sales/banking) that require strong consistency.
- Lack of a consistent model can lead to solution lock-in, i.e., migrating from one solution to other may require significant remodeling of the application.
- Limited support for aggregation (SUM, AVG, COUNT, GROUP BY) as compared to relational databases.

NoSQL - Key-Value Databases

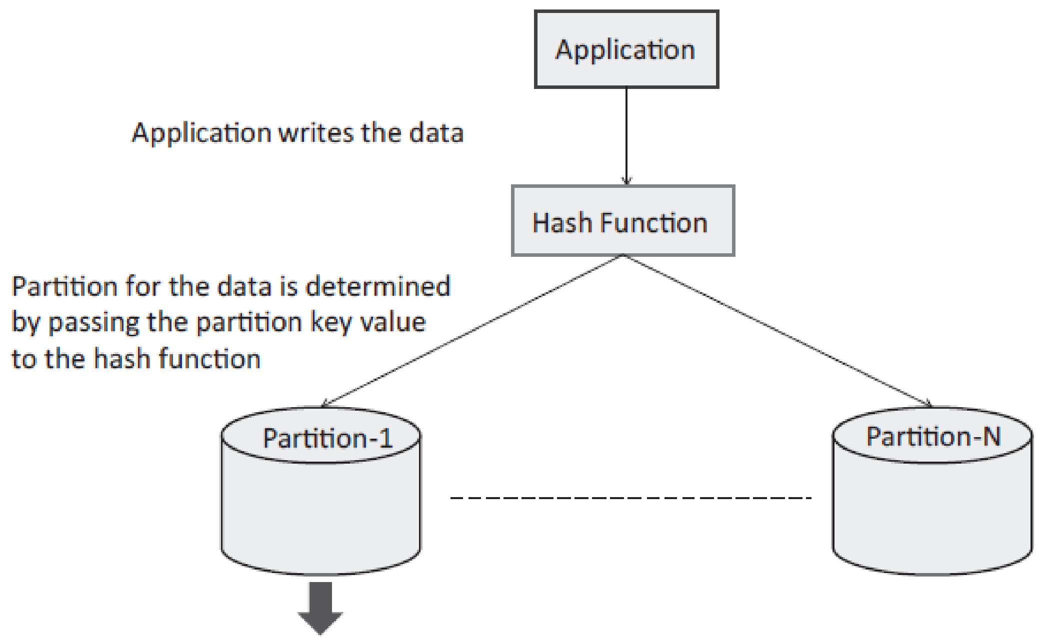
- These databases store data in the form of key-value pairs. The keys are used to identify uniquely the values stored in the database.
- The database uses the key to determine where the value should be stored. The data is partitioned across the storage nodes by the keys. For determining the partitions for the keys, hash functions are used. The partition number for a key is obtained by applying a hash function to the key. The hash functions are chosen such that the keys are evenly distributed across the partitions.
- The values can be virtually of any type (such as strings, integers, floats, binary large object (BLOB), etc.).
- Key-value databases do not have tables like in relational databases. However, some key-value databases support tables, buckets or collections to create separate namespaces for the keys.

NoSQL - Key-Value Databases (cnt'd)

- Key-value databases are suited for applications that require storing unstructured data without a fixed schema. These databases can be scaled up horizontally and can store a very large number of key-value pairs.
- Unlike relational databases which provide specialized query languages (such as SQL), the key-value databases only provide basic querying and searching capabilities. Key-value databases are suitable for applications for which the ability to store and retrieve data in a fast and efficient manner is more important than imposing structure or constraints on the data. For example, key-value databases can be used to store configuration data, user data, transient or intermediate data (such as shopping cart data), item-attributes and BLOBs (such as audio and images).

Key-Value Databases - Amazon DynamoDB

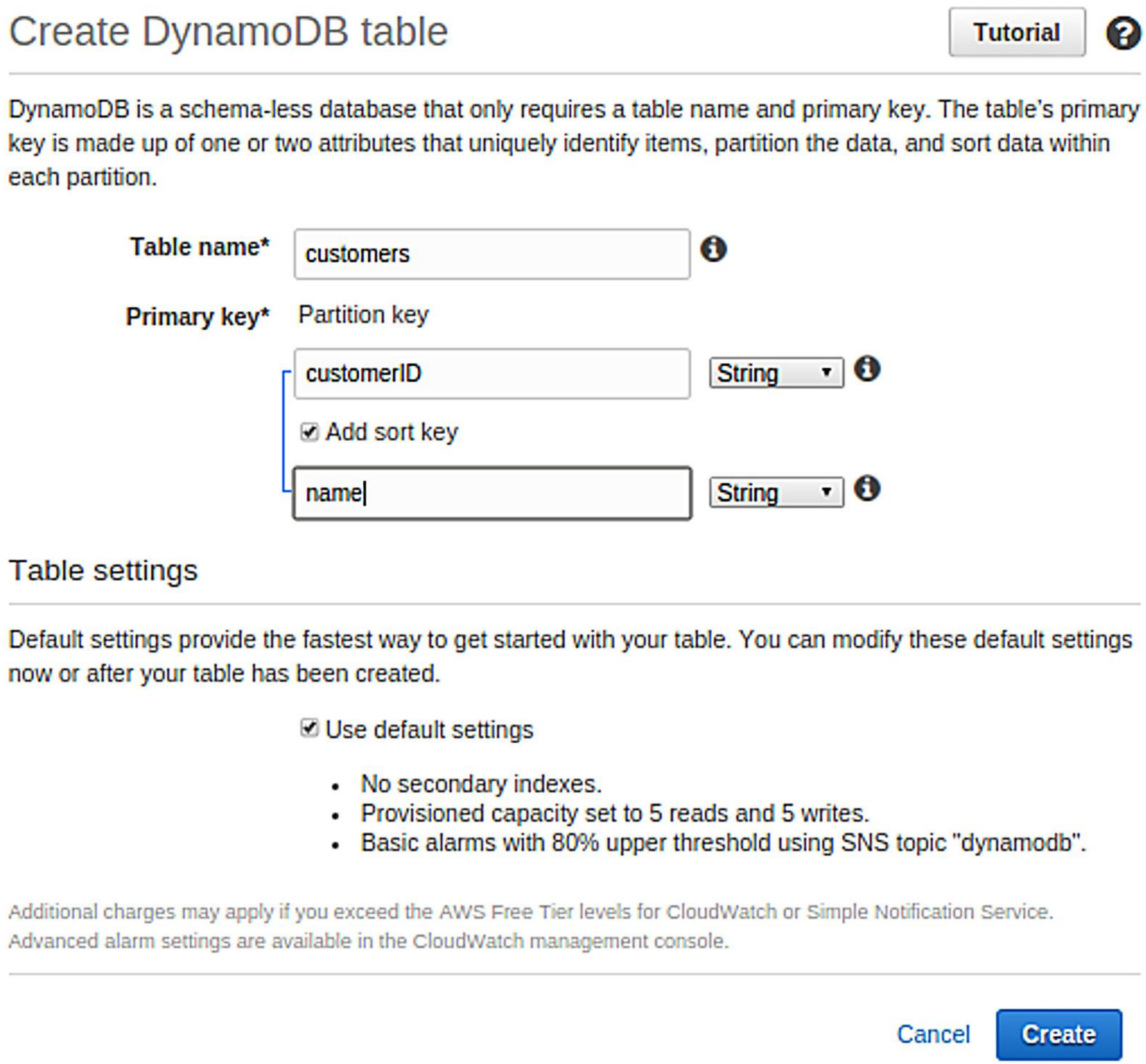
- DynamoDB's data model includes Tables, Items, and Attributes. A table is a collection of items and each item is a collection of attributes. Tables in DynamoDB do not have a fixed schema.
- The primary key is a combination of a partition key (or hash key) and an optional sort key.
- Items are composed of attributes. The attributes can be added at runtime. Different items in a table can have different attributes. Each attribute is a key-value pair.



Key	value
161305173	{name: `Ivor Merritt', address: `Ap #527-9960 Vel St.', city: `Lauw',zip: `5624',country: `Peru'}
162307206	{name: `Cade Nguyen', address: `486, 6221 Et St.', city: `Barnstaple',zip: `10903',country: `Ukraine'}

Amazon DynamoDB – Creating Table

- You can either create a table from the DynamoDB dashboard or using the DynamoDB APIs.
- Figure shows an example of creating a DynamoDB table.
- In this example, the customerID is specified as the partition key and the customer name as the sort key. We use rest of the default settings for secondary indexes, provisioned capacity and alarms.



Amazon DynamoDB – Writing the Table

- Python example of writing data to a DynamoDB table where each row of the CSV file is read one by one in a loop and the customer data is written to the DynamoDB table.
- Use `import boto.dynamodb2`

```
table=Table('customers',connection=conn)

reader = csv.reader(open('customers.csv','r'))
header=reader.next()

for row in reader:
  item = table.put_item(data={
   'customerID':row[0],
   'name':row[1],
   'address': row[2],
   'city': row[3],
   'zip': row[4],
   'country': row[5],
   'createdAt': row[6]
  },overwrite=True)
```

Amazon DynamoDB – Reading the Table

- For reading items, DynamoDB provides scan and query operations.
- The scan operation is used to retrieve all items in the table. You can specify optional filtering criteria.
- Python example of reading data from DynamoDB using scan operations.

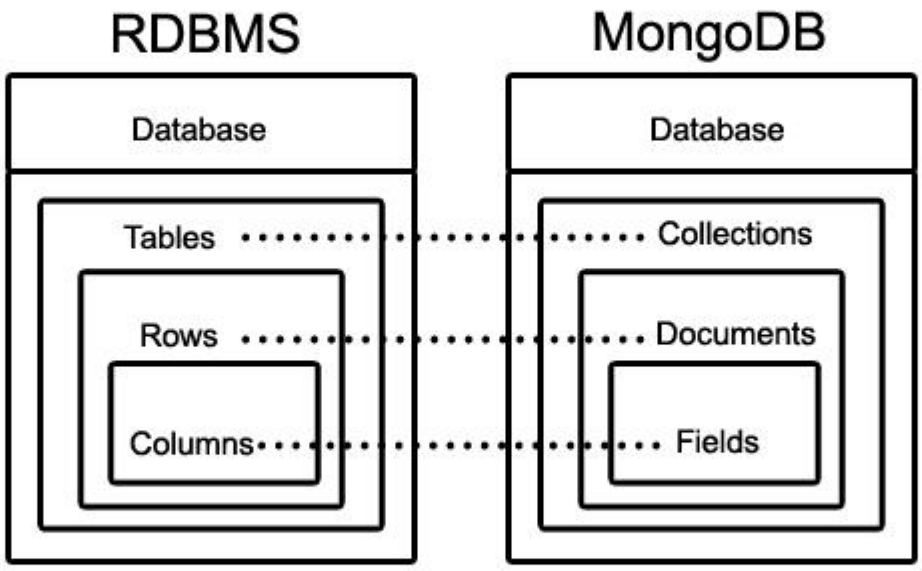
```
table=Table('customers',connection=conn)

#Scan table
result=table.scan()

for item in result:
  print item.items()

#Scan table with filter - FilterExpression
result = table.scan(country__eq='Turiye')

result = table.scan(name__beginswith='A')

for item in result:
print item.items()
```

Document Databases

- Document store databases store semi-structured data in the form of documents which are encoded in different standards such as JSON, XML, BSON.
- For example, in an eCommerce application a document can be created for each product record. Each document comprises of the data on the product features and attributes.
- Documents are organized in different ways in different document database such in the form of collections, buckets or tags.
- Each document is identified by a unique key or ID. There is no need to define any schema for the documents before storing them in the database.
- Document databases are useful for data with a varying number of fields.
- While it is possible to store JSON or XML-like documents as values in a key-value database, the benefit of using document databases over key-value databases is that these databases allow efficiently querying the documents based on the attribute values in the documents.
- Document databases do not provide the join functionality (because data is in denormalised form) provided by relational databases (all data is in normalised form). Therefore, all data that needs to be retrieved together is stored in a document.

Document Databases - MongoDB

- The basic unit of data stored by MongoDB is a document. A document includes a JSON-like set of key-value pairs.
- Documents are grouped together to form collections.
- Collections do not have a fixed schema and different documents in a collection can have different sets of key-value pairs.



https://www.educba.com/what-is-mongodb/

Document Databases - MongoDB

ID	Document

56fd4f59849f6367af489537	{
	"title" : "Motorola Moto G (3rd Generation)",
	"features" : [
	"Advanced water resistance",
	"13 MP camera",
	"5in HD display",
	"Quad core processing power",
	"5MP rear camera",
	"Great 24hr battery",
	"4G LTE Speed"
	],
	"specifications" : {
	"Color" : "Black",
	"Size" : "16 GB",
	"Dimensions" : "0.2 x 2.9 x 5.6 inches",
	"Weight" : "5.4 ounces"
	},
	"price" : 219.99
	}

56fd504d849f6367af489538	{
	"title" : "Canon EOS Rebel T5",
	"features" : [
	"18 megapixel CMOS (APS-C) sensor",
	"EF-S 18-55mm IS II standard zoom lens",
	"3-inch LCD TFT color, liquid-crystal monitor",
	"EOS 1080p full HD movie mode"
	],
	"specifications" : {
	"Color" : "Black",
	"MaximumAperture" : "f/3.5",
	"Dimensions" : "3.94 x 3.07 x 5.12 inches",
	"Weight" : "1.06 pounds"
	},
	"price" : 399
	}

Document Databases - MongoDB

Relational

Customer ID	First Name	Last Name	City
0	John	Doe	New York
1	Mark	Smith	San Francisco
2	Jay	Black	Newark
3	Meagan	White	London
4	Edward	Daniels	Boston

Phone Number	Type	DNC	Customer ID
1-212-555-1212	home	T	0
1-212-555-1213	home	T	0
1-212-555-1214	cell	F	0
1-212-777-1212	home	T	1
1-212-777-1213	cell	(null)	1
1-212-888-1212	home	F	2

MongoDB

```
{   customer_id : 1,
    first_name : "Mark",
    last_name : "Smith",
    city : "San Francisco",
    phones: [   {
          number : "1-212-777-1212",
          dnc : true,
          type : "home"
    },
    {
          number : "1-212-777-1213",
          type : "cell"
    }]
}
```

MongoDB - Writing Data (shell)

```
#Switch to new database named storedb
> use storedb
switched to db storedb

post = {
 "title" : "Motorola Moto G (3rd Generation)",
 "features" : [
  "Advanced water resistance",
  "13 MP camera which includes a color-balancing dual LED Flash",
  "5in HD display",
  "Quad core processing power",
  "5MP rear camera",
  "Great 24hr battery performance with a 2470mAh battery",
  "4G LTE Speed"
 ],
 "specifications" : {
  "Color" : "Black",
  "Size" : "16 GB",
  "Dimensions" : "0.2 x 2.9 x 5.6 inches",
  "Weight" : "5.4 ounces"
 },
 "price" : 219.99
}
> db.collection.insert(post)
WriteResult({ "Inserted" : 1 })
```

MongoDB – Getting Data (shell)

```
#Get all documents
> db.collection.find()

{ "_id" : ObjectId("56fd4f59849f6367af489537"),
"title" : "Motorola Moto G (3rd Generation)",
"features" : [ "Advanced water resistance",
"13 MP camera which includes a color-balancing dual LED Flash",
"5in HD display", "Quad core processing power", "5MP rear camera",
"Great 24hr battery performance with a 2470mAh battery", "4G LTE Speed"
],
"specifications" : { "Color" : "Black", "Size" : "16 GB",
"Dimensions" : "0.2 x 2.9 x 5.6 inches", "Weight" : "5.4 ounces" },
"price" : 219.99 }

{ "_id" : ObjectId("56fd504d849f6367af489538"),
"title" : "Canon EOS Rebel T5",
"features" : [ "18 megapixel CMOS (APS-C) sensor",
"EF-S 18-55mm IS II standard zoom lens",
"3-inch LCD TFT color, liquid-crystal monitor",
"EOS 1080p full HD movie mode" ],
"specifications" : { "Color" : "Black",
"MaximumAperture" : "f3.5", "Dimensions" : "3.94 x 3.07 x 5.12 inches",
"Weight" : "1.06 pounds" }, "price" : 399 }
```

MongoDB – Writing Data (Using pyMongo Python Library)

Assuming you have downloaded and installed MongoDB and instance is running .

```
from pymongo import MongoClient

client = MongoClient()
db = client['storedb']
collection = db['current']

item = {
 "title" : "Motorola Moto G (3rd Generation)",
 "features" : [
  "Advanced water resistance",
  "13 MP camera which includes a color-balancing dual LED Flash",
  "5in HD display",
  "Quad core processing power",
  "5MP rear camera",
  "Great 24hr battery performance with a 2470mAh battery",
  "4G LTE Speed"
 ],
 "specifications" : {
  "Color" : "Black",
  "Size" : "16 GB",
  "Dimensions" : "0.2 x 2.9 x 5.6 inches",
  "Weight" : "5.4 ounces"
 },
 "price" : 219.99
}
```

MongoDB – Writing Data (pyMongo Python Library)

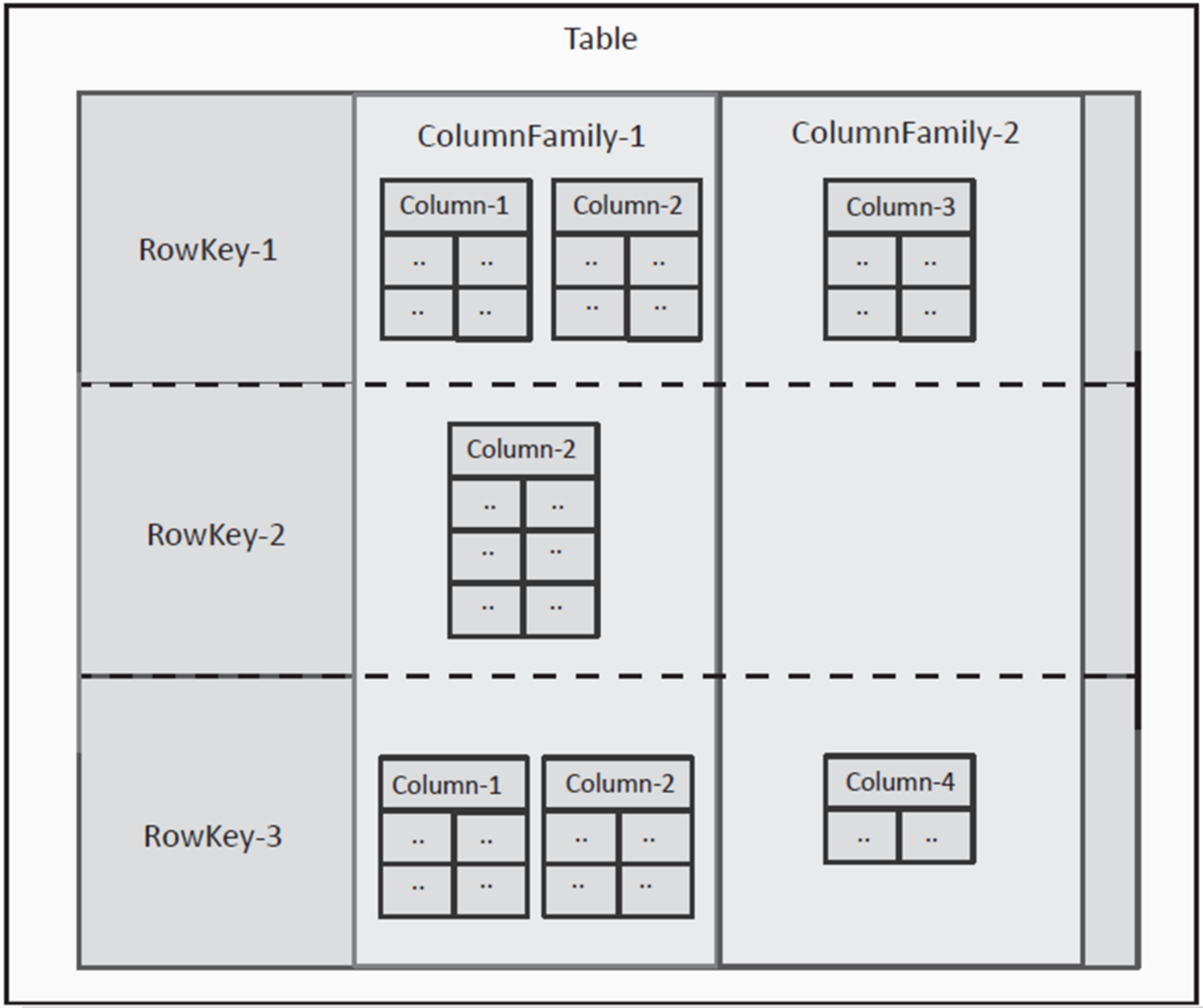
```
#Insert an item
collection.insert_one(item)

#Retrieve all items (you may want to use pprint)
results=collection.find()
for item in results:
 print item

#Find an item
results = collection.find({"title" : "Motorola Moto G"})

for item in results:
 print item
```

Column Family Databases

- In column family databases the basic unit of data storage is a column, which has a name and a value.
- A collection of columns make up a row which is identified by a row-key.
- Columns are grouped together into columns families.
- The number of columns in a column family database can vary across different rows. So data within column family databases can be sparse.
- Column family databases support high-throughput reads and writes and have distributed and highly available architectures.

Column Family Databases - HBase

- A table is consists of rows, which are indexed by the row key. Each row includes multiple column families. Each column family includes multiple columns (column qualifier) . Each column includes multiple cells or entries which are timestamped.



- HBase tables are indexed by the row key, column key and timestamp.
- HBase columns families are declared at the time of creation of the table and cannot be changed later. Columns can be added dynamically, and HBase can have millions of columns.

https://hbase.apache.org/book.html#_preface

Column Family Databases - HBase

- HBase cells cannot be over-written. Since the cells are versioned with timestamps, when newer values are added, the older values are also retained.
- No data types (no string, integer etc.) , data is stored in cells as byte arrays (binary data). The applications are responsible for correctly interpreting the data type.
- HBase stores data as key-value pairs where the keys are multi-dimensional.
- A key includes: (Table, RowKey, ColumnFamily, Column, TimeStamp) as shown in figure. For each entry/cell, multiple versions are stored, which are timestamped.

Key Length	Value Length	Row Length	Row Key	Column Family Length	Column Family	Column Qualifier	Time Stamp	Key Type	Value

Column Family Databases - HBase

Column Family

Column Qualifier(s)

Value(s)

CustID	CustomerDetails							Relatives		Accounts		
	Name	Nickname	Address	Email	Mobile	Fax	Home	Wife	Sister	Checking	Savings	Business Checking
101	Adam	A-Man	123 Main	Adam@email.com	555-555-1234	555-555-2222	555-234-5325	Debby	Kim	$1,500	$25,000	$8,250
CustID	CustomerDetails							Relatives		Accounts		
	Name	Address	Email	Mobile						Checking		
102	Bob	12 East St.	Bob@email.com	555-562-1234						$250		
CustID	CustomerDetails							Relatives		Accounts		
	Name	Nickname	Address	Email	Home			Father	Mother	Savings		
103	Christopher	Chris	504 Rogers Road	Chris@email.com	555-232-3332			Thomas	Casey	$2,000		
CustID	CustomerDetails							Relatives		Accounts		
	Name	MaidenName	Address	Email	Mobile			BusinessPartner		Businesss Checking		
104	Diana	Haines	222 Randal Road	Diana@email.com	555-844-2133			Samantha		$57,000		

https://youtu.be/4ESW0u24WYM

Column Family Databases - HBase

	Row Key	Family "Details"	Family "Relatives"	Family "Accounts"
Row	101	Name: Adam Nickname: A-Man Address: 123 Main Email: Adam@email.com Mobile: 555-555-1234 Fax: 555-555-2222 Home: 555-234-5325	Wife: Debby Sister: Kim	Checking: $1,500 Savings: $25,000 Business: $8.250
Row	102	Name: Bob Address: 12 East St. Email: Bob@email.com Mobile: 555-562-1234		Checking: $250
Row	103	Name: Christopher Nickname: Chris Address: 504 Rogers Road Email: Chris@email.com Home: 555-232-3332	Father: Thomas Mother: Casey	Savings: $2,000

https://youtu.be/4ESW0u24WYM

Column Family Databases - HBase

- HBase supports the following operations:
  - Get: Get operation is used to return values for a given row key.
  - Scan: Scan operation returns values for a range of row keys.
  - Put: Put operation is used to add a new entry.
  - Delete: Delete operation adds a special marker called Tombstone to an entry. Entries marked with Tombstones are removed during the compaction process.

HBase Usage Examples – Command Line

- HBase comes with an interactive shell from where the users can perform various Hbase operations.
- The HBase shell can be launched as follows:

```
# Launch HBase Shell
$ hbase shell
hbase(main):001:0>
```

- To create a table, the **create** command is used as shown below. The table name and column families are specified while creating a table. In this example, we created a table named products having two column families named details and sale.

```
# Create HBase table
hbase(main):002:0> create 'products', 'details', 'sale'
=> Hbase::Table - products
```

- The **list** command can be used to list all the tables in HBase, as shown below:

```
#List HBase tables
hbase(main):003:0> list
TABLE
products
1 row(s) in 1.7470 seconds
=> ["products"]
```

HBase Usage Examples – Command Line

- To write data to HBase, the **put** command can be used. The box below shows an example of writing to the products table. For row with row keys row-1 and row-2 data is written to the column family details and column (name).

Table ("products")

row ("row-1")

```
> put 'products', 'row-1', 'details:name', 'Cloud Book'
> put 'products', 'row-2', 'details:name', 'IoT Book'
```

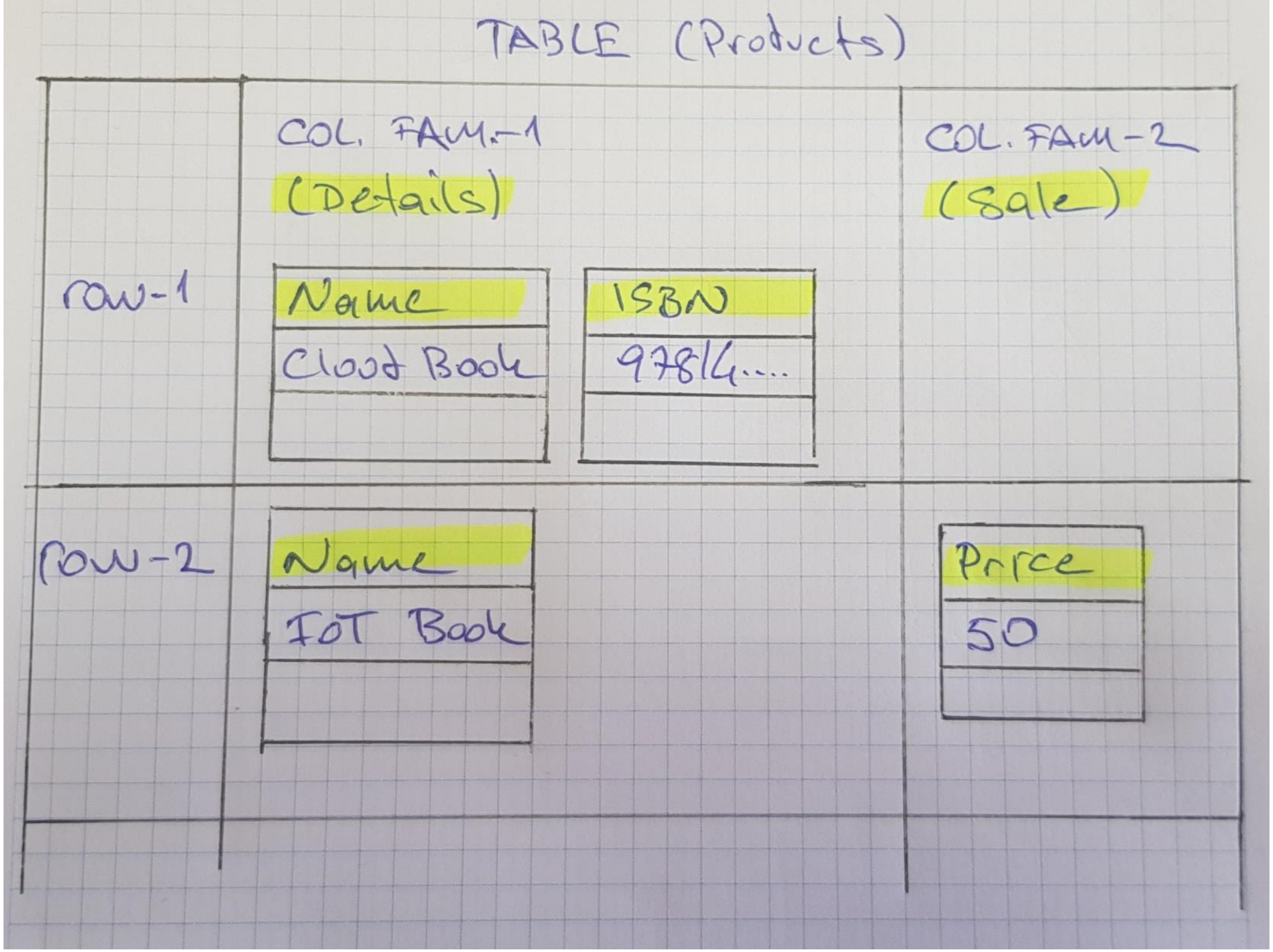
column "name" in "row-1" and column family "details"

value (cell) in column "name" (row-1 / details)

- Columns can be added dynamically. The box below shows examples of adding new columns to the rows previously created.

```
> put 'products', 'row-1', 'details:ISBN', '9781494435141'
> put 'products', 'row-2', 'sale:Price', '50'
```

HBase Usage Examples – Command Line



HBase Usage Examples – Command Line

- For reading data, HBase provides **get** and **scan** operations. The box below shows an example of reading the row with row-key row-1.

```
hbase(main):027:0> get 'products', 'row-1'
COLUMN CELL
details:name timestamp=1434772884378, value=Cloud Book
details:ISBN timestamp=1434772890556, value=9781494435141
```

- The results of the **get** operation show two cells in row-1. The values are timestamped, and multiple versions can be stored for a cell.

- The box below shows an example of a scan operation which returns all rows in a table.

```
> scan 'products'
ROW COLUMN+CELL
row-1 column=details:name, timestamp=143477288437８, value=Cloud Book
row-2 column=details:name, timestamp=143477292367８, value=IoT Book
2 row(s) in 0.0210 seconds
```

Assuming you have a running thrift server, HBase and the table “product” is already crated.

```
# Start thrift server first:

import happybase

connection = happybase.Connection(host='localhost')
table = connection.table('products')

# Put
table.put('row-1', 'details:name': 'Cloud Book')

# Get
row = table.row('row-1')
print row['details:name']

# Scan
for key, data in table.scan():
print key, data

# Delete row
row = table.delete('row-1')
```

HBase – Python Examples (Using “happybase” Python Library)

BATCH:

The Table.put() and Table.delete() methods both issue a command to the HBase Thrift server immediately. This means that using these methods is not very efficient when storing or deleting multiple values. It is much more efficient to aggregate a bunch of commands and send them to the server in one go. This is exactly what the Batch class, created using Table.batch(), does. A Batch instance has put and delete methods, just like the Table class, but the changes are sent to the server in a single round-trip using Batch.send() .

HBase – Python Examples (Using "happybase" Python Library and using efficient batch() method)

```
# Batch put
b = table.batch()

b.put('row-key-1', 'details:name': 'Cloud Book',
'details:ISBN': '9781494435141',
'sale:StartSale': '01-01-2014', 'sale:Price':'50')

b.put('row-key-2', 'details:name': 'IoT Book',
'details:ISBN': '9780996025515',
'sale:StartSale': '01-01-2015', 'sale:Price':'55')

b.send()
```

Graph Databases

- Graph stores are NoSQL databases designed for storing data that has graph structure with nodes and edges.
- The graph databases model data in the form of nodes and relationships. Nodes represent the entities in the data model. Nodes have a set of attributes. A node can represent different types of entities, for example, a person, place (such as a city, restaurant or a building) or an object (such as a car).
- The relationships between the entities are represented in the form of links between the nodes. Links also have a set of attributes. Links can be directed or undirected.
- Directed links denote that the relationship is unidirectional. For example, for two entities author and book, a unidirectional relationship (directed) called 'writes' exists between them, such that an author writes a book.
- Whereas for two friends, say A and B, the friendship relationship between A and B is bidirectional (undirected).

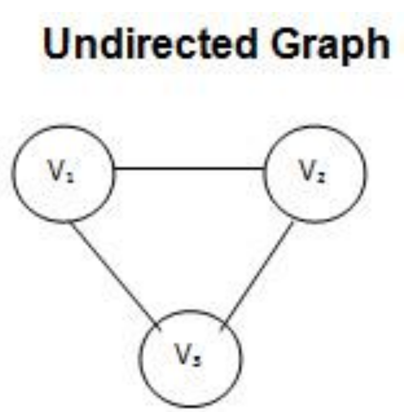


Figure 1: An Undirected Graph

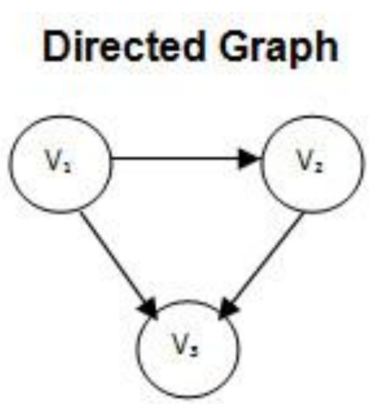


Figure 2: A Directed Graph

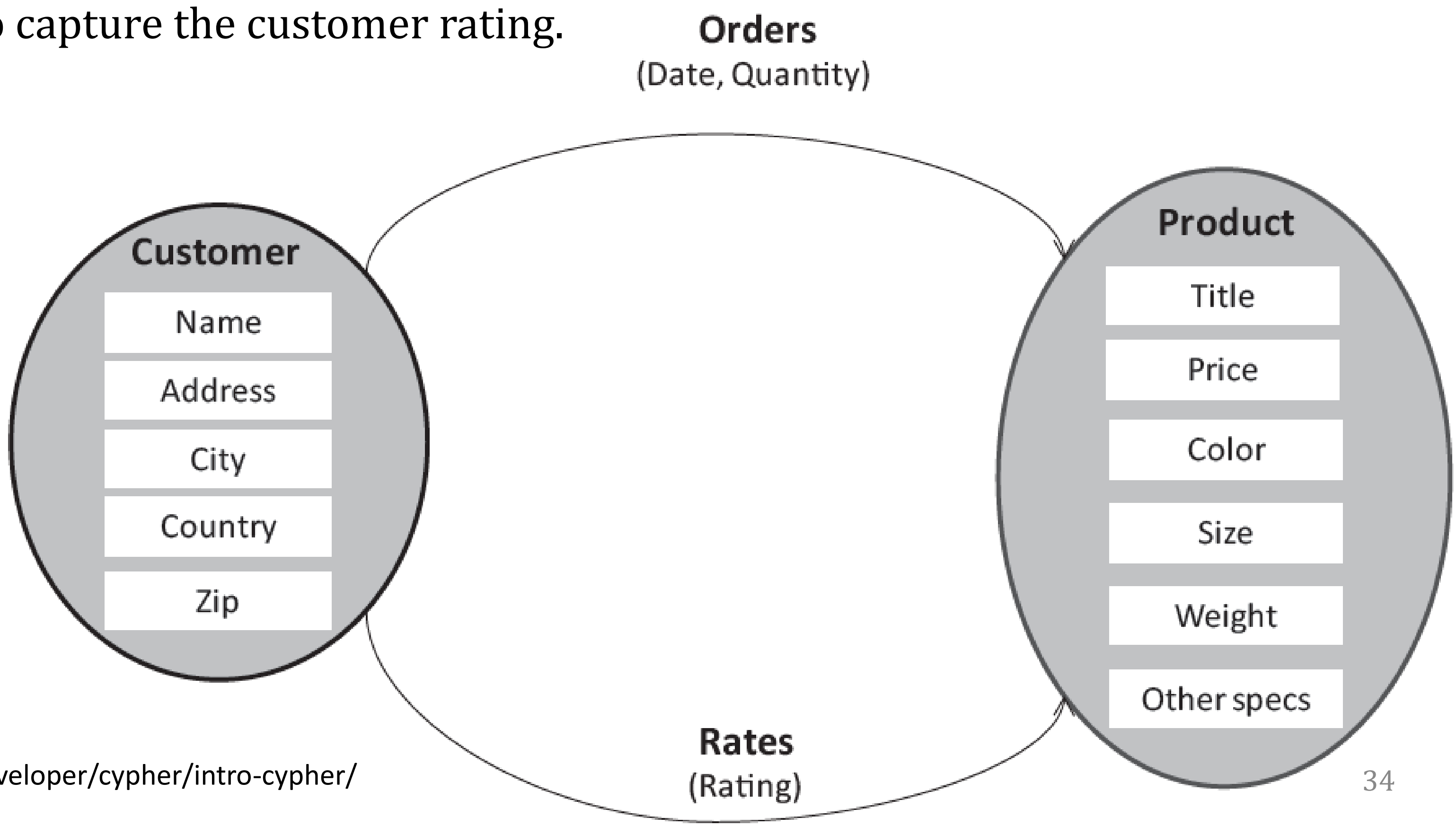
Graph Databases (cont'd)

- Querying for related entities in graph databases is much simpler and faster than relational databases as the complex join operations are avoided.

- Graph databases are suitable for applications in which the primary focus is on querying for relationships between entities and analysing the relationships (i.e social media, financial, networking).

Graph Databases - Neo4j

- In this graph, we have two types of nodes: Customer and Product. The Customer nodes have attributes such as customer name, address, city, country and zip code. The Product nodes have attributes such as product title, price and various other product-specific properties (such as color, size, weight, etc.). There are two types of relationships between the customer and product nodes: Orders or Rates. The Order relationship between a customer and product has properties such as the order date and quantity. The Rates relationship between a customer and product has a single property to capture the customer rating.



https://neo4j.com/developer/cypher/intro-cypher/

Neo4j - Cypher Constructs

- For create, read, update and delete (CRUD) operations, Neo4j provides a query language called Cypher. Cypher has some similarities with the SQL query language used for relational databases. Figure describes the usage of some of the commonly used Cypher constructs.

Creating a node

CREATE (n:LABEL {property:value}) RETURN n

Label assigned to node

n is the variable which captures the result

Node properties in the form of key-value pairs

Creating a relationship

CREATE (node1)-[: RELATIONSHIP]->(node2)

Label assigned to relationship

Neo4j - Cypher Constructs (cont'd)

Querying for a node

MATCH (node) RETURN node.property

Node to query for

Node properties to be returned

Querying for a relationship

MATCH (node1)-[:RELATIONSHIP]->(node2) RETURN node1, node2

Relationship label to query for

Variables which capture the nodes with the relationship being queried

Graph Databases - Neo4j

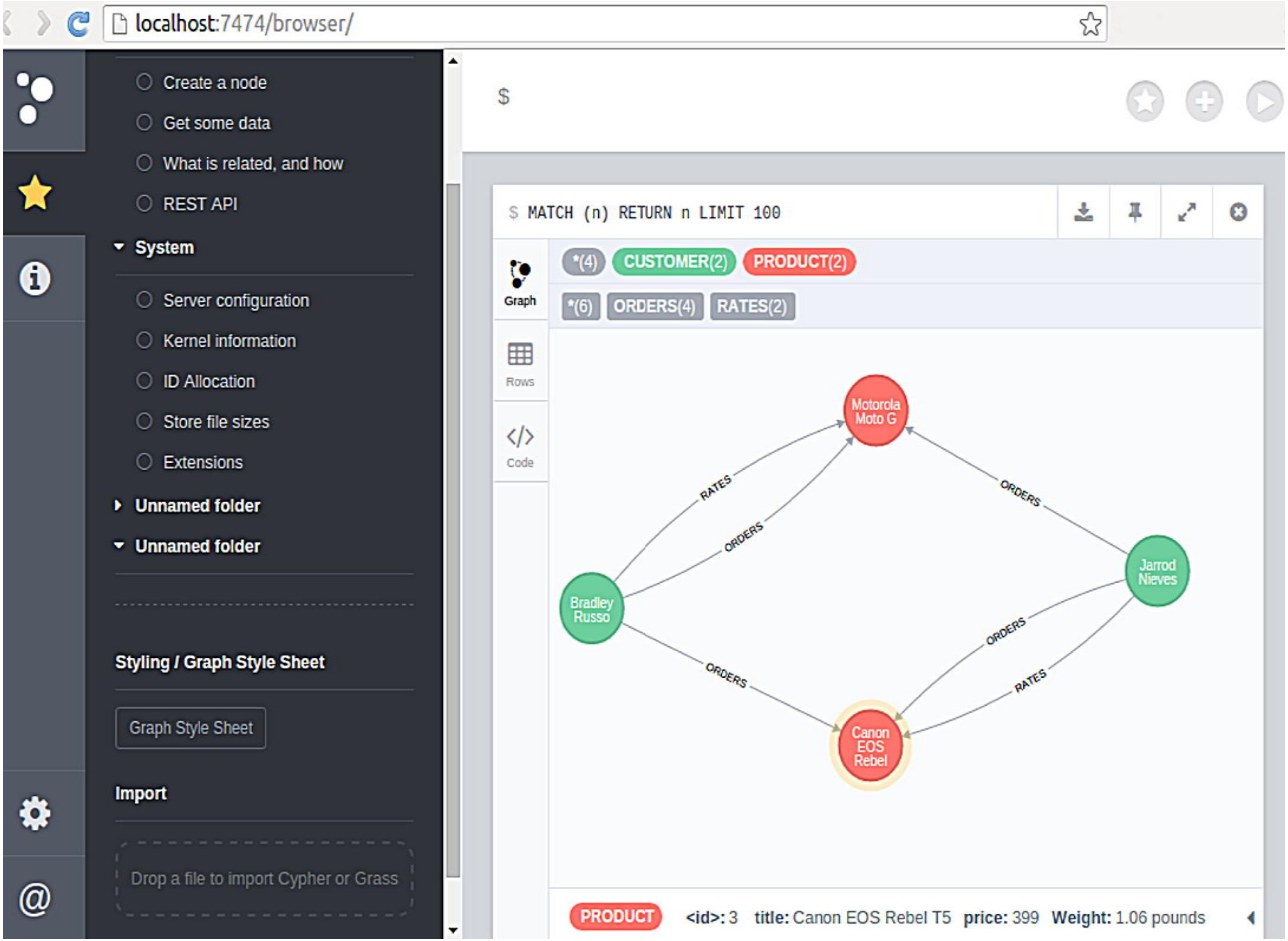
```
#Create customer
CREATE (c:CUSTOMER {name: "Bradley Russo",
  address:"P.O. Box 486, 6221 Et St.,Barnstaple",
  country:"Ukraine", zipcode:"10903"});

#Create product
CREATE (p:PRODUCT {title : "Motorola Moto G",
  Color : "Black", Size : "16 GB",
  Weight : "5.4 ounces", price : 219.99 });

#Return all data
MATCH (n) RETURN n;

#Query for a customer
MATCH (n:CUSTOMER {name: "Bradley Russo"}) RETURN n;

#Query for a product
MATCH (n:PRODUCT) WHERE n.price>200 RETURN n;

#Create relationship between customer and product
MATCH(c:CUSTOMER{name:"Bradley Russo"}),
 (p:PRODUCT{title:"Motorola Moto G"}) WITH c, p
 CREATE (c)-[:RATES]->(p);
```

- Code examples of using Cypher for creating customer and product nodes and the relationships between the nodes.

Graph Databases - Neo4j

- Neo4j also exposes a variety of REST APIs for performing the CRUD operations. These REST APIs enabled the development of language-specific client libraries for Neo4j.
- One of them is the Python Py2neo client library for Neo4j.

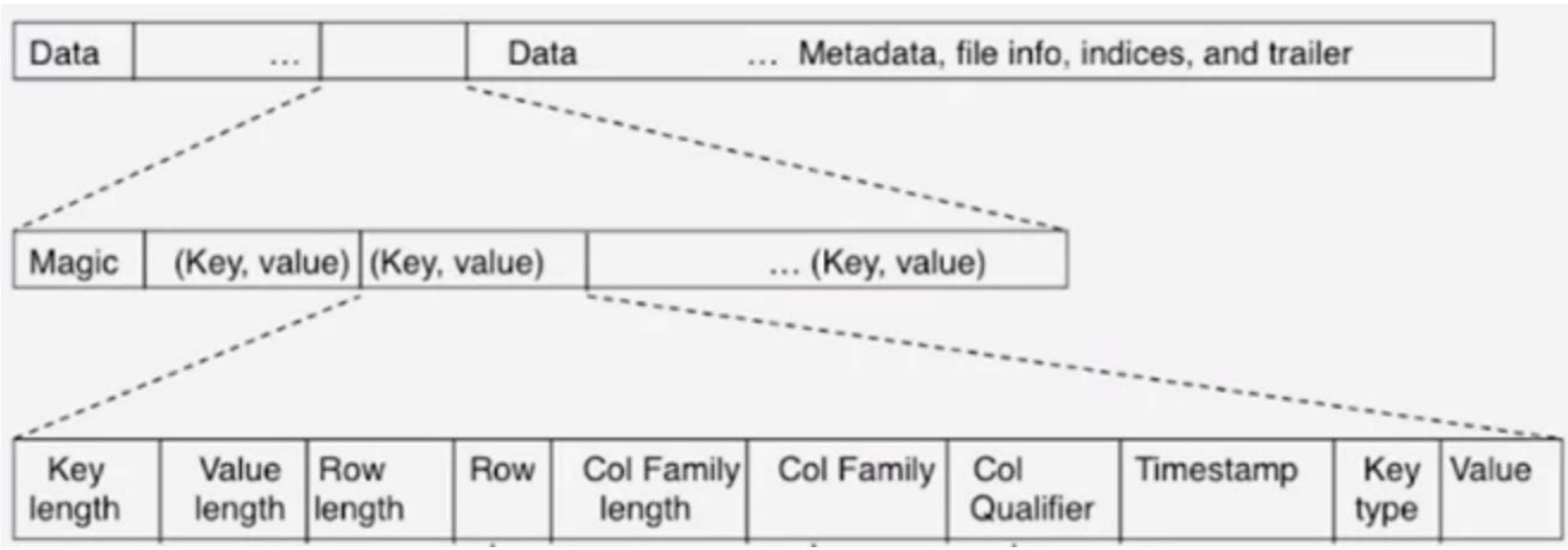
- Neo4j provides a web interface from where you can execute Cypher statements and view the graphs in the database that created using Python program.


localhost:7474/browser/
Create a node
Get some data
What is related, and how
REST API
System
Server configuration
Kernel information
ID Allocation
Store file sizes
Extensions
Unnamed folder
Unnamed folder
Styling / Graph Style Sheet
Graph Style Sheet
Import
Drop a file to import Cypher or Grass
$
$ MATCH (n) RETURN n LIMIT 100
Graph
Rows
Code
*(4)
CUSTOMER(2)
PRODUCT(2)
*(6)
ORDERS(4)
RATES(2)
Motorola Moto G
Bradley Russo
Jarrod Nieves
Canon EOS Rebel
RATES
ORDERS
ORDERS
ORDERS
ORDERS
RATES
PRODUCT
<id>: 3 title: Canon EOS Rebel T5 price: 399 Weight: 1.06 pounds

Comparison of NoSQL databases

	Key-Value DB	Document DB	Column Family DB	Graph DB
Data model	Key-value pairs uniquely identified by keys	Documents (having key-value pairs) uniquely identified by document IDs	Columns having names and values, grouped into column families	Graphs comprising of nodes and relationships
Querying	Query items by key, Database specific APIs	Query documents by document-ID, Database specific APIs	Query rows by key, Database specific APIs	Graph query language such as Cypher, Database specific APIs
Use	Applications involving frequent small reads and writes with simple data models	Applications involving data in the form of documents encoded in formats such as JSON or XML, documents can have varying number of attributes	Applications involving large volumes of data, high throughput reads and writes, high availability requirements	Applications involving data on entities and relationships between the entities, spatial data
Examples	DynamoDB, Cassandra	MongoDB, CouchDB	HBase, Google BigTable	Neo4j, AllegroGraph

Backup Slides
Column Family Databases - HBase



https://www.coursera.org/lecture/cloud-computing/1-5-hbase-lZNdW