Università degli Studi di Roma “Tor Vergata”
Dipartimento di Ingegneria Civile e Ingegneria Informatica

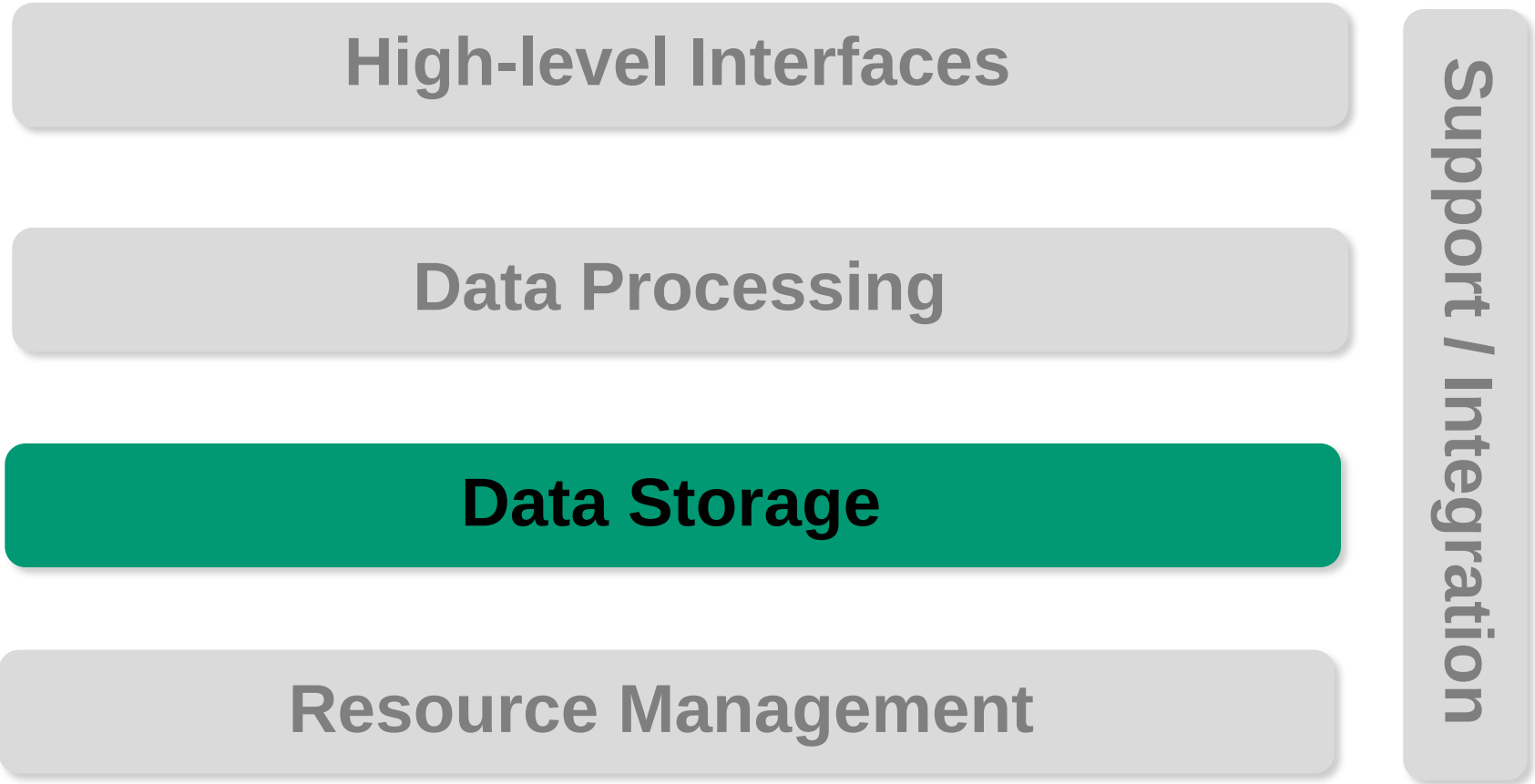
NoSQL: HBase and Neo4j

A.A. 2017/18

Matteo Nardelli

Laurea Magistrale in
Ingegneria Informatica - II anno

The reference Big Data stack



Column-family data model

- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
- Similar to a key/value store, but the value can have multiple attributes (*columns*)
- Data model: a two-level map structure:
  - A set of <row-key, aggregate> pairs
  - Each aggregate is a group of pairs <column-key, value>
  - Column: a set of data values of a particular type
- Structure of the aggregate visible
- Columns can be organized in families
  - Data usually accessed together

Suitable use cases for column-family stores

- Queries that involve only a few columns
- Aggregation queries against vast amounts of data
  - E.g., average age of all of your users
- Column-wise compression
- Well-suited for OLAP-like workloads (e.g., data warehouses) which typically involve highly complex queries over all data (possibly petabytes)

HBase

- Apache HBase:
  - open-source implementation providing Bigtable-like capabilities on top of Hadoop and HDFS
  - CP system (in the CAP space)
- Data Model
  - HBase is based on Google's Bigtable model
  - A table store rows, sorted in alphanumerical order
  - A row consists of a set of columns
  - Columns are grouped in column families
  - A table defines a priori its column families (but not the columns within the families)

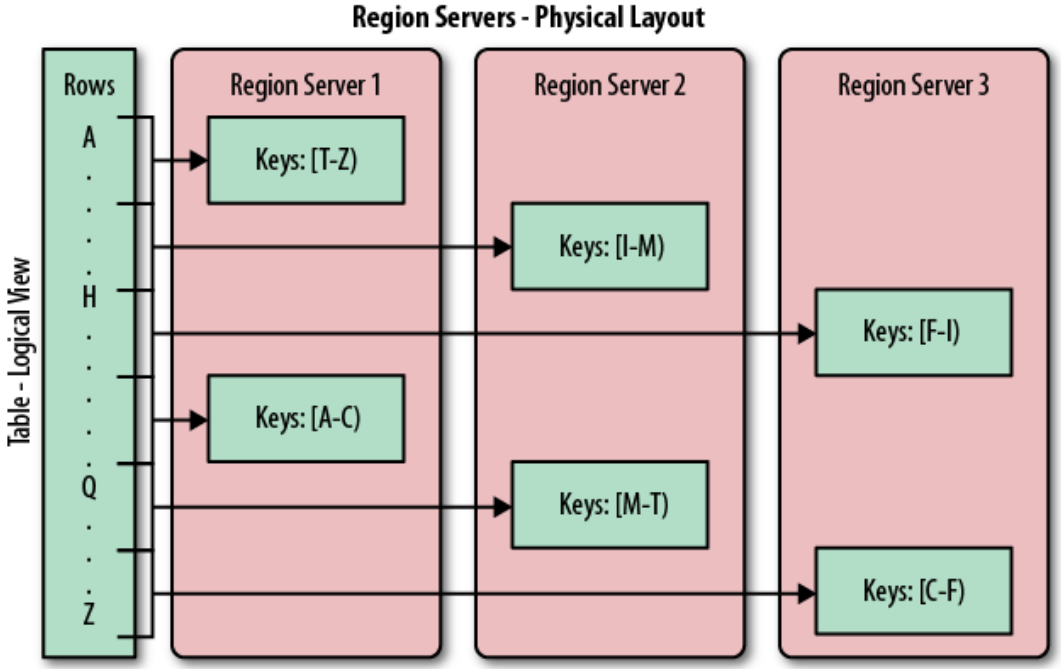
Row key	Column key	Timestamp	Cell value
cutting	info:state	1273516197868	IT
parser	role:Hadoop	1273616297466	g91m

(`info` and `role` are column families)

HBase: Auto-sharding

Region:

- the basic unit of scalability and load balancing
- similar to the tablet in Bigtable
- a contiguous range of rows stored together
- each region is served by exactly one region server
- they are dynamically split by the system when they become too large

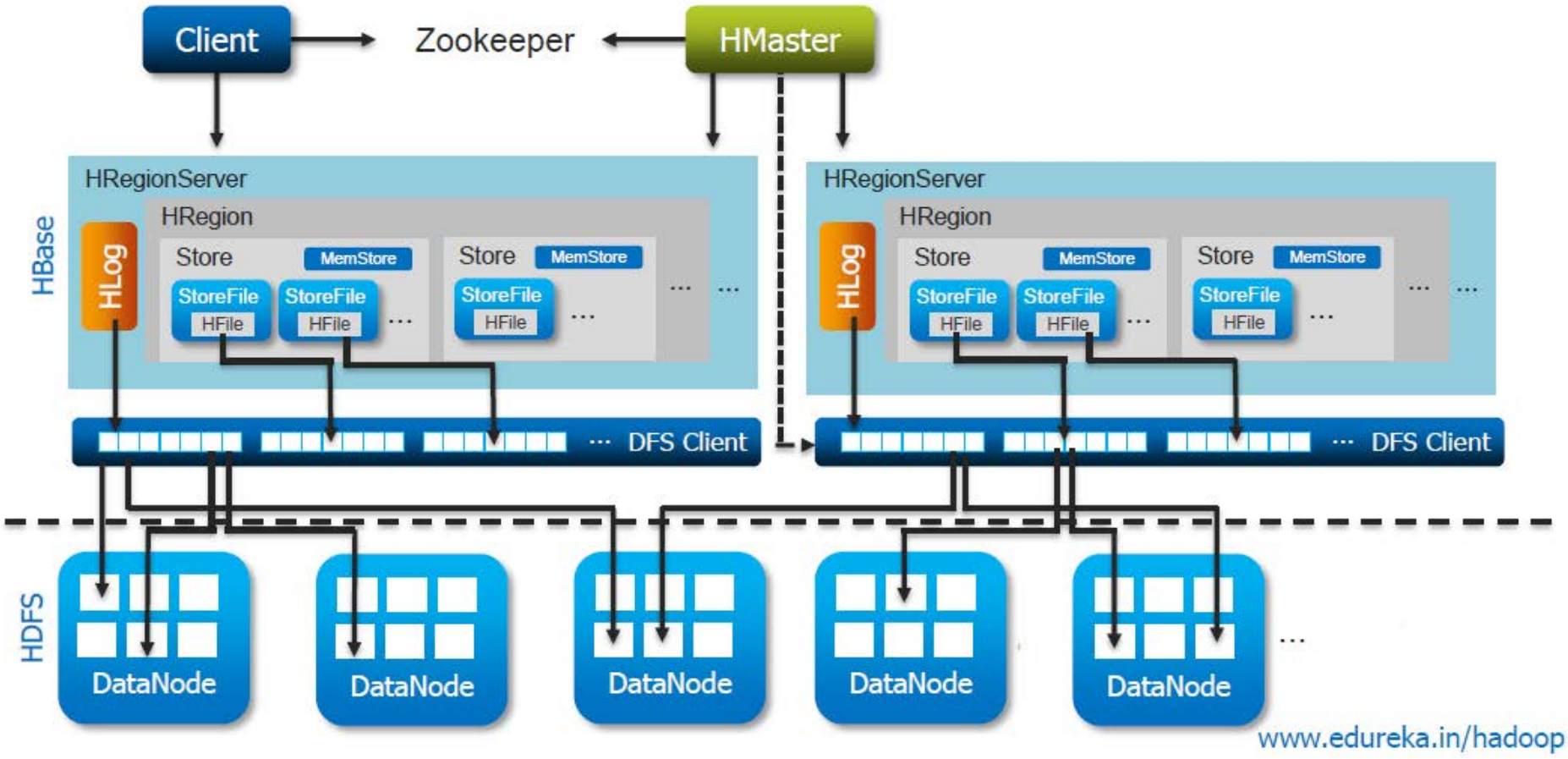


HBase: Architecture

Three major components:

- the client library
- one master server
  - The master is responsible for assigning regions to region servers and uses Apache ZooKeeper to facilitate that task
- many region servers
  - manage the persistence of data
  - region servers can be added or removed while the system is up and running to accommodate changing workloads

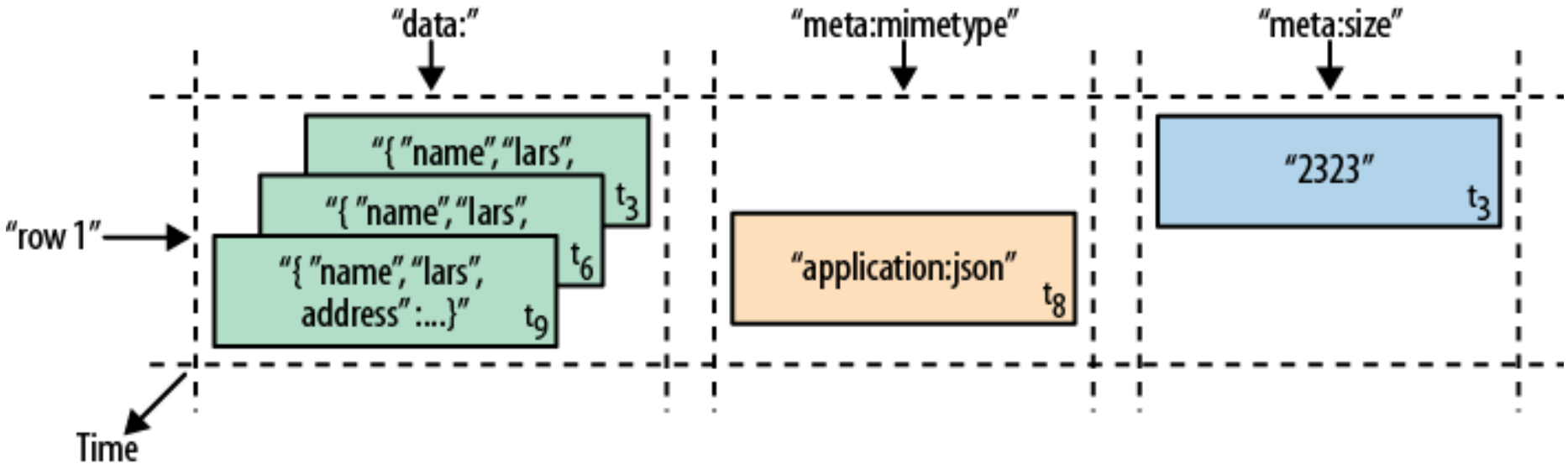
HBase: Architecture



HBase: Versioning

- Cells may exist in multiple versions, and different columns have been written at different times.

By default, the API provides a coherent view of all columns wherein it automatically picks the most current value of each cell.



HBase: Strengths

- The column-oriented architecture allows for huge, wide, sparse tables as storing NULLs is free.
- Highly scalable due to the flexible schema and row-level atomicity
- Since a row is served by exactly one server, HBase is strongly consistent, and using its multi-versioning can help you to avoid edit conflicts
- The storage format is ideal for reading adjacent key/value pairs
- Table scans run in linear time and row key lookups or mutations are performed in logarithmic order
- Bigtable has been in use for a variety of different use cases from batch-oriented processing to real-time data-serving

Hands-on HBase
(Docker image)

HBase with Dockers

- We use a lightweight container with a standalone HBase

```
$ docker pull harisekhon/hbase
```

- We can now create an instance of HBase; since we are interesting to use it from our local machine, we need to forward several HBase ports and update the hosts file;

```
$ docker run -ti --name=hbase-docker -h hbase-docker -
p 2181:2181 -p 8080:8080 -p 8085:8085 -p 9090:9090 -p
9095:9095 -p 16000:16000 -p 16010:16010 -p 16201:16201
-p 16301:16301 harisekhon/hbase
```

```
# append the following line to /etc/hosts
127.0.0.1    hbase-docker
```

HBase Client

- We interact with HBase through its Java APIs
- Using Maven, include the hbase-client dependency:

```
<dependency>
	<groupId>org.apache.hbase</groupId>
	<artifactId>hbase-client</artifactId>
	<version>1.3.0</version>
</dependency>
```

HBase Client

```
public Connection getConnection() throws ... {

   Configuration conf = HBaseConfiguration.create();
   conf.set("hbase.zookeeper.quorum", ZOOKEEPER_HOST);
   conf.set("hbase.zookeeper.property.clientPort",
                                          ZOOKEEPER_PORT);
   conf.set("hbase.master", HBASE_MASTER);

   /* Check configuration */
   HBaseAdmin.checkHBaseAvailable(conf);

   Connection connection =
             connectionFactory.createConnection(conf);
   return connection;

}
```

This is only an excerpt, check the HBaseClient.java file

HBase Client: Create Table

```
public void createTable(String table,
                        String... columnFamilies) {

      Admin admin = ...
      HTableDescriptor tableDescriptor = ... table ...

      for (String columnFamily : columnFamilies) {
            tableDescriptor.addFamily(columnFamily);
      }

      admin.createTable(tableDescriptor);

}
```

This is only an excerpt, check the HBaseClient.java file

HBase Client: Drop Table

```
public void dropTable(String table) {

        Admin admin = ...
        TableName tableName = ... table ...

        // To delete a table or change its settings,
        // you need to first disable the table
        admin.disableTable(tableName);

        admin.deleteTable(tableName);

}
```

This is only an excerpt, check the HBaseClient.java file

HBase Client: Put Data

```
public void put(String table, String rowKey,
                String columnFamily,
                String column, String value) {

   Table hTable =
             getConnection().getTable( ... table ... );

   Put p = new Put(b(rowKey));
   p.addColumn(b(columnFamily), b(column), b(value));

   // Saving the put Instance to the HTable
   hTable.put(p);

   hTable.close();
}
```

This is only an excerpt, check the HBaseClient.java file

HBase Client: Get Data

```
public String get(String table, String rowKey,
                  String columnFamily,
                  String column) {

   Table hTable =
             getConnection().getTable( ... table ... );

   Get g = new Get(b(rowKey));
   g.addColumn(b(columnFamily), b(column));

   Result result = hTable.get(g);

   return Bytes.toString(result.getValue());

}
```

This is only an excerpt, check the HBaseClient.java file

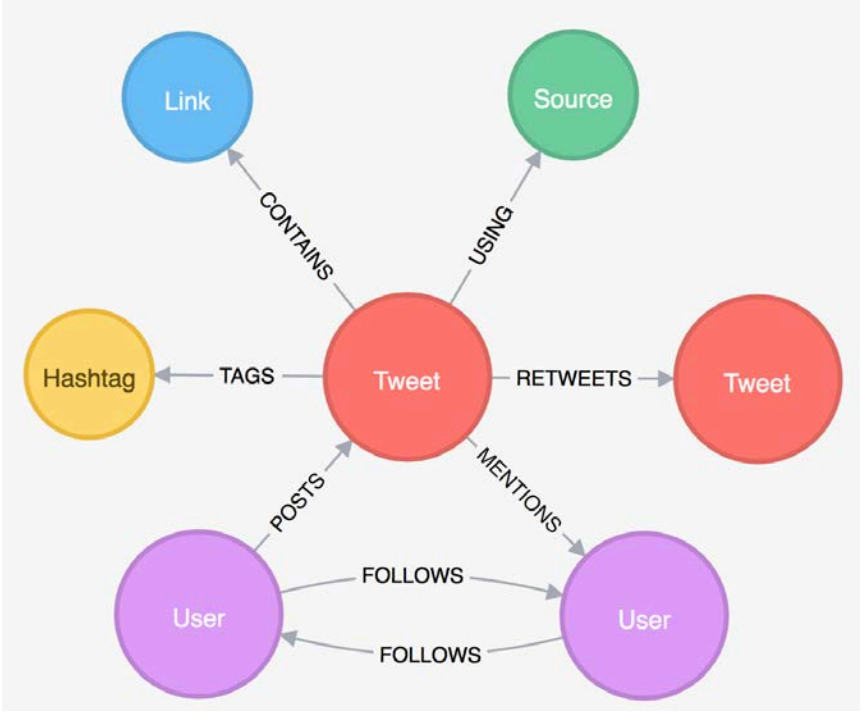
HBase Client: Delete Data

```
public void delete(String table, String rowKey) {

    Table hTable =
              getConnection().getTable( ... table ... );

    Delete delete = new Delete(b(rowKey));

    // deleting the data
    hTable.delete(delete);

    // closing the HTable object
    hTable.close();

}
```

This is only an excerpt, check the HBaseClient.java file

Graph data model

- Uses graph structures
  - Nodes are the entities and have a set of attributes
  - Edges are the relationships between the entities
    - E.g.: an author writes a book
  - Edges can be directed or undirected
  - Nodes and edges also have individual properties consisting of key-value pairs



Graph data model

- Powerful data model
  - Differently from other types of NoSQL stores, it concerns itself with relationships
  - Focus on visual representation of information (more human-friendly than other NoSQL stores)
  - Other types of NoSQL stores are poor for interconnected data

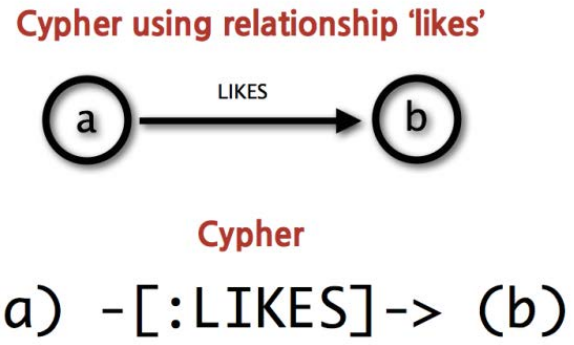
- Cons:
  - Sharding: data partitioning is difficult
  - Horizontal scalability
    - When related nodes are stored on different servers, traversing multiple servers is not performance-efficient
  - Requires rewiring your brain

Suitable use cases for graph databases

- Good for applications where you need to model entities and relationships between them
  - Social networking applications
  - Pattern recognition
  - Dependency analysis
  - Recommendation systems
  - Solving path finding problems raised in navigation systems
  - …
- Good for applications in which the focus is on querying for relationships between entities and analyzing relationships
  - Computing relationships and querying related entities is simpler and faster than in RDBMS

Neo4j: data model

- A graph records data in nodes and relationships
- Nodes are often used to represent entities
  - A node can have properties, relationships, and can also be labeled with one or more labels
  - Note that a node can have relationships to itself
- Relationships organize nodes by connecting them
  - A relationship connects two nodes; a start node and an end node
  - A relationship can have properties



Neo4j: data model

- Properties (both nodes and relationships) can be of different type:
  - Numeric values
  - String values
  - Boolean values
  - Lists of any other type of value
- Labels assign roles or types to nodes
  - A label is a named graph construct that is used to group nodes into sets
  - All nodes labeled with the same label belong to the same set
  - Labels can be added and removed at runtime
  - A node can have multiple labels

Neo4j: Cypher

- A traversal navigates through a graph to find paths;
  - starts from starting nodes to related nodes, finding answers to questions
- Cypher provides a declarative way to query the graph powered by traversals and other techniques
- A path is one or more nodes with connecting relationships, typically retrieved as a query or traversal result

- Cypher: is a textual declarative query language
  - It uses a form of ASCII art to represent graph-related patterns

Cypher

```
(a) -[:LIKES]-> (b)
```

Hands-on Neo4j
(Docker image)

Neo4j with Dockers

- We use the official neo4j container

```
$ docker pull neo4j:3.0
```

- Create a container with Neo4j and forward its ports

```
$ docker run
      --publish=7474:7474
      --publish=7687:7687
      --volume=$HOME/neo4j/data:/data
      neo4j:3.0
```

- We will interact with Neo4j using its webUI

```
http://localhost:7474
```

Cypher syntax

- Cypher uses a pair of parentheses (usually containing a text string) to represent a node

```
(varname:Label { p_name: p_value, ... } )
```

  - () represents a node
  - `varname` (optional) assigns a name to the node that can be used elsewhere within a single statement.
  - the `Label` (prefixed with a colon ":") declares the node's type (or label).
  - the node's properties are represented as a list of key/value pairs, enclosed within a pair of braces

Cypher syntax

- Cypher uses a pair of dashes (--) to represent an undirected relationship. Directed relationships have an arrowhead at one end (<--, -->).
  - It is possible to create only directed relationship, although they can be queried as undirected

```
-[role:ACTED_IN {roles: ["Neo"]}]->
```

Bracketed expressions (`[...]`) are used to add details:

- a variable (e.g., `role`) can be defined, to be used elsewhere in the statement.
- the relationship's type (e.g., `:ACTED_IN`) is analogous to the node's label.
- the properties (e.g., `roles`) are entirely equivalent to node properties.

Cypher syntax

Variables:

To increase modularity and reduce repetition, Cypher allows patterns to be assigned to variables

```
acted_in = (:Person)-[:ACTED_IN]->(:Movie)
```

https://neo4j.com/developer/cypher-query-language/

Cypher syntax: Create

Create a node with label Person and property name with value "you":

```
CREATE (you:Person {name:"You"})
RETURN you
```

Create a more complex structure: add a new node and a new relationship with the existing one

```
MATCH  (you:Person {name:"You"})
CREATE (you)-[like:LIKE]->(neo:Database {name:"Neo4j"})
RETURN you, like, neo
```

Cypher syntax: Find, Update and Remove

Find a node (basic syntax)

```
MATCH (you {name:"You"})-[:FRIEND]->(yourFriends)
RETURN you, yourFriends
```

Update an existing node (similarly, to update a relationship)

```
MATCH (n {property:value})
SET n :NewLabel
RETURN n
```

Remove a property (or a Label) from a node (or relationship)

```
MATCH (b {name: "Bruce Springsteen"})
REMOVE b.nickname RETURN b
```

Cypher syntax: Delete

Delete a node:

```
MATCH (a:ToDel)
DELETE a
```

Note that a node cannot be deleted if it participates in a relationship. To remove also relationships, we need to detach the node, delete it and its relationships:

```
MATCH (b {name: "Bruce Springsteen"})
DETACH DELETE b;
```

Cypher syntax: Read Clauses

These clauses read data from the data store:

- `MATCH` Specify the patterns to search for in the database
- `OPTIONAL MATCH` Specify the patterns to search for in the database while using nulls for missing parts of the pattern
- `WHERE` Adds constraints to the patterns in a `MATCH` or `OPTIONAL MATCH` clause or filter the results of a `WITH` clause
- `START` Find starting points through legacy indexes

Read more: http://neo4j.com/docs/developer-manual/current/cypher/clauses/

Cypher syntax: Write Clauses

These clauses write data to the data store:

- `CREATE` Create nodes and relationships
- `MERGE` Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.
- `ON CREATE` (used with MERGE) it specifies the actions to take if the pattern needs to be created.
- `SET` Update labels on nodes and properties on nodes and relationships.
- `DELETE` Delete graph elements (nodes, relationships or paths).
- `REMOVE` Remove properties and labels from nodes and relationships.

Cypher syntax: General Clauses

These comprise general clauses that work in conjunction with other clauses:

- `RETURN` Defines what to include in the query result set.
- `ORDER BY` A sub-clause following `RETURN` or `WITH`, specifying that the output should be sorted in particular way.
- `LIMIT` Constrains the number of rows in the output.
- `SKIP` Defines from which row to start including the rows in the output
- `WITH` Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.
- `UNION` Combines the result of multiple queries.

Cypher syntax: Operators

Within clauses, we often rely on operators to combine and compare nodes/relationships or access to their properties

General operators:

`DISTINCT`, `.` for property access,

`[]` for dynamic property access

Mathematical operators:

`+`, `-`, `*`, `/`, `%`, `^`

Comparison operators:

`=`, `<>`, `<`, `>`, `<=`, `>=`, `IS NULL`, `IS NOT NULL`

Cypher syntax: Operators

String-specific comparison operators:

`STARTS WITH`, `ENDS WITH`, `CONTAINS`

Boolean operators

`AND`, `OR`, `XOR`, `NOT`

String operators

`+` for concatenation, `=~` for regex matching

List operators

`+` for concatenation,

`IN` to check existence of an element in a list,

`[]` for accessing element(s)

Cypher syntax: Relationship pattern length

Relationship pattern length:

```
(a)-[*2]->(b)
```

It is possible to specify a length (2 in the example) in the relationship description of a pattern.

It can be a variable length:

- `*3..5` (between 3 and 5),
- `*3..` (greater than 3),
- `*..5` (less than 5),
- `*` (any length)

Read more: http://neo4j.com/docs/developer-manual/current/cypher/functions/

Cypher syntax: Relationship pattern

Relationship pattern:

- nodes and relationship expressions are the building blocks for more complex patterns;
- patterns can be written continuously or separated with commas

Examples:

- friend-of-a-friend:

```
(user)-[:KNOWS]-(friend)-[:KNOWS]-(foaf)
```

- shortest path:

```
path = shortestPath( (user)-[:KNOWS*..5]-(other) )
```

http://neo4j.com/docs/developer-manual/current/cypher/clauses/match/