Advanced Management of Data

N$_{\text{ot}}$O$_{\text{nly}}$SQL Systems (2)

Focus of DB systems

Traditional relational systems

- centralized data storage
- structured data storage
- data consistency / ACID compliance (a transaction must be atomic, consistent, isolated, and durable)
- aggregation
- powerful query languages

NOSQL systems

- distributed data storage
- flexible data storage
    - schema-less data sets that include structured and semistructured data
- scalability
- high performance (at scale)
- availability through replication

NOSQL Systems

Categories of NOSQL Systems

- Document stores
- Key-Value stores
- Wide column stores
- Graph-Databases
- Hybrid NOSQL systems

Key-Value Stores

Data Model

- every data item (value) must be associated with a unique key
- retrieving the value by supplying the key must be very fast

Value

- can have very different formats for different key-value storage systems:
  - string / array of bytes: the application using the key-value store has to interpret the structure of the data value
  - structured data rows (tuples) similar to relational data
  - semistructured data using a self-describing data format

Amazon Dynamo DB

DynamoDB data model

- uses the concepts of tables, items, and attributes

Table

- holds a collection of self-describing items
- does not have a schema

Item

- consists of a number of (attribute, value) pairs
- a value can be single-valued or multivalued
- can be specified in JSON format

Amazon Dynamo DB

Key (also called primary key)

- must be specified when a table is created
- must exist in every item in the table and can be a
  - single attribute *A*

    A is used to build a hash index on the items in the table.

    The items are not ordered on the value of *A* in storage.
  - pair of attributes *(A, B)*

    A will be used for hashing.

    The *B* values will be used for ordering the records with the same *A* value.

    A table with this type of key can have additional secondary indexes defined on its attributes.

Project Voldemort

Voldemort is an open source key-value system and is based on the techniques of DynamoDB.

Data model

- A collection of (key, value) pairs is kept in a Voldemort store s
- At the basic storage level, both keys and values are arrays of bytes (strings)

Simple basic operations

- s.put(key, value) inserts an item as a key-value pair
- s.delete(key) deletes the item whose key is from s
- value = s.get(key) retrieves the value associated with key
- The application can use these basic operations to build its own requirements, such as joins

Project Voldemort

High-level formatted data values

- The values in the (key, value) items can be specified in JSON
- Other data object formats can also be specified if the application provides a serializer class

Serializer class

- realizes the conversion between the user format and the storage format and has to include
  - operations to convert the user format into an array of bytes for storage as a value
  - operations to convert back an array of bytes retrieved via s.get(key) into the user format
- Voldemort has some built-in serializers for formats other than JSON

Project Voldemort

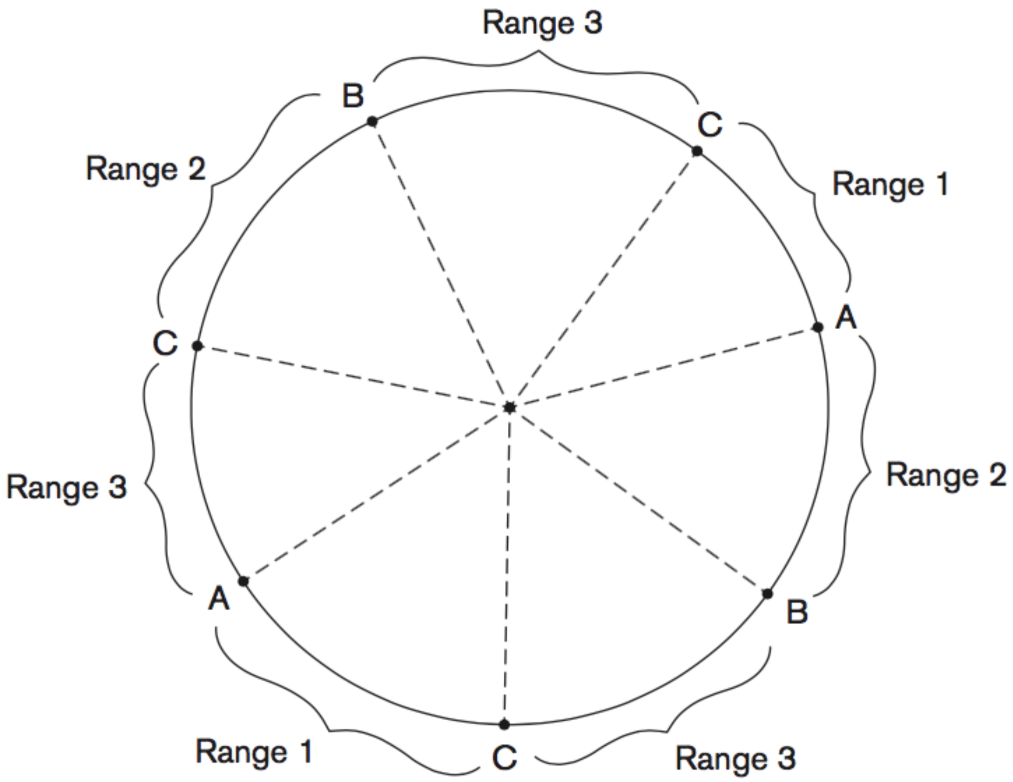
Consistent hashing

- special kind of hashing, which minimizes the number of keys that have to be remapped when the size of a hash table is changed
- assumes that the result of the hash function h(key) is an integer value, usually in the range 0 to $H_{max} = 2^{n-1}$, where n is chosen based on the desired range for the hash values

Visualization

- consider all possible integer hash values 0 to H_{max} to be evenly distributed on a ring
- the nodes in the distributed system are also located on that ring
- usually each node will have several randomly positioned locations on the ring
- a (key, value) item will be stored on the node whose position in the ring follows the position of h(k) in a clockwise direction

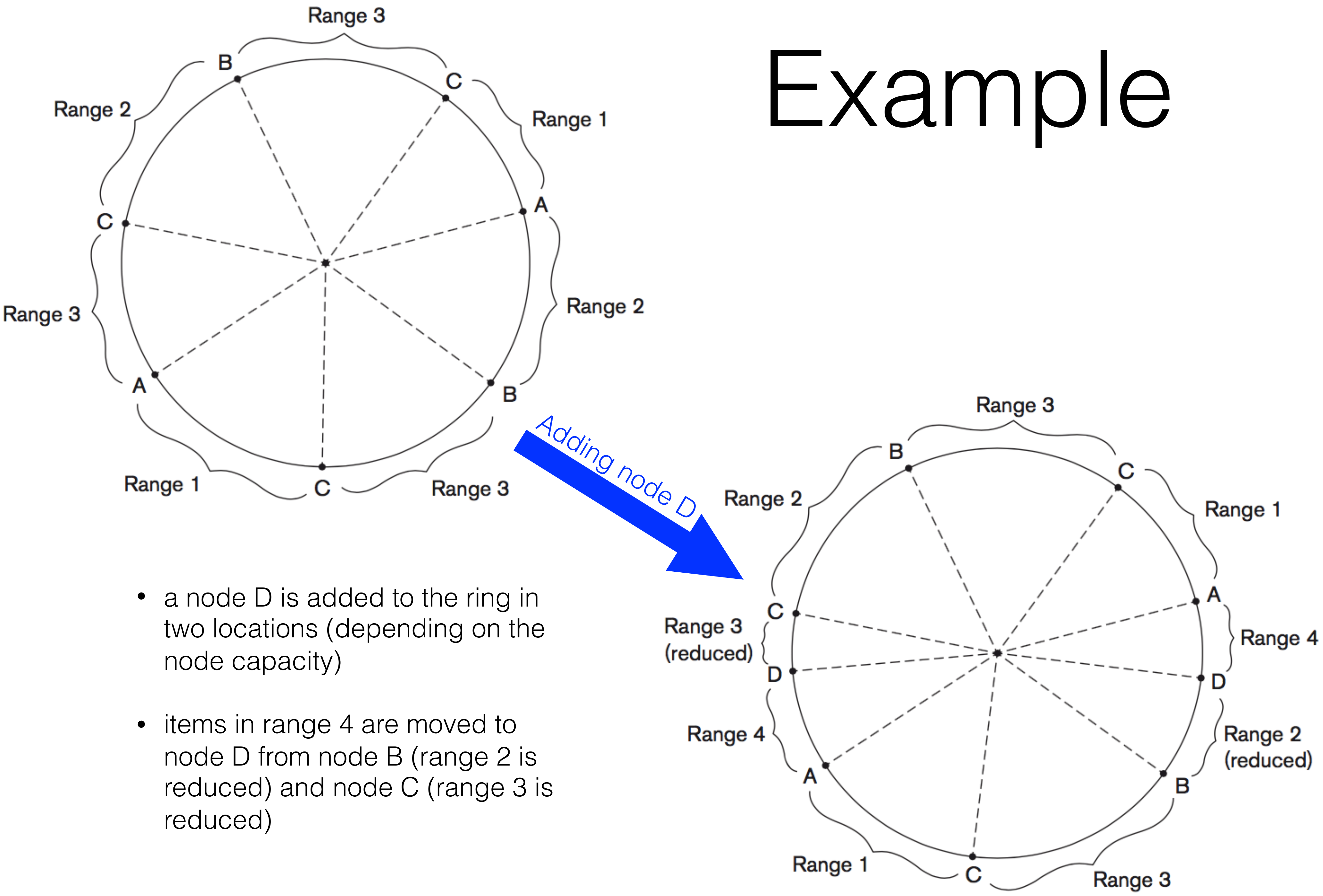
Example



[Elmasri & Navathe]

- given is a distributed cluster with three nodes A, B, and C
- node C has a bigger capacity than nodes A and B.
- two instances each of A and B are placed on the ring
- three instances of node C are placed on the ring, because C has a higher capacity
- the hash values that map to the points in range 1 store their (key, value) items in node A, range 2 in node B, range 3 in node C

Example



- a node D is added to the ring in two locations (depending on the node capacity)
- items in range 4 are moved to node D from node B (range 2 is reduced) and node C (range 3 is reduced)

[Elmasri & Navathe]

Project Voldemort

Potential problem (if items are replicated)

- concurrent write operations are allowed by different processes

  ➔ two or more different values could be associated with the same key at different nodes

Versioning and read repair

- technique to achieve consistency when reading an item
- concurrent writes are allowed, but each write is associated with a *vector clock* value
- if different versions of the same value have been read from different nodes
  - first, the system tries to resolve to a single final value using a vector clock algorithm
  - if this is not possible, all versions are passed back to the application, which is now responsible to select one version and pass a single value to the nodes

Other Key-Value Stores

Redis key-value cache and store

- caches its data in main memory to further improve performance
- offers master-slave replication and high availability
- offers persistence by backing up the cache to disk

Apache Cassandra

- hybrid NOSQL system, offers features from several NOSQL categories:
  - wide column
  - key-value
- is used by Twitter, Reddit, …

NOSQL Systems

Categories of NOSQL Systems

- Document stores
- Key-value stores
- Wide column stores
- Graph-Databases
- Hybrid NOSQL systems

Wide Column Stores

Working Principle

- vertical partitioning - tables are partitioned by column into column families
- each column family is stored in its own files
- versioning of data values is allowed
- the key is multidimensional (in contrast to key-value stores)

Examples

- Google distributed storage system (BigTable)
- Apache Hbase

Hbase

Data Model

- data is organized using the concepts of
  - namespaces
  - tables
  - column families, column qualifiers, columns
  - rows
  - data cells
- data is stored in a self-describing form by associating columns with data values, where data values are strings

Hbase

Tables

- data is stored in named tables as rows

Row

- is self-describing
- has a unique row key

Row key

- string that must have the property that it can be ordered lexicographically
- characters that do not have a lexicographic order in the character set cannot be used as part of a row key

Hbase

Column

- specified by a combination of ColumnFamily:ColumnQualifier

Column Family

- related columns are grouped together and stored in the same file ➔ kind of vertical partitioning
- is named and associated with a table
- statically specified when the table is created and cannot be changed later
- each column family can be associated with many column qualifiers

Column Qualifier

- dynamically specified as new rows are created and inserted into the table
  ➔ self-describing data model

Hbase

Versioning

- Hbase can keep several versions of a data item
- each version is associated with a timestamp

Timestamp

- long integer number that represents the system time when the version was created
- measures the number of milliseconds since timestamp value zero (‘January 1, 1970’)

Hbase

Cell

- holds a basic data item
- the key (address) of a cell is specified by a combination of (table, rowid, columnfamily, columnqualifier, timestamp)
- if timestamp is left out, the latest version of the item is retrieved
- optionally, a default number of versions to be retrieved can be specified

Namespace

- collection of tables that are typically used together by user applications
- corresponds to a database that contains a collection of tables in relational terminology

Hbase - CRUD Operations

Create

creates a new table and specifies one or more column families associated with that table:

```
create <tablename>, <column family>, <column family>, …
```

The `put` operation is used for inserting new data:

```
put <tablename>, <rowid>, <column family>:<column qualifier>, <value>
```

Read

The `scan` operation retrieves all the rows:

```
scan <tablename>
```

The `get` operation is for retrieving the data associated with a single row in a table:

```
get <tablename>,<rowid>
```

Example

Creating a table called EMPLOYEE with three column families:

```
create ‘EMPLOYEE’, ‘Name’, ‘Address’, ‘Details’
```

Inserting row data in the EMPLOYEE table:

```
put ‘EMPLOYEE’, ‘row1’, ‘Name:Fname’, ‘John’
put ‘EMPLOYEE’, ‘row1’, ‘Name:Lname’, ‘Smith’
put ‘EMPLOYEE’, ‘row1’, ‘Name:Nickname’, ‘Johnny’
put ‘EMPLOYEE’, ‘row1’, ‘Details:Job’, ‘Engineer’
put ‘EMPLOYEE’, ‘row1’, ‘Details:Review’, ,Good’

put ‘EMPLOYEE’, ‘row2’, ‘Name:Fname’, ‘Alicia’
put ‘EMPLOYEE’, ‘row2’, ‘Name:Lname’, ‘Zelaya’
put ‘EMPLOYEE’, ‘row2’, ‘Name:MName’, ‘Jennifer’
put ‘EMPLOYEE’, ‘row2’, ‘Details:Job’, ‘DBA’
put ‘EMPLOYEE’, ‘row2’, ‘Details:Supervisor’, ‘James Borg’

put ‘EMPLOYEE’, ‘row3’, ‘Name:Fname’, ‘James’
put ‘EMPLOYEE’, ‘row3’, ‘Name:Minit’, ‘E’
put ‘EMPLOYEE’, ‘row3’, ‘Name:Lname’, ‘Borg’
put ‘EMPLOYEE’, ‘row3’, ‘Name:Suffix’, ‘Jr.’
put ‘EMPLOYEE’, ‘row3’, ‘Details:Job’, ‘CEO’
put ‘EMPLOYEE’, ‘row3’, ‘Details:Salary’, ‘1,000,000’
```

different rows can have different self-describing column qualifiers

[Elmasri & Navathe]

Hbase - CRUD Operations

Update

The `put` operation is also used for updating existing data items (inserting new versions):

```
put <tablename>, <rowid>, <column family>:<column qualifier>, <value>
```

Delete

The `deleteall` operation deletes all cells in a row

```
deleteall ‘<table name>’, ,<row>’
```

Complex Operations

Application programs are responsible to implement more complex operations.

Hbase

Region

- each Hbase table is divided into a number of regions
- each region will hold a range of the row keys in the table
- each region will have a number of stores, where each column family is assigned to one store within the region

Region server

Storage node to which regions are assigned for storage

Master server

Node that is responsible for monitoring the region servers and for splitting a table into regions and assigning regions to region servers.

Hbase

Distributed System Concepts

Hbase is built on top of both Hadoop Distributed File System (HDFS) and Zookeeper:

- Apache HDFS is used for distributed file services
- Apache Zookeeper is a distributed hierarchical key-value store and used for
    - managing the naming, distribution, and synchronization of the Hbase data on the distributed Hbase server nodes
    - coordination and replication services

Zookeeper can itself have several replicas on several nodes for availability.

Zookeeper keeps the needed data in main memory to speed access to the master servers and region servers.

NOSQL Systems

Categories of NOSQL Systems

- Document stores
- Key-value stores
- Wide column stores
- Graph-Databases
- Hybrid NOSQL systems

Neo4j

Data Model

- data is organized as a graph, which is a collection of nodes, relationships, and properties

Node

- can contain properties
- can contain labels, which groups nodes with the same label into subsets for querying purposes

Relationship

- is directed, each relationship has a start node and an end node
- can contain properties
- has a relationship type, which helps to identify similar relationship types for querying purposes

Neo4j

Properties

- store the data items associated with nodes and relationships as list of key-value pairs

Path

Related nodes can be found by traversing the relationships using path expressions.

A path specifies a traversal of part of the graph.

It is typically used as part of a query to specify a pattern, where the query will retrieve from the graph data that matches the pattern.

A path is typically specified by a start node, followed by one or more relationships, leading to one or more end nodes that satisfy the pattern.

Neo4j

Schema

- optional
- features related to schema creation involve creating indexes and constraints based on the labels and properties

Node identifiers

Neo4j automatically creates an internal unique system-defined identifier for each node.

Indexing

To retrieve individual nodes using other properties of the nodes efficiently, the user can create indexes for the collection of nodes that have a particular label.

Typically, one or more of the properties of the nodes in that collection can be indexed.

Neo4j

The Cypher Query Language

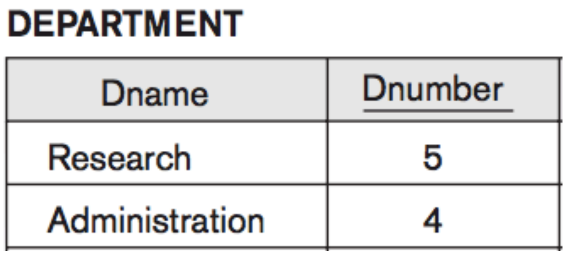
- high-level query language
- nodes and relationships can be created, updated and deleted
- nodes and relationships can be found by specifying patterns ➔ declarative access
- a query is made up of clauses
- the result from one clause can be the input to the next clause

Neo4j

Basic Cypher clauses

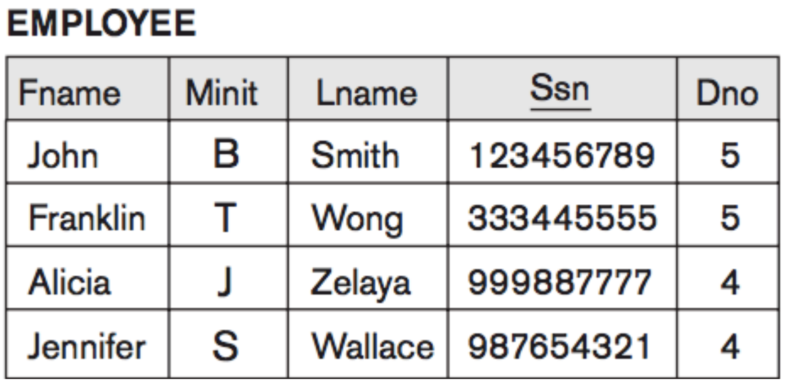
• Create a node:	`CREATE`	`<node, optional labels and properties>`
• Create a relationship:	`CREATE`	`<relationship, relationship type and optional properties>`
• Delete nodes / relationships:	`DELETE`	`<nodes or relationships>`
• Specify property values and labels:	`SET`	`<property values and labels>`
• Remove property values and labels:	`REMOVE`	`<property values and labels>`

Example



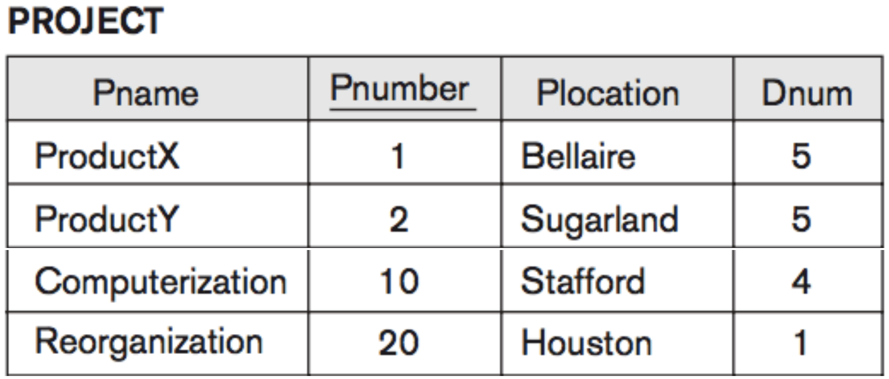
DEPARTMENT

Dname	Dnumber
Research	5
Administration	4



EMPLOYEE

Fname	Minit	Lname	Ssn	Dno
John	B	Smith	123456789	5
Franklin	T	Wong	333445555	5
Alicia	J	Zelaya	999887777	4
Jennifer	S	Wallace	987654321	4



PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1

Creating nodes:

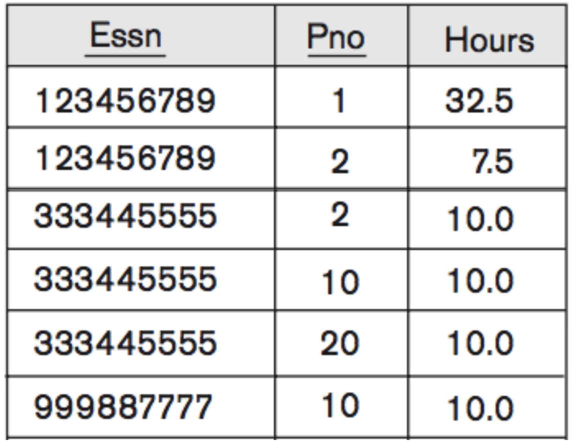
labels

```
CREATE (e1: EMPLOYEE, {Empid: ‘1’, Lname: ‘Smith’, Fname: ‘John’, Minit: ‘B’})
CREATE (e2: EMPLOYEE, {Empid: ‘2’, Lname: ‘Wong’,  Fname: ‘Franklin’})
CREATE (e3: EMPLOYEE, {Empid: ‘3’, Lname: ‘Zelaya’,  Fname: ‘Alicia’})
CREATE (e4: EMPLOYEE, {Empid: ‘4’, Lname: ‘Wallace’, Fname: ‘Jen’, Minit: ’S’})

CREATE (d1: DEPARTMENT, {Dno: ‘5’, Dname: ‘Research’})
CREATE (d2: DEPARTMENT, {Dno: ‘4’, Dname: ,Administration’})

CREATE (p1: PROJECT, {Pno: ‘1’,  Pname: ‘ProductX’})
CREATE (p2: PROJECT, {Pno: ‘2’,  Pname: ‘ProductY’})
CREATE (p3: PROJECT, {Pno: ‘10’, Pname: ‘Computerization’})
CREATE (p4: PROJECT, {Pno: ‘20’, Pname: ,Reorganization’})
```

based on [Elmasri & Navathe]

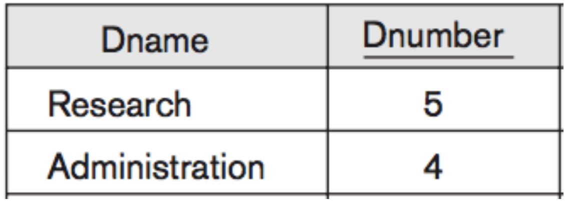
Example

WORKS_ON



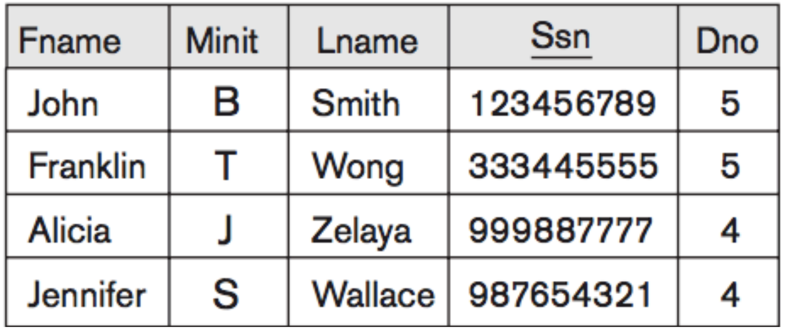
Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
333445555	2	10.0
333445555	10	10.0
333445555	20	10.0
999887777	10	10.0

DEPARTMENT



Dname	Dnumber
Research	5
Administration	4

EMPLOYEE



Fname	Minit	Lname	Ssn	Dno
John	B	Smith	123456789	5
Franklin	T	Wong	333445555	5
Alicia	J	Zelaya	999887777	4
Jennifer	S	Wallace	987654321	4

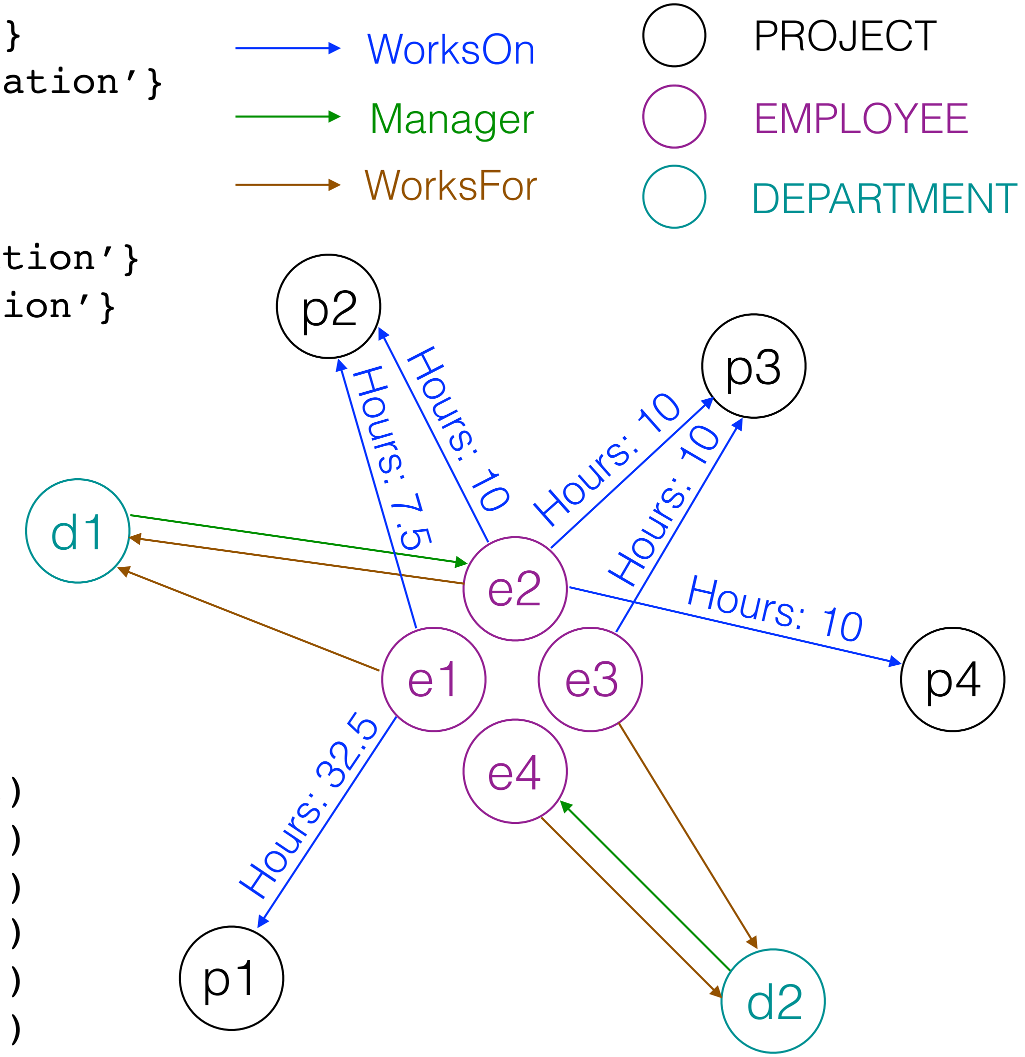
Creating relationships:

```
CREATE (e1) — [ : WorksFor ] —> (d1)
CREATE (e2) — [ : WorksFor ] —> (d1)
CREATE (e3) — [ : WorksFor ] —> (d2)
CREATE (e4) — [ : WorksFor ] —> (d2)

CREATE (d1) — [ : Manager ] —> (e2)
CREATE (d2) — [ : Manager ] —> (e4)

CREATE (e1) — [ : WorksOn, {Hours: ‘32.5’} ] —> (p1)
CREATE (e1) — [ : WorksOn, {Hours: ‘7.5’ } ] —> (p2)
CREATE (e2) — [ : WorksOn, {Hours: ‘10.0’} ] —> (p2)
CREATE (e2) — [ : WorksOn, {Hours: ’10.0’} ] —> (p3)
CREATE (e2) — [ : WorksOn, {Hours: ‘10.0’} ] —> (p4)
CREATE (e3) — [ : WorksOn, {Hours: ’10.0’} ] —> (p3)
```

based on [Elmasri & Navathe]

Example

```
e1: EMPLOYEE, {Empid: ‘1’, Lname: ‘Smith’, Fname: ‘John’, Minit: ‘B’}
e2: EMPLOYEE, {Empid: ‘2’, Lname: ‘Wong’,  Fname: ‘Franklin’}
e3: EMPLOYEE, {Empid: ‘3’, Lname: ‘Zelaya’,  Fname: ‘Alicia’}
e4: EMPLOYEE, {Empid: ‘4’, Lname: ‘Wallace’, Fname: ‘Jen’, Minit: ’S’}

d1: DEPARTMENT, {Dno: ‘5’, Dname: ‘Research’}
d2: DEPARTMENT, {Dno: ‘4’, Dname: ‚Administration’}

p1: PROJECT, {Pno: ‘1’,  Pname: ‘ProductX’}
p2: PROJECT, {Pno: ‘2’,  Pname: ‘ProductY’}
p3: PROJECT, {Pno: ‘10’, Pname: ‘Computerization’}
p4: PROJECT, {Pno: ‘20’, Pname: ‚Reorganization’}


(e1) — [ : WorksFor ] –> (d1)
(e2) — [ : WorksFor ] –> (d1)
(e3) — [ : WorksFor ] –> (d2)
(e4) — [ : WorksFor ] –> (d2)

(d1) — [ : Manager ] –> (e2)
(d2) — [ : Manager ] –> (e4)

(e1) — [ : WorksOn, {Hours: ‘32.5’} ] –> (p1)
(e1) — [ : WorksOn, {Hours: ‘7.5’ } ] –> (p2)
(e2) — [ : WorksOn, {Hours: ‘10.0’} ] –> (p2)
(e2) — [ : WorksOn, {Hours: ‘10.0’} ] –> (p3)
(e2) — [ : WorksOn, {Hours: ‘10.0’} ] –> (p4)
(e3) — [ : WorksOn, {Hours: ’10.0’} ] –> (p3)
```



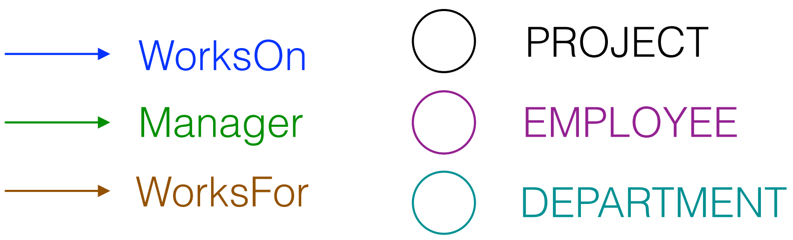
based on [Elmasri & Navathe]

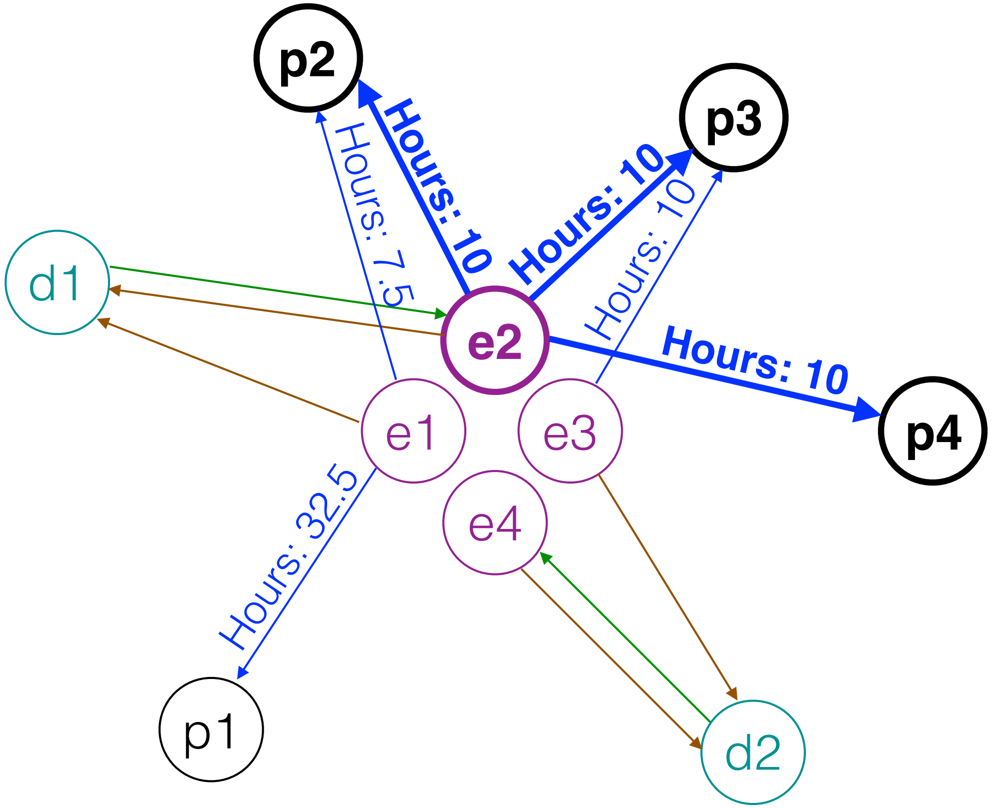
Neo4j

Basic query clauses of Cypher

MATCH <pattern>	finds nodes and relationships that match a pattern
WITH <specifications>	specifies aggregates and other query variables
WHERE <condition>	specifies conditions on the data to be retrieved
RETURN <data>	specifies the data to be returned
ORDER BY <data>	orders the data to be returned
LIMIT <max number>	limits the number of returned data items

Example





Cypher query

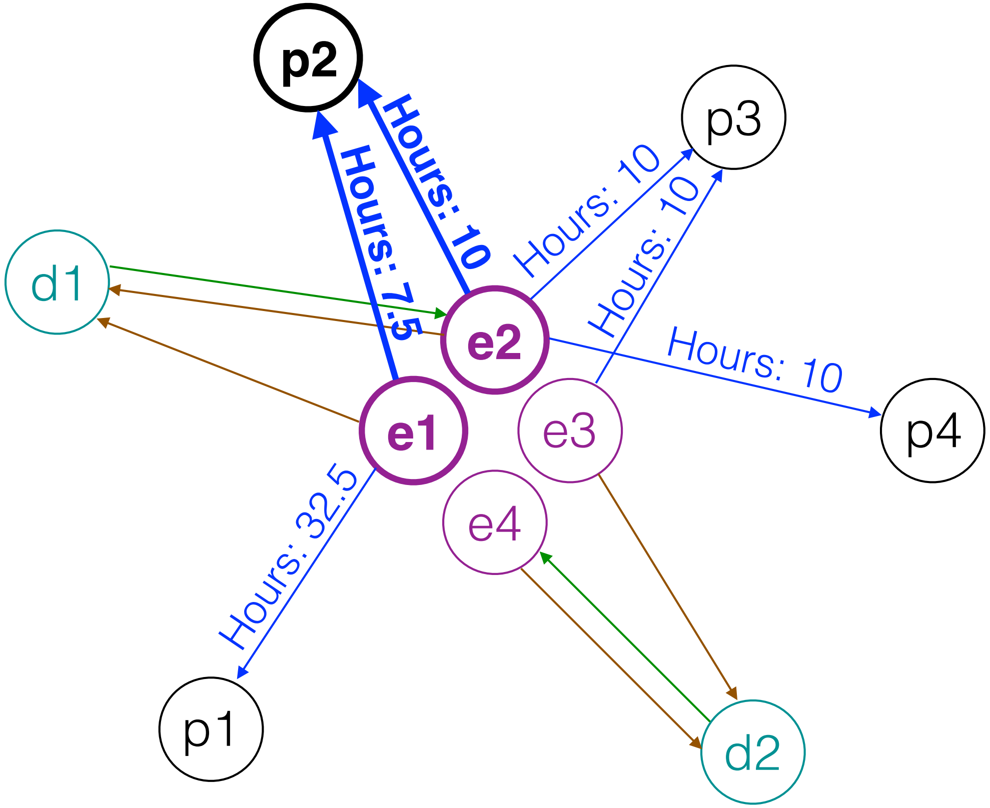
```
MATCH (e: EMPLOYEE {Empid: ‘2’}) — [ w: WorksOn ] → (p)
RETURN e.Fname, w.Hours, p.Pname
```

<u>Result:</u>

```
Franklin, 10.0, ProductY
Franklin, 10.0, Computerization
Franklin, 10.0, Reorganization
```

based on [Elmasri & Navathe]

Example



→ WorksOn

→ Manager

→ WorksFor

○ PROJECT

○ EMPLOYEE

○ DEPARTMENT

Cypher query

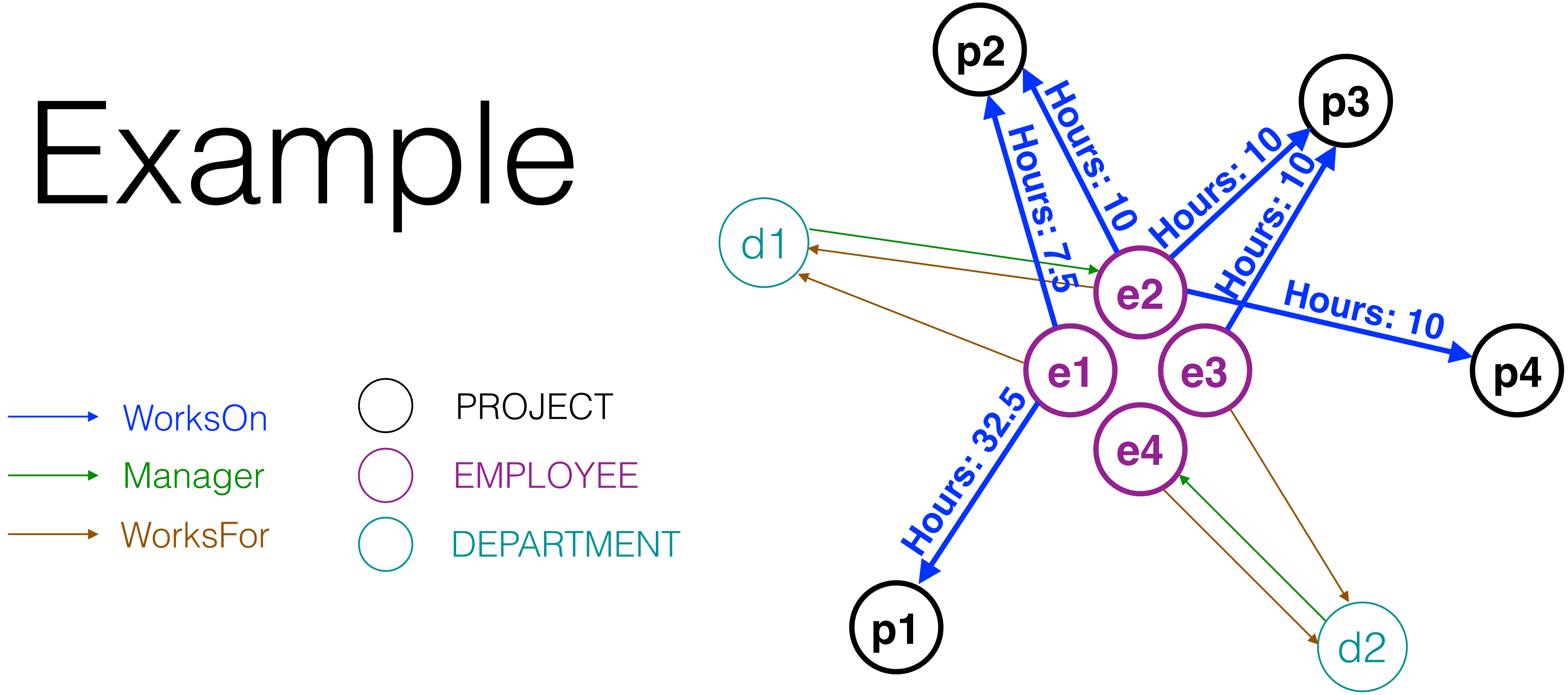
```
MATCH (e ) — [ w: WorksOn ] → (p: PROJECT {Pno: 2})
RETURN p.Pname, e.Lname , w.Hours
```

<u>Result:</u>

```
ProductY, Smith, 7.5
ProductY, Wong, 10.0
```

based on [Elmasri & Navathe]

Example



Cypher query

```
MATCH (e) — [ w: WorksOn ] → (p)
RETURN e.Lname , w.Hours, p.Pname
ORDER BY e.Lname
```

Result:

```
Smith, 32.5, ProductX
Smith, 7.5, ProductY
Wong, 10.0, ProductY
Wong, 10.0, Computerization
Wong, 10.0, Reorganization
Zelaya, 10.0, Computerization
```

based on [Elmasri & Navathe]

Example

→ WorksOn

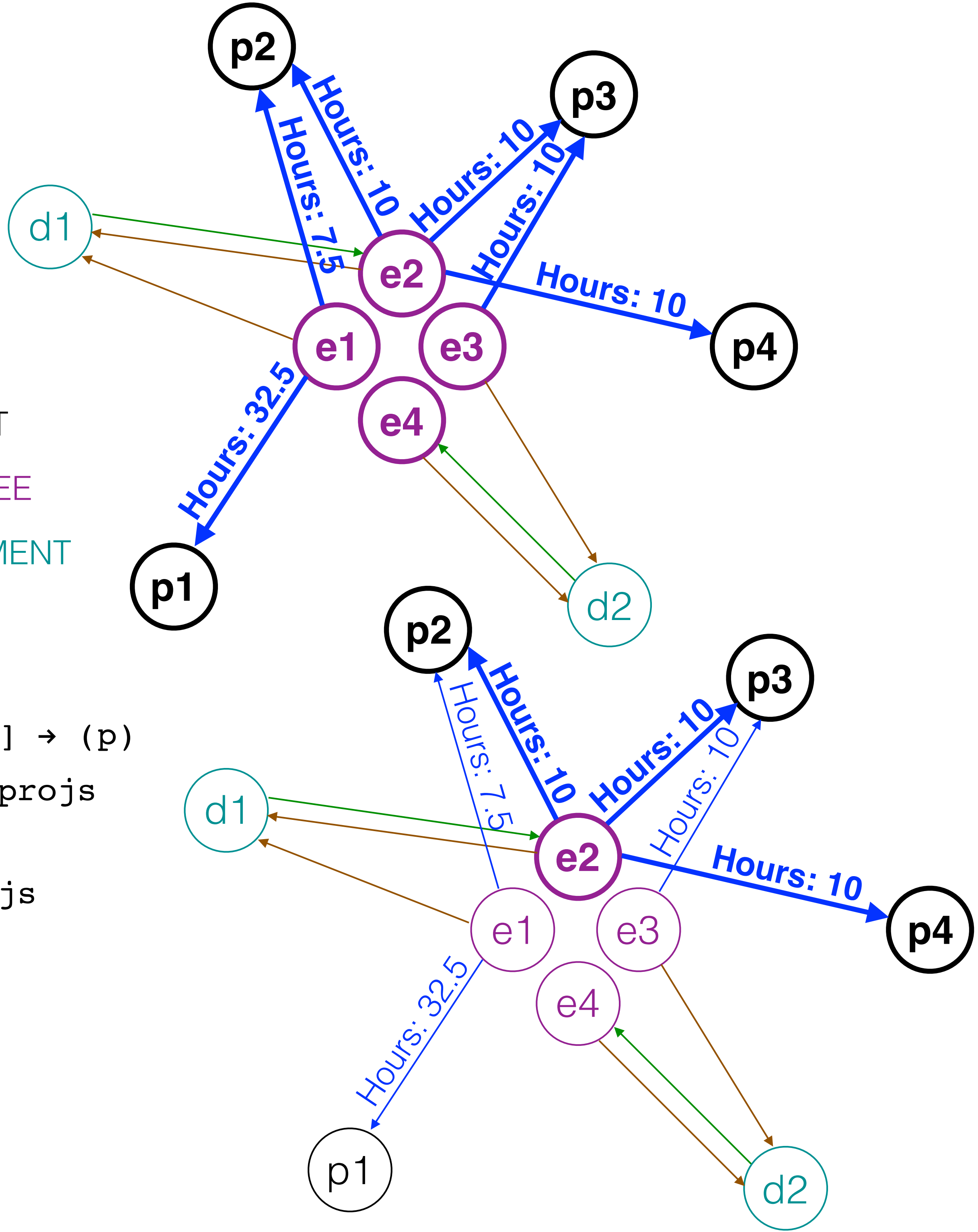
→ Manager

→ WorksFor

○ PROJECT

○ EMPLOYEE

○ DEPARTMENT



Cypher query

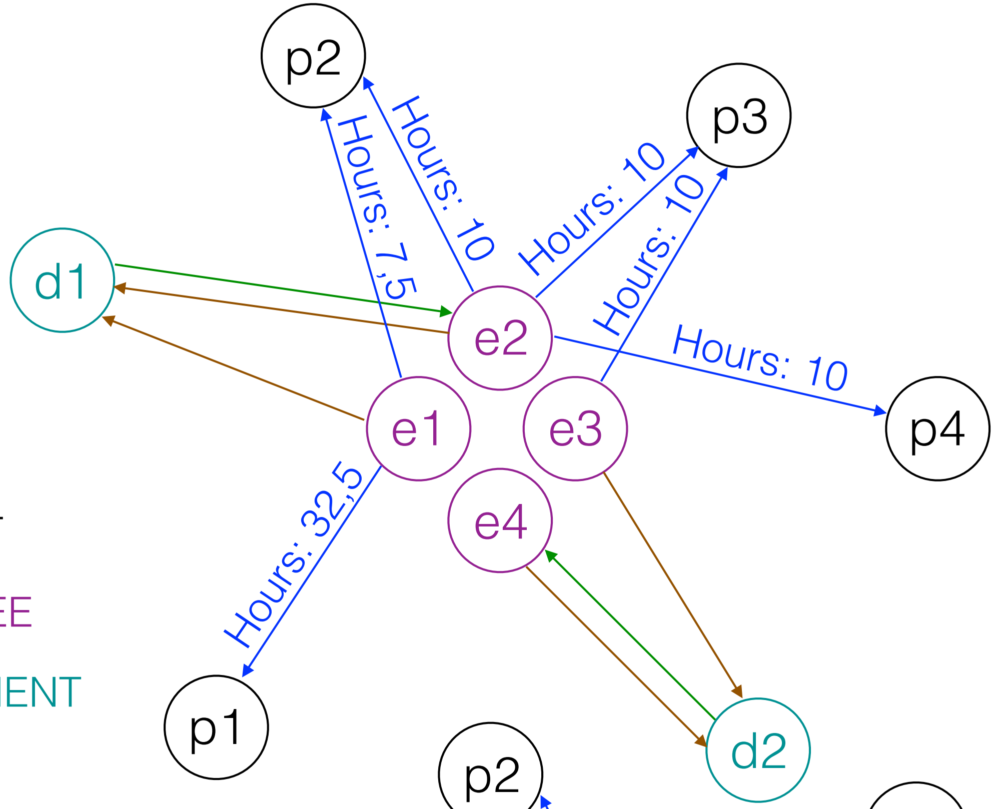
```
MATCH (e) — [ w: WorksOn ] → (p)
WITH e, COUNT(p) AS numOfprojs
WHERE numOfprojs > 2
RETURN e.Lname , numOfprojs
ORDER BY numOfprojs
```

Result:

```
Wong, 3
```

based on [Elmasri & Navathe]

Example



WorksOn

Manager

WorksFor

PROJECT

EMPLOYEE

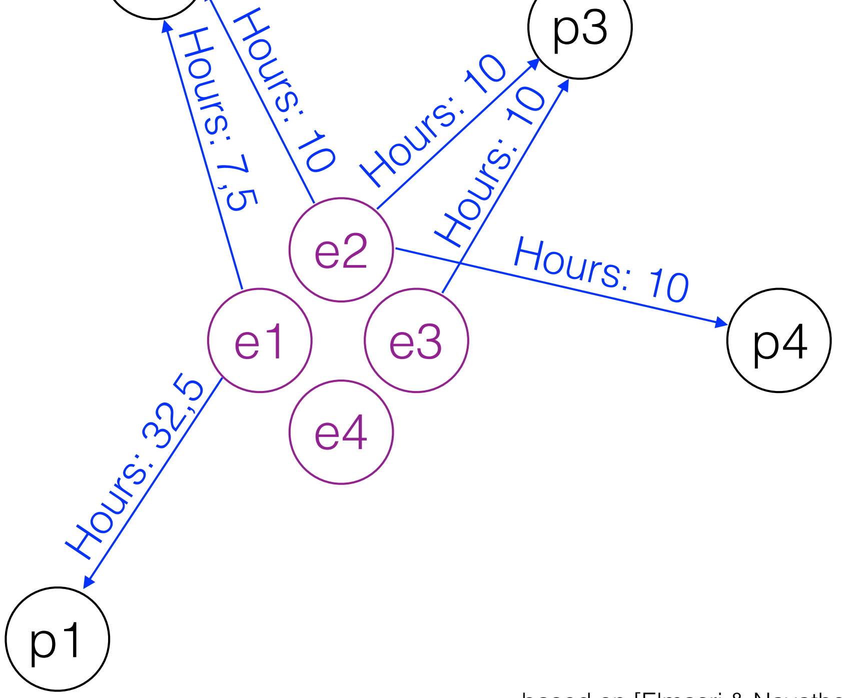
DEPARTMENT

Cypher query

```
MATCH (e) — [ w: WorksOn ] → (p)
RETURN e , w, p
```

<u>Result:</u>

is a graph



based on [Elmasri & Navathe]

Neo4j

Interfaces

- supports other interfaces that can be used to create, retrieve, and update nodes and relationships in a graph database
- supports ACID properties
- has a graph visualization interface, so that a subset of the nodes and edges in a database graph can be displayed as a graph

Master-slave replication

- Neo4j can be configured on a cluster of distributed system nodes, where one node is designated the master node
- the data and indexes are fully replicated on each node in the cluster
- various ways of synchronizing the data between master and slave nodes can be configured in the distributed cluster

Ranking of DB Systems

Popularity of various DB Systems in January 2018 based on Google Trends, number of technical discussions, job offers, social network relevance, … ➔ http://db-engines.com/en/ranking

1. Oracle (ORDBMS)
2. MySQL (RDBMS)
3. Microsoft SQL Server (ORDBMS)
4. PostgreSQL (ORDBMS)
5. MongoDB (Document Store)
6. IBM DB2 (ORDBMS)
7. Microsoft Access (RDBMS)
8. Apache Cassandra (Wide Column Store, Key-Value Store)
9. Redis (Key-Value Store)
10. …
11. SQLite (RDBMS)

…

15. Apache Hbase (Wide Column Store)

…

21. Neo4j (Graph Database)
22. Amazon DynamoDB

…